Kurzfassung

Die vorliegende Belegarbeit untersucht den prototypischen Entwurf eines autonomen Lagerroboters, der in der Lage ist, ein vorab definiertes Produkt autonom aus einem Lager zu entnehmen. Hierzu wurde eine modulare Systemarchitektur entwickelt, die auf ROS2 als Middleware basiert und spezifische Nodes, Topics, Services und Actions integriert, um die Funktionalitäten des Roboters zu realisieren. Im Rahmen der Arbeit wurden die Konzepte des *Simultaneous Localization and Mapping (SLAM)* zur Kartenerstellung und der *Navigation* mit den ROS2-Paketen `slam_toolbox` und `Nav2` umgesetzt, um eine Umgebungskarte zu erstellen und den Roboter erfolgreich zu navigieren.
Weitere Schwerpunkte der Arbeit lagen auf der Durchführung praktischer Tests, der Dokumentation von Ergebnissen sowie der Bereitstellung von und Anleitungen und Problemlösungen zur Nutzung der Technologien in der Lehre und als Demonstrator. Abschließend wurden Verbesserungspotenziale für eine zukünftige Weiterentwicklung des Projekts aufgezeigt, sowie die Erweiterung der Betriebsumgebung für realistischere Testszenarien.

Stichwörter
Robitk, ROS2, SLAM, Navigation

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung und Zielsetzung

In den letzten Jahren konnten durch Innovationen im Bereich der künstlichen Intelligenz und die kontinuierliche Steigerung der Rechenleistung moderner Computer erhebliche Fortschritte im Bereich der Robotik erzielt werden. In diesem Zuge wurden erfolgreich Anwendungen entwickelt, die von der Massenproduktion in der Industrie über den Einsatz in Katastrophengebieten bis hin zu alltäglichen Haushaltsaufgaben wie Wischen oder Aufräumen reichen. [1]
Im Kontext dieser Entwicklungen verfolgt die vorliegende Belegarbeit das Ziel, den prototypischen Entwurf eines autonomen Lagerroboters zu analysieren und zu implementieren. Der Roboter soll in der Lage sein, ein vorab definiertes Produkt autonom aus einem Lager zu entnehmen, das mit weiteren Produkten bestückt ist.
Des Weiteren zielt diese Arbeit darauf ab, eine fundierte Grundlage für die Anwendung von *Simultaneous Localization and Mapping (SLAM)* zur Kartenerstellung und Lokalisierung sowie der *Navigation* mit eigens erstellten Karten bereitzustellen. Beide Konzepte sollen Studierenden und Lehrenden so einfacher zugänglich gemacht werden, um ihnen einen praxisorientierten Einstieg zu ermöglichen und auftretende Fragestellungen zu klären. Die verwendeten Technologien umfassen den *TurtleBot3*, den *OpenManipulator*, *ROS2* sowie die relevanten Pakete `slam_toolbox` und `Nav2`.
Das folgende Kapitel befasst sich mit den theoretischen Grundlagen der eingesetzten Technologien. Es schließen sich Abschnitte an, die das entwickelte Konzept sowie die Implementierung im Detail beschreiben. Abschließend erfolgt eine Evaluierung der Funktionsweise und Leistungsfähigkeit des Prototyps.

2 Theoretische Grundlagen

Anwendungen in der Robotik unterscheiden sich in vielerlei Hinsicht von anderen Softwareanwendungen. Sie müssen komplexe Ziele erreichen und dabei mit einer dynamischen Umgebung interagieren. Oft muss dabei auf unvorhergesehene Veränderungen reagiert und mit Unsicherheiten sowie Störungen umgegangen werden. Diese Anforderungen beeinflussen maßgeblich die Auswahl der Technologie, Architektur sowie das Design von Robotern. [2]
In diesem Kapitel wird daher eingehend erläutert, welche spezifischen Technologien für den Entwurf des Prototyps ausgewählt wurden, welche Funktionen sie erfüllen und wie ihre Funktionsweise in den Entwicklungsprozess integriert wird. Besonderes Augenmerk liegt dabei auf den Technologien ROS2, dem Turtlebot3, dem OpenManipulator sowie den Paketen `Nav2` und `slam_toolbox`.

2.1 ROS2

ROS 2 (Robot Operating System 2) ist im wesentlichen eine Publisher-Subscriber basierte Middleware, welche speziell für die Entwicklung von Robotersystemen entwickelt wurde [3]. Es stellt die Weiterentwicklung des ursprünglichen *Robot Operating Systems (ROS)* dar. Im Vergleich zu seinem Vorgänger wurden insbesondere Verbesserungen in den Bereichen Sicherheit und Zuverlässigkeit vorgenommen sowie die Unterstützung für groß angelegte eingebettete Systeme verbessert. ROS 2 wird in einer Vielzahl von Anwendungsbereichen eingesetzt, darunter kommerzielle Robotik, wie etwa bei OTTO Motors und Clearpath Robotics, sowie in Forschung und Bildung oder im Bereich der autonomen Systeme. [4]
ROS 2 basiert auf dem Data Distribution Service (DDS), einem offenen Kommunikationsstandard, der in kritischen Infrastrukturen wie Militär und Luftfahrt eingesetzt wird. [4]
Das Herzstück von ROS 2 stellen die sogenannten *Nodes* dar. Nodes sind kleine, spezialiserte Programme, welche in einem eigenen Prozess laufen und mit anderen Nodes über *Interfaces* kommunizieren können. [3]
Die einfachste Art der Kommunikation zwischen Nodes läuft über *Topics*. Topics sind Kommunikationskanäle, die den asynchronen Austausch von Daten mittels eines Publish/-Subscribe-Modells ermöglichen. Innerhalb dieses Modells kann eine Node (z. B. Node 1) Daten, wie etwa Sensormesswerte, auf einem Topic veröffentlichen. Eine andere Node (z. B. Node 2) kann diesen Topic abonnieren, um die von Node 1 veröffentlichten Daten zu empfangen und weiterzuverarbeiten. Die übertragenen Daten sind streng typisiert und werden als *Messages* bezeichnet. [3]
Weiterhin können Nodes über *Services* miteinander kommunizieren. Dabei handelt es sich im Wesentlichen um *Remote Procedure Calls*, bei denen eine *Client-Node* eine *Server-Node* aufruft, um beispielsweise eine Berechnung durchzuführen. In der Regel wartet die Client-Node, bis die Server-Node die angeforderte Berechnung abgeschlossen hat. Aufgrund dieses synchronen Kommunikationsmodells sind Services für zeitkritische oder

länger andauernde Aufgaben weniger geeignet [3].
Für länger laufende Aufgaben, wie etwa die Navigation eines Roboters zu einem bestimmten Zielort, kommen *Actions* zum Einsatz. Ähnlich wie bei Services initiiert hier eine Client-Node einen Remote Procedure Call auf einer Server-Node. Im Gegensatz zu Services erfolgt die Verarbeitung jedoch asynchron, wobei die Möglichkeit besteht, dass der Server während der Ausführung kontinuierlich Feedback über den Fortschritt der Aufgabe an den Client übermittelt. Die Definition von Actions erfolgt meist in separaten Interface-Packages, wobei eine Action aus folgende Bestandteilen besteht: *Goal*, *Result* und *Feedback*. Bei diesen einzelnen Bestandteilen einer Action handelt es sich selbst nur um Messages, welche Parameter eines definierten Typs beinhalten. Als Beispiel können die selbst definierten Actions für dieses Projekt im Anhang im Abschnitt 7.1 betrachtet werden. [3]

2.2 TurtleBot3

Der TurtleBot3 ist ein auf ROS basierender mobiler Roboter, der speziell für Lehr- und Bildungszwecke entwickelt wurde. Die für diese Arbeit verwendete Version des TurtleBot3 wurde im Jahr 2017 von ROBOTIS entwickelt, mit dem Ziel, eine kostengünstige und modulare Plattform bereitzustellen, die es ermöglicht, verschiedene Aspekte der Robotik zu erlernen. Darüber hinaus zeichnet sich der TurtleBot3 durch seine Erweiterbarkeit aus, da er problemlos um zusätzliche Sensoren oder andere elektronische Komponenten ergänzt werden kann. [5]
Der in dieser Arbeit eingesetzte TurtleBot3 ist das Modell *Waffle Pi*, das mit einem *Raspberry Pi 4* als *Single Board Computer (SBC)* ausgestattet ist. Der Raspberry Pi 4 ermöglicht die Steuerung des Roboters und verfügt zudem über eine integrierte Kamera. Zu den weiteren zentralen Komponenten des Roboters gehören ein 360° *Laser Distance Sensor (LDS)*, eine *Inertial Measurement Unit (IMU)* mit einem 3-Achsen-Gyroskop sowie -Beschleunigungssensor, und DYNAMIXEL Servomotoren, die die beiden Antriebsräder des Roboters steuern. Der TurtleBot3 wird im weiteren Verlauf der Arbeit vereinfacht als TurtleBot bezeichnet. [5].
Für den prototypischen Einsatz als Lagerrobotor, der in der Lage sein soll, Produkte zu greifen und abzulegen, wurde der TurtleBot um einen mechanischen Greifarm, den *OpenManipulator-X* erweitert. Dieser wurde per USB und *U2D2 Board* mit dem Raspberry Pi verbunden und hinter dem LDS auf dem TurtleBot montiert.

2.3 OpenManipulator-X

Beim OpenManipulator-X (in dieser Arbeit vereinfacht als OpenManipulator bezeichnet) handelt es sich um einen mechanischen Greifarm, der – ebenso wie der TurtleBot – von ROBOTIS entwickelt wurde. Der Arm besteht aus vier DYNAMIXEL *X430* Servomotoren, von denen drei die Gelenke des Arms und einer den zweifingerigen Greifer steuern.
Der OpenManipulator kann sowohl im *Joint Space* als auch im *Task Space* bewegt werden. Im *Joint Space* werden die einzelnen Gelenkwinkel separat eingestellt, um eine gewünschte Ausgangsposition zu erreichen. Im *Task Space* hingegen erfolgt die Bewegung in einem kartesischen Koordinatensystem, wobei die Zielposition durch inverse Kinematik berechnet wird. [6]

Der OpenManipulator ist vollständig mit ROS sowie dem TurtleBot kompatibel [6]. Aufgrund von einem höheren Integrationsaufwand, welcher den Umfang dieses Projekts überstiegen hätte, wird in dieser Arbeit das von ROBOTIS bereitgestellte Paket `turtlebot3_manipulation` für die Zusammenarbeit zwischen TurtleBot und OpenManipulator nicht genutzt. Wie Roboter und Arm dennoch zusammenarbeiten können wird in den Kapiteln 3 und 4 erläutert.

2.4 slam_toolbox

Das ROS2-Package `slam_toolbox` ermöglicht die einfache Integration von *Simultaneous Localization and Mapping (SLAM)*, einer Technik, die es einem Roboter erlaubt, gleichzeitig eine Karte zu erstellen und sich innerhalb dieser zu lokalisieren. Es wurde mit dem Ziel entwickelt, eine Open-Source-Lösung bereitzustellen, die autonomen Systemen das Navigieren in großen und dynamischen Umgebungen ermöglicht. Dabei bietet es eine Vielzahl verschiedener Mapping-Verfahren sowie grundlegende Lokalisierungsfunktionen, die auf einem typischen SBC eines mobilen Roboters ausgeführt werden können. [7]
Im Gegensatz zu klassischen SLAM-Ansätzen, die auf Filter-Methoden basieren, verwenden die Verfahren in `slam_toolbox` graphbasierte SLAM-Methoden. Dabei wird die Karte nicht direkt als Belegungsraster gespeichert, sondern in Form eines Graphen, dem sogenannten *Pose-Graph*[1]. In diesem Graphen repräsentieren die Knoten die Positionen des Roboters (Position und Orientierung), während die Kanten die Übergänge zwischen den Knoten darstellen. Diese Übergänge basieren auf Messwerten des LDS oder anderer Sensoren [8].
Filter-basierte Methoden sind weniger geeignet, um Karten über mehrere Sitzungen hinweg zu erstellen, da die Daten sequenziell verarbeitet und anschließend verworfen werden. Im Gegensatz dazu ermöglichen graphbasierte Methoden die Wiederherstellung und Aktualisierung älterer Roboterpositionen, was die Kartenerstellung flexibler und präziser macht. Darüber hinaus sind graphbasierte Ansätze effizienter und skalierbarer, insbesondere für große Umgebungen. Sie sind besonders vorteilhaft bei Problemen wie *Loop Closures*, bei denen der Roboter einen bereits bekannten Punkt der Karte erneut besucht. Dabei können Odometriefehler reduziert werden, da die Beziehungen zwischen den Positionen im Graphen gespeichert sind und so alte Positionen besser erkannt werden können [8].
Das Package `slam_toolbox` bietet drei Betriebsmodi: *synchrones Mapping*, *asynchrones Mapping* und *reine Lokalisierung*.
Beim synchronen Mapping werden neue Messwerte in einem Buffer gespeichert und in der Reihenfolge ihres Eingangs verarbeitet. Dadurch werden alle verfügbaren Informationen in die Kartenerstellung integriert, was die Qualität der Karte verbessert. Dieser Modus eignet sich besonders für kleinere Räume, ist aber aufgrund der erhöhten Rechenleistung nur bedingt für Echtzeitanwendungen wie der Navigation geeignet.
Im Gegensatz dazu verarbeitet asynchrones Mapping stets die neuesten Messwerte innerhalb eines festgelegten Zeitintervalls. Dies ermöglicht eine schnellere Verarbeitung ohne Verzögerungen, wodurch es ideal für Echtzeitanwendungen ist. Allerdings können hierbei nicht alle Messwerte verarbeitet werden, was die Genauigkeit der Karte beeinträchtigen kann. [7]

[1] Es ist jedoch möglich, die erstellten Karten in `RViz` als Belegungsraster zu exportieren, sodass sie mit Anwendungen kompatibel sind, die ein Belegungsraster zur Kartenverarbeitung erfordern.

Der Modus der reinen Lokalisierung fügt der Karte im Gegensatz zu den Mapping-Verfahren keine dauerhaften Änderungen hinzu. Stattdessen wird ein *Rolling Buffer* verwendet, der neue Messwerte temporär in den Pose-Graph integriert, diese jedoch nach einer gewissen Zeit wieder entfernt. Dadurch kann die Lokalisierung präziser erfolgen, da Veränderungen in der Umgebung berücksichtigt werden. Dieser Mechanismus wird als *Elastic Pose-Graph Deformation* bezeichnet. [7]

2.5 Nav2

`Nav2` ist ein ROS2-Paket und der Nachfolger des ursprünglichen *ROS Navigation Stacks*. Es wurde entwickelt, um eine Open-Source-Lösung für die Navigation verschiedener Robotertypen bereitzustellen, die in einer Vielzahl von Umgebungen und unter unterschiedlichen Bedingungen eingesetzt werden kann. [9]
`Nav2` basiert grundlegend auf sogenannten *Behavior Trees (BT)* und verschiedenen Action-Servern, welche durch Nodes implementiert sind [10]. BTs sind eine Alternative zu *Finite State Machines (FSMs)*, und modellieren den Zustand eines Systems oder Objekts. Bei ihnen handelt es sich um Baumstrukturen, deren Blätter Aktionen repräsentieren, die der Roboter ausführen soll. Um die in der jeweiligen Situation auszuführende Aktion zu bestimmen, wird der Baum von der Wurzel ausgehend durchlaufen, wobei auf jeder Ebene Entscheidungen getroffen werden, bis ein Blatt erreicht wird. Im Vergleich zu FSMs können BTs übersichtlicher und modularer sein, da sie weniger Übergänge erfordern und Aktionen einfacher wiederverwendet werden können. [11]
In `Nav2` wird die Implementierung von BTs durch die Bibliothek `BehaviorTree.CPP V4` realisiert. Dank ihrer Modularität können BTs auch hierarchisch genutzt werden, sodass `Nav2` selbst als Subtree für die Aufgabe der Navigation in einem größeren BT eingebunden werden kann. Eine vereinfachte Version der Architektur von Nav2 wird in Abbildung 2.1 dargestellt. [10]
Die Blätter des BTs selbst sind dabei Action-Clients. Diese senden Anfragen an Action-Server, die daraufhin spezifische Aufgaben im Navigationsprozess ausführen. Die genaue Implementierung der Funktionalität, wie etwa die verwendeten Algorithmen, wird im Action-Server durch Plugins definiert. [10]
Der zentrale Action-Server ist der *BT Navigator Server*. Er steuert den Behavior Tree und koordiniert die kleineren Action-Server. Dieser wird über die `NavigateToPose`-Action angesprochen. Die kleineren Action-Server übernehmen spezifische Teilaufgaben und umfassen unter anderem:

- *Planner Server*: Dieser Server hält eine globale Repräsentation der Umgebung in Form einer *Costmap*[2]. Basierend auf dieser Repräsentation berechnet der Planner Server den globalen Pfad zum Ziel. Die verwendeten Algorithmen können zur Laufzeit durch Plugins angepasst werden. [10]

[2]Eine Costmap ist ein 2D-Raster, bei dem jede Zelle mit einem Kostenwert belegt ist. Der Kostenwert gibt an, ob die Zelle frei, blockiert oder unbekannt ist. In `Nav2` ist die Costmap die zentrale Methode zur Darstellung der Umgebung. Mithilfe von *Costmap Filters* können Bereiche der Karte definiert werden, die ein spezielles Verhalten des Roboters erfordern, z. B. gesperrte Zonen oder Geschwindigkeitsbeschränkungen. [10]

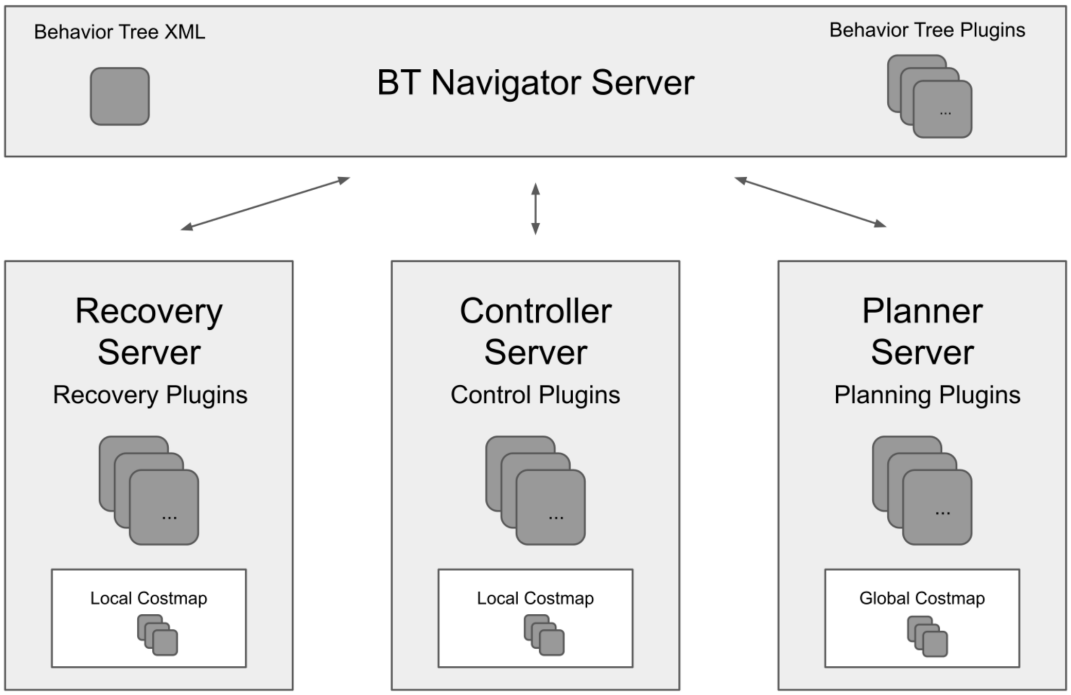


Abbildung 2.1: Vereinfachte Darstellung der *Nav2* Architektur. Die Abbildung wurde aus Macenski u. a. [9] übernommen.

- *Smoother Server*: Dieser Server optimiert den vom Planner Server berechneten Pfad, um ihn für den Roboter besser ausführbar zu machen. Dies beinhaltet unter anderem das Glätten von Kurven, um ruckartige Bewegungen zu vermeiden. Die verwendeten Algorithmen können, wie beim Planner Server, zur Laufzeit durch Plugins konfiguriert werden. Zusätzlich besitzt der Smoother Server einen Subscriber, der die aktuelle globale Costmap bezieht, um redundante Instanzen der Costmap und deren parallele Aktualisierung zu vermeiden. [10]

- *Controller Server*: Der Controller Server ist verantwortlich dafür, den Roboter entlang des geplanten Pfads zu bewegen. Die exakte Funktionsweise kann durch Plugins angepasst werden, wodurch verschiedene Steuerungsalgorithmen verwendet werden können. Wie andere Server besitzt auch der Controller Server einen Subscriber für die aktuelle Costmap, um sicherzustellen, dass Bewegungen stets an die aktuelle Umgebung angepasst sind. [10]

- *Recovery Server*: Dieser Server dient der Ausführung von Recovery-Prozessen, wenn der Roboter in einen fehlerhaften Zustand gerät. Recovery-Verfahren umfassen unter anderem das Löschen der Costmap oder ein Drehmanöver auf der Stelle. Diese Maßnahmen dienen dazu, dynamische Hindernisse aus der Costmap zu entfernen oder neue freie Wege zu finden. [10]

- *Behavior Server*: Dieser Server ermöglicht es, spezifisches Verhalten des Roboters auszuführen, das aus einer Sequenz von Aktionen besteht. Beispiele hierfür sind das Andocken an eine Ladestation oder das Umgehen eines Hindernisses. Die Aktionen des Behavior Servers werden durch Plugins definiert, die jeweils eigene Action-Server bereitstellen können. Auch der Behavior Server bezieht die aktuelle Costmap, um sein Verhalten an die Umgebung anzupassen. [10]

Zusätzlich zu den standardmäßig verfügbaren Action-Servern erlaubt die Modularität von Behavior Trees, benutzerdefinierte Action-Server hinzuzufügen, die spezifische Aufgaben erfüllen. [10]
Damit `Nav2` reibungslos funktioniert, müssen zwei essenzielle Transformationen bereitgestellt werden [10]:

- `odom` zu `base_link`: Diese Transformation beschreibt die Position des Roboters relativ zu seinem Ausgangspunkt (basierend auf der Odometrie).
- `map` zu `odom`: Diese Transformation verbindet die globale Karte mit dem lokalen Koordinatensystem des Roboters.

In dieser Arbeit wird die Transformation von `odom` zu `base_link` durch das Paket `turtlebot3_bringup` und die Transformation von `map` zu `odom` durch `slam_toolbox` bereitgestellt. Diese beiden Transformationen sind unerlässlich, um die Odometrie mit den Kartendaten zu kombinieren und so eine präzise Navigation zu ermöglichen.
Ein weiteres zentrales Feature von `Nav2`, ist das *Waypoint Following*. Es kommt zum Einsatz, wenn der Roboter einer Reihe von definierten Zielpunkten folgen muss, um an jedem Punkt eine Aufgabe auszuführen. Leider war diese Funktion während der praktischen Umsetzung des Projekts noch nicht bekannt und wurde daher nicht integriert. In der Theorie hätte sie jedoch verwendet werden können, um die optimale Route durch das prototypische Warenlager zu berechnen und sicherzustellen, dass der Roboter alle relevanten Punkte erreicht, an denen er Produkte analysieren soll. [10]

3 Konzept

Ziel der Arbeit ist es, den TurtleBot in Verbindung mit dem OpenManipulator zu einem autonomen Lagerroboter zu machen. Er soll dabei automatisch ins Lager navigieren und dort ein spezifisches Produkt auf einer von drei festgelegten Positionen finden und zurückbringen.
Unterschiedliche Produkte werden hierbei durch verschiedenfarbige Bälle dargestellt. Das Lager selbst besteht aus drei Säulen, auf denen die Bälle positioniert werden können.
Für eine erfolgreiche Umsetzung müssen dementsprechend folgende Funktionalitäten implementiert werden:

- *Navigation zum Lager*: Der Roboter muss wissen, wo sich das Lager befindet und an welchen Stellen die Produkte gelagert sind. Die Navigation wird mithilfe von `Nav2` in Kombination mit `slam_toolbox` durchgeführt.
- *Erkennen des Produkts*: Der Roboter muss in der Lage sein, das vorgezeigte Produkt zu erkennen und es von anderen Produkten im Lager unterscheiden. Im Kontext dieser Arbeit bedeutet das, dass die Farbe des Balls korrekt erkannt werden muss. Umgesetzt wird diese Funktionalität mithilfe der Python Bibliothek `OpenCV`.
- *Manipulation des Produkts*: Sobald der Roboter das Lager erreicht, muss er das Produkt aufnehmen können. Zudem sollte er in der Lage sein, es gezielt abzulegen. Diese Aufgabe wird durch den OpenManipulator übernommen.

Um diese Funktionalitäten zu realisieren, ist eine flexible und modulare Architektur erforderlich. Diese wird mithilfe von ROS2 umgesetzt und basiert auf verschiedenen Packages, die jeweils spezifische Aufgaben übernehmen. Jedes Package beinhaltet verschiedene Nodes und stellt eigene Topics, Services und Actions bereit.

3.1 Architekturentwurf

Der Architekturentwurf des Systems ist in Abbildung 3.1 dargestellt. Die Strukturierung der Packages sowie deren Interaktionen basiert auf dem Architekturkonzept von Siebert [12].
Im Folgenden Abschnitt wird genauer auf einzelne Komponenten sowie deren Interaktion eingegangen.

3.1.1 Packages

Zur Implementierung der Architektur kamen sowohl vorgefertigte als auch eigens entwickelte Packages zum Einsatz. Während die vorgefertigten Packages hauptsächlich für die Lokalisierung und Navigation des Roboters verwendet werden, beschreiben die selbst entwickelten Packages dessen spezifisches Verhalten als Lagerroboter. Zu den zentralen vorgefertigten Packages zählen:

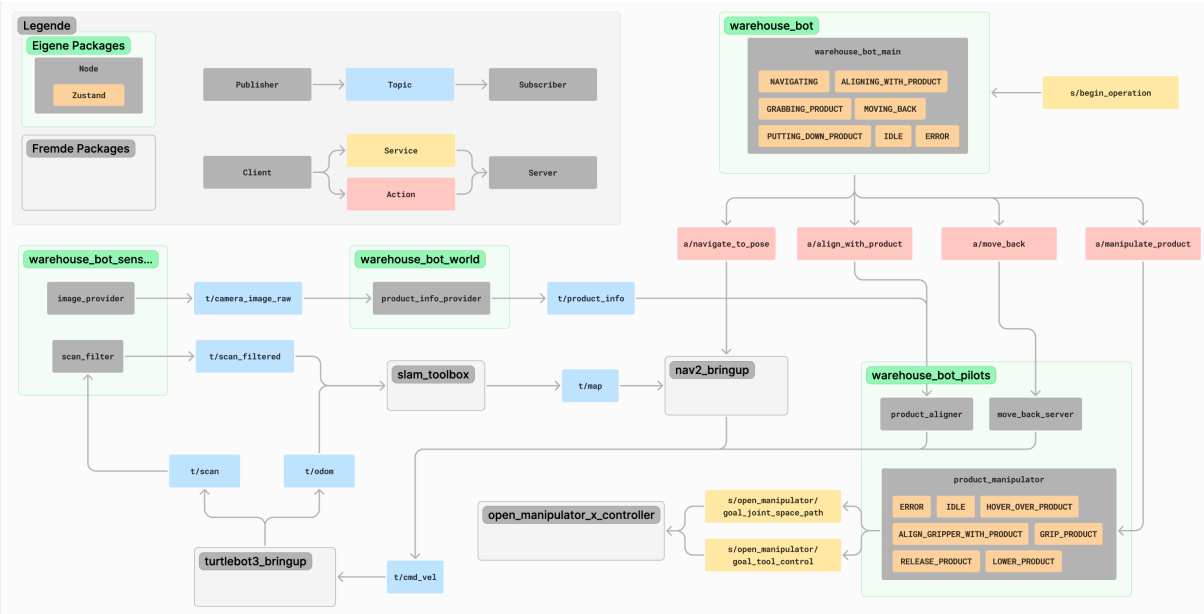


Abbildung 3.1: Architekturentwurf. Der Fokus liegt auf der Interaktion zwischen fremden Packages (hellgrau), eigenen Packages (grün), Nodes (dunkelgrau) mit Zuständen (orange), Topics (dunkelgrau), Services (gelb) und Actions (rot).

- `turtlebot3_bringup`: Enthält die Launch-Dateien und Konfigurationen, die benötigt werden, um den TurtleBot3 in Betrieb zu nehmen. Es stellt wesentliche Funktionalitäten wie die Odometrie, Sensoren und die Steuerung des Antriebs bereit.
- `slam_toolbox`: Ermöglicht die gleichzeitige Kartenerstellung und Lokalisierung mithilfe von SLAM. So kann der Roboter eine Karte der Umgebung erstellen und sich gleichzeitig darin lokalisieren.
- `nav2_bringup`: Dient dazu, den Navigation Stack 2 zu starten. Es enthält die notwendigen Konfigurationen und Launch-Dateien, um Funktionen wie Pfadplanung, Hindernisvermeidung und Zielnavigation bereitzustellen.
- `open_manipulator_x_controller`: Stellt die Steuerungsmechanismen für den OpenManipulator bereit. Es ermöglicht das Ansteuern der Gelenke und des Greifers des Arms, indem es Services für die Bewegung im Joint- und Task Space bereitstellt.

Die eigens entwickelten Packages passen das Verhalten des Roboters speziell an die Arbeit im Lager an. Diese Packages erweitern die Funktionalität des Systems und integrieren die verschiedenen Module nahtlos zu einer Gesamtarchitektur, die es dem Roboter ermöglicht, seine Aufgaben effizient und präzise zu erfüllen.
Zu den eigens entwickelten Packages gehören:

- `warehouse_bot`: Ist für die Koordination der Funktionalitäten wie Navigation, Positionierung und Manipulation zuständig. Es stellt selbst keine Topics, Services oder Actions bereit, sorgt aber dafür, dass der Lagerroboter die richtigen Aktionen zu den korrekten Zeitpunkten ausführt. Zudem befinden sich hier alle Launch- und Kofnigurationsdateien, welche für das Starten des Lagerroboters nötig sind.

- `warehouse_bot_sensors`: Verwaltet die Sensorik des Roboters und stellt entsprechende Topics zur Verfügung.
- `warehouse_bot_world`: Stellt die Teile des Weltmodells dar, welche nicht von `Nav2` oder `slam_toolbox` bereitgestellt werden.
- `warehouse_bot_pilots`: Enthält Nodes, welche die Aktorik des Lagerroboters ansteueren. Dazu zählen die Räder des TurtleBots aber auch der OpenManipulator selbst.

3.1.2 Nodes

In diesem Abschnitt wird lediglich auf selbst entwickelte Nodes eingegangen. Die Analyse und Beschreibung der Nodes aus vorgefertigten Packages wie `Nav2` oder `slam_toolbox` wird an dieser Stelle bewusst nicht vertieft, da diese durch umfangreiche Dokumentation bereits gut nachvollziehbar sind und eine detaillierte Darstellung den Umfang dieser Arbeit überschreiten würde. Stattdessen liegt der Fokus auf den Eigenentwicklungen, die eine zentrale Rolle für die erfolgreiche Implementierung der Gesamtarchitektur spielen.
Zu den wesentlichen eigens entwickelten Nodes zählen:

- `warehouse_bot_main`: Diese Node fungiert als zentrale Steuereinheit des Roboters und ist gleichzeitig Action-Client für vier verschiedene Action-Server. Ihre Funktionsweise basiert auf einer State Machine, die eine klare Zustandsstruktur vorgibt und abhängig vom aktuellen Zustand eine Anfrage an den entsprechenden Action-Server schickt. Ihre Ausführung wird über den `begin_operation`-Service gestartet.
- `image_provider`: Stellt das aktuelle, unbearbeitet Kamerabild auf der Topic `/camera_image_raw` zur Verfügung. Das Bild stammt aus der integrierten Webcam des TurtleBots und wird mit `OpenCV` aufgenommen.
- `scan_filter`: Abonniert den Topic `/scan`, auf dem die aktuellen Daten des LDS veröffentlicht werden. Ihre Aufgabe besteht darin, alle Informationen herauszufiltern, die während der Navigation nicht sinnvoll verarbeitet werden können. Dazu gehören insbesondere Bereiche hinter dem LDS, da sich dort der OpenManipulator befindet, der andernfalls fälschlicherweise als Hindernis erkannt wird. Nach der Bereinigung stellt die Node die gefilterten LDS-Daten auf dem Topic `/scan_filtered` für die weitere Verarbeitung bereit.
- `product_info_provider`: Diese Node verarbeitet die rohen Kameradaten, die über das Topic `/camera_image_raw` empfangen werden. Sie analysiert das Bild, um festzustellen, ob das Produkt im Sichtfeld des Roboters erscheint. Darüber hinaus berechnet sie die Position des Produkts im Bild und ermittelt, wie nah das Produkt zum Roboter ist. Diese Informationen werden anschließend auf dem Topic `/product_info` veröffentlicht.
- `product_aligner`: Hierbei handelt es sich um einen der Action-Server, welcher von der `warehouse_bot_main` Node aufgerufen wird, um den Roboter so vor dem Produkt auszurichten, damit der OpenManipulator auf eine vorher festgelegte Position bewegt werden kann um das Produkt zu greifen. Außerdem erkennt sie, ob sich

überhaupt das korrekte Produkt vor dem Roboter befindet und gibt eine entsprechende Rückmeldung wenn dies nicht der Fall ist. Sie arbeitet mit den Daten, welche über den Topic `/product_info` bereitgestellt werden.

- `product_manipulator`: Ebenfalls ein Action-Server, welcher von `warehouse_bot_main` aufgerufen werden kann, um Produkte zu greifen und abzulegen. Gleichzeitig stellt er einen Service-Client für für die Services `/open_manipulator/goal_joint_space_path` und `/open_manipulator/goal_tool_control` dar, welche durch das Package `open_manipulator_x_controller` bereitgestellt werden.

- `move_back_server`: Ein Action-Server, welcher ebenfalls von `warehouse_bot_main` aufgerufen wird. Er sorgt dafür, dass der Bot sich nach dem Greifen des Produkts von der Säule entfernt um ausreichend Raum für die anschließende Navigation zu gewährleisten.

3.2 Produkterkennung

Die Produkterkennung erfolgt in der `product_info_provider`-Node anhand der rohen Kameradaten vom Topic `/camera_image_raw` mithilfe von `OpenCV`. Hierbei handelt es sich um eine Open-Source-Bilbiothek für Computer-Vision-Aufgaben. Die Bibliothek stellt eine umfassende Sammlung von über 2500 Machine-Learning-Algorithmen bereit, die ein breites Spektrum an Anwendungsbereichen der Bildverarbeitung abdecken. Daher eignet sich `OpenCV` beispielsweise für Gesichtserkennung, Objektklassifikation aber auch für das Arbeiten mit Videos, z.B. um Objekte zu tracken. [13]

3.3 Manipulation des Produkts

Das Greifen und Loslassen des Produkts wird durch die `product_manipulator`-Node in Kombination mit dem OpenManipulator ausgeführt. Der OpenManipulator ist über ein *U2D2-Board* mit dem Raspberry Pi des TurtleBots verbunden. Da jedoch die für die Zusammenarbeit zwischen TurtleBot und OpenManipulator vorgesehenen ROS-Pakete nicht verwendet werden, ist der Manipulator nicht darüber informiert, dass er sich auf dem TurtleBot befindet. Dies führt dazu, dass die absolute Position des Produkts relativ zur Position des TurtleBots nicht genutzt werden kann, um die exakte Greifposition des OpenManipulator zu berechnen.
An dieser Stelle übernimmt die `product_aligner`-Node eine zentrale Rolle. Diese Node ermöglicht es dem TurtleBot, sich basierend auf den Informationen des Topics `/product_info` in einer vorgegebenen Entfernung präzise mittig zum Produkt auszurichten. Sobald die Positionierung erfolgreich abgeschlossen ist, kann der OpenManipulator das Produkt an einer im Vorhinein definierten Position greifen, an welcher sich das Produkt durch die vorherige Positionierung befinden sollte.

3.4 Lokalisierung und Navigation

Die Navigation erfolgt durch das Package `Nav2`, welches mit einer vorher durch `slam_toolbox` erstellten Karte arbeitet. Zudem wird die Lokalisierungsfunktion von `slam_toolbox` genutzt, welche die Transformation von `map` zu `odom` während der Navigation bereitstellt und vom Roboter genutzt wird, um seine aktuelle Position in der Karte zu bestimmen.

4 Implementierung

In diesem Kapitel wird die praktische Umsetzung der Funktionalitäten des Lagerroboters näher untersucht. Dabei wird sowohl die Implementierung einzelner Nodes als auch aufgetretene Probleme, Herausforderungen und potenzielle Verbesserungsmöglichkeiten thematisiert.
Zu Beginn wird zunächst ein Überblick über die eingesetzte Software und deren Versionen gegeben.

4.1 Systemkonfiguration und Softwareversionen

Für diese Arbeit wurden sowohl auf dem Raspberry Pi des TurtleBots als auch in der virtuellen Maschine Ubuntu 22.04.5 LTS eingesetzt. Die Virtualisierung erfolgte mit VMWare Workstation Player 17 in der Version 17.5.2. Als ROS2-Distribution kam Humble zum Einsatz.
Die installierten Pakete auf dem TurtleBot sowie in der VM sind in der Tabelle 4.1 dargestellt.

TurtleBot	**VM**
ROS2 Installation	
ROS2 Humble	ROS2 Humble
ROS2 Pakete	
ros-humble-turtlebot3*	ros-humble-rviz2
colcon-common-extensions	colcon-common-extensions
	ros-humble-navigation2
	ros-humble-nav2-bringup
	ros-humble-slam-toolbox
open_manipulator_msgs	open_manipulator_msgs
	dynamixel-workbench
	open_manipulator
	open_manipulator_dependencies
	robotis_manipulator
Python Pakete	
python-dotenv 1.0.1	python-dotenv 1.0.1
	transitions 0.9.2

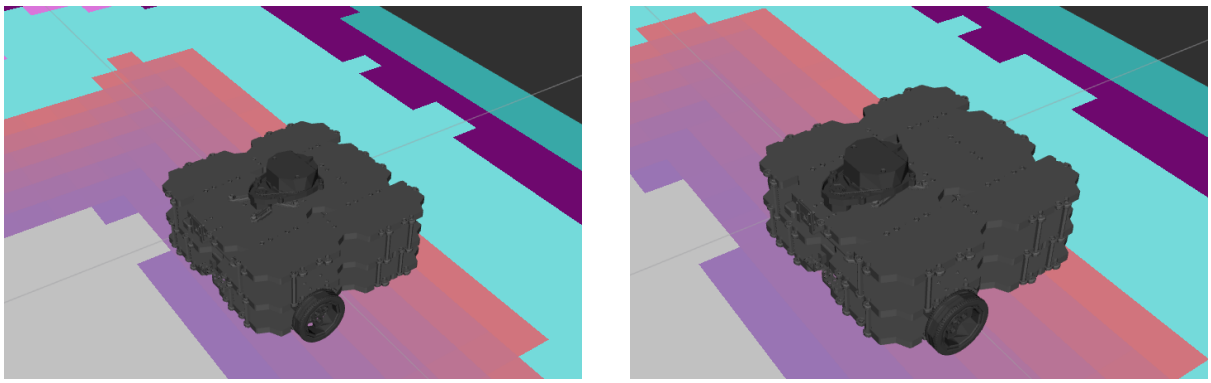
Tabelle 4.1: Übersicht der benötigten Pakete für TurtleBot und VM. Die Pakete für den OpenManipulator müssen selbst gebaut werden, siehe Abschnitt 7.3

4.2 Arbeit mit slam_toolbox und Nav2

Dieser Abschnitt bietet einen umfassenden Überblick über die Nutzung der ROS2-Packages `slam_toolbox` und `Nav2` für die Aufgabenbereiche Kartenerstellung, Lokalisierung und Navigation. Zu Beginn werden vorbereitende Hinweise erläutert, bevor in den nachfolgenden Abschnitten detailliert auf die einzelnen Aspekte eingegangen wird.

4.2.1 Vorbereitung und Hinweise

Wenn, wie in dieser Arbeit, der TurtleBot mit einem montierten OpenManipulator verwendet wird, muss der LDS in der URDF-Datei[1] des Roboters manuell nach vorne verschoben werden, da seine Position von der vorgesehenen Position auf dem TurtleBot abweicht. Dieser Schritt ist essenziell, um eine erfolgreiche Kartenerstellung und Navigation zu gewährleisten. Sollte der LDS in einer anderen Weise positioniert sein, muss dies ebenfalls berücksichtigt werden, da er den primären Sensor für Mapping und Navigation darstellt. Eine Anleitung dazu findet sich in 7.4 im Anhang. Ein Vergleich der falschen und korrekten Position des LDS in rviz ist in der Abbildung 4.1 dargestellt.



(a) falsche Position (b) korrekte Position

Abbildung 4.1: Vergleich der falschen LDS Position mit der korrekten Position.

4.2.2 Verwendung von slam_toolbox

Wie in Kapitel 2 erläutert, kann `slam_toolbox` in drei verschiedenen Modi betrieben werden. Für jeden dieser Modi stellt das `slam_toolbox`-Package die entsprechenden Launch- (.py) und Konfigurationsdateien (.yaml) bereit. Für die praktische Umsetzung sollten diese jedoch in ein eigenes Package kopiert werden, um sie projektbezogen anpassen zu können.
Die für diese Arbeit relevanten Dateien sind:

- `online_async_launch.py` + `mapper_params_online_async.yaml`: Ermöglichen Kartenerstellung durch *asynchrones* Mapping.

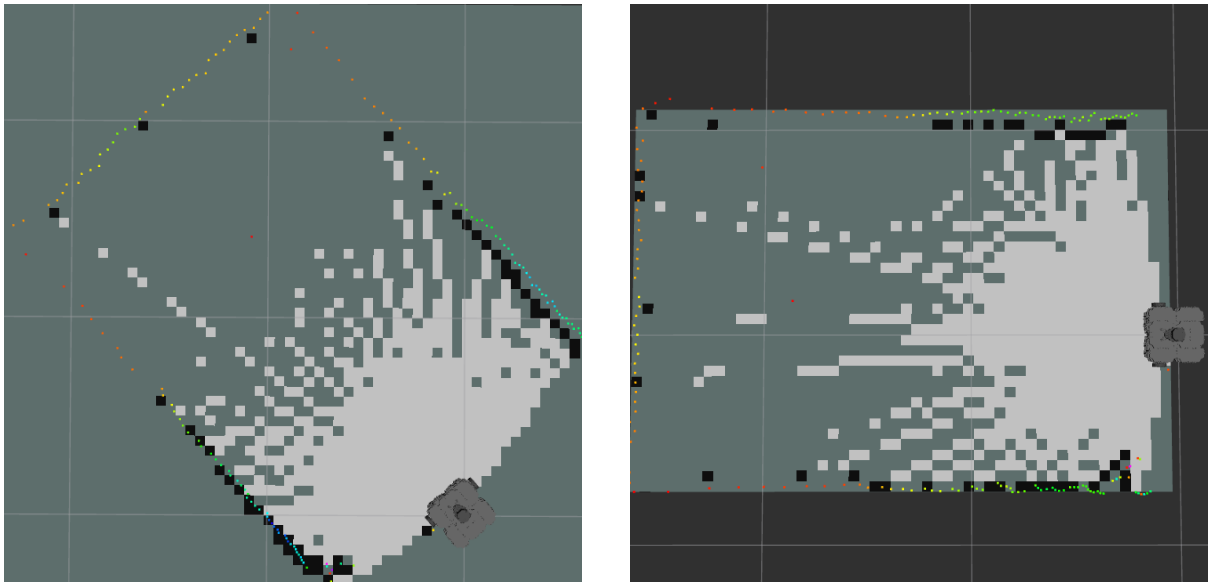
[1] Eine URDF-Datei (Unified Robot Description Format) ist eine XML-basierte Datei, die die physikalischen und kinematischen Eigenschaften eines Roboters beschreibt, einschließlich seiner Geometrie, Trägheit, Gelenke und Verbindungen, um die Simulation, Visualisierung und Steuerung in ROS zu ermöglichen. [3]

- `online_sync_launch.py` + `mapper_params_online_sync.yaml`: Ermöglichen Kartenerstellung durch *synchrones* Mapping.
- `localization_launch.py` + `mapper_params_localization.yaml`: Ermöglichen Lokalisierung während der Navigation.

Wichtige Parameter, welche in den Konfigurationsdateien angepasst werden müssen und ihre zugehörigen Werte werden in der nachfolgenden Auflistung beleuchtet. Alle Informationen stammen aus SteveMacenski [14].

- `base_frame`: `base_footprint` - Der Punkt, der den Mittelpunkt des Roboters beschreibt.
- `scan_topic`: `scan_filtered` - Die Quelle für die LDS-Daten. In diesem Fall werden die gefilterten Daten der Topic `scan_filtered` verwendet.
- `mode`: `mapping` während der Kartenerstellung / `localization` während der Navigation.
- `map_start_at_dock`: `true` (nur während der Navigation) - Wird verwendet, wenn der Roboter während der Navigation an der gleichen Position starten soll, an der auch die Kartenerstellung begonnen hat. Ist das für den spezifischen Anwendungsfall nicht der Fall, kann `map_start_at_dock` auf `false` gesetzt werden und stattdessen die Position des Roboters mit `map_start_pose` angegeben werden.
- `resolution`: `0.02` – Die Auflösung der zu erstellenden Karte, die angibt, wie viel Fläche in Metern ein Rasterpunkt (Pixel) der Karte in der realen Welt darstellt. In dieser Arbeit wurde eine Auflösung von 0,02 gewählt, was bedeutet, dass ein Punkt der Karte eine Fläche von 2 cm $\times$ 2 cm in der realen Welt repräsentiert. Kleinere Auflösungen ermöglichen zwar eine detailliertere Karte, erfordern jedoch mehr Rechenleistung während der Kartenerstellung und Navigation.

Die Kartenerstellung kann in `RViz` visualisiert werden, während der TurtleBot manuell mithilfe des Packages `turtlebot3_teleop` per Tastatur gesteuert wird.
Vor Beginn der Kartenerstellung sollte darauf geachtet werden, den Roboter in der gewünschten Anfangsposition einzuschalten. Wird der TurtleBot nach dem Einschalten bewegt, kann die integrierte IMU dazu führen, dass die Karte in `RViz` „schief“ dargestellt wird. Ob dies lediglich optische oder auch praktische Auswirkungen auf das Mapping und die Navigation hat, wurde im Rahmen dieser Arbeit nicht untersucht. Zur Sicherheit wird jedoch empfohlen, den Roboter korrekt zu positionieren. Eine visuelle Darstellung einer schiefen und einer geraden Karte ist in Abbildung 4.2 zu sehen.
Während der Kartenerstellung sollte der Roboter möglichst langsam bewegt werden, idealerweise mit nur 1–2 Tastenanschlägen pro Bewegungsrichtung, da eine zu schnelle Bewegung zu Verzerrungen in der Karte führen kann. Es wurde festgestellt, dass die Kartenerstellung besonders präzise Ergebnisse liefert, wenn der Roboter an festen Punkten langsam im Kreis gedreht wird, anstatt sich kontinuierlich durch die gesamte Karte zu bewegen.
Der Mapping-Prozess gilt als abgeschlossen, wenn alle für den Roboter zugänglichen Bereiche des erzeugten Rasters vollständig belegt (weiß) sind und alle Wände oder Hindernisse korrekt durch einen schwarzen Rahmen markiert wurden. Eine korrekte Karte für die Umgebung im KI-Labor (inklusive des Lagers) ist in Abbildung 4.3b zu sehen.

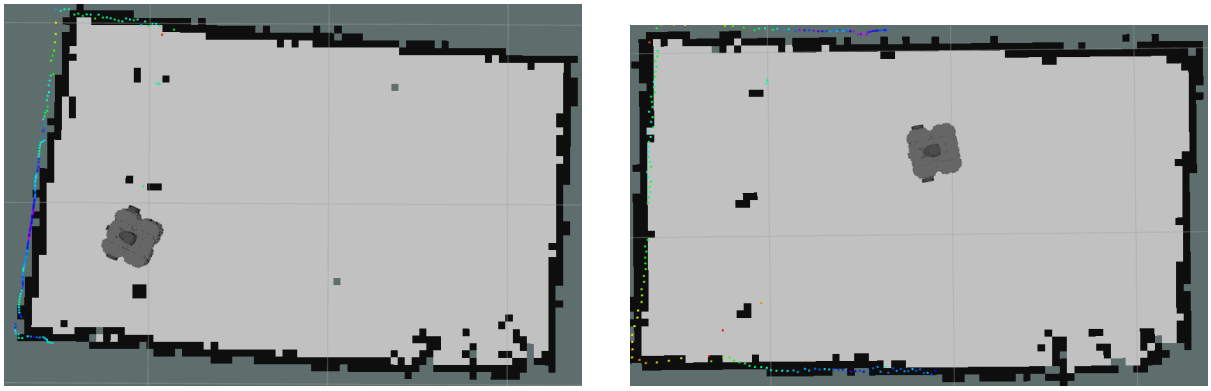


(a) schiefe Karte (b) gerade Karte

Abbildung 4.2: Vergleich einer schiefen Karte, welche durch nachträgliche Anpassung der Position des Roboters nach dem Start entsteht, und einer geraden Karte.

Die Hindernisse in der unteren rechten Ecke der Karte stellen LEGO-Roboter dar, die sich zum Zeitpunkt der Aufnahme noch auf der Fläche befanden. Für die finale Kartenerstellung wurden alle nicht vorgesehenen Hindernisse entfernt, um eine präzise und unverfälschte Karte zu gewährleisten.
Für die Kartenerstellung wurde in dieser Arbeit asynchrones Mapping verwendet. Zwar ermöglicht synchrones Mapping theoretisch eine detailliertere Karte, da es alle verfügbaren Informationen integriert, jedoch führte die dafür zusätzlich erforderliche Rechenleistung in der Praxis häufig zu schlechteren Ergebnissen, insbesondere bei hochauflösenden Karten.
Ein beispielhafter Vergleich von resultierenden Karten, welche durch die beiden Methoden erstellt wurden, findet sich in Abbildung 4.3. Es ist erkennbar, dass beide Karten einen vergleichbaren Detailgrad aufweisen. Allerdings zeigt die Karte, die mit synchronem Mapping erstellt wurde, eine deutliche Verzerrung. Dies ist vermutlich auf die begrenzte Rechenleistung des im TurtleBot verbauten Raspberry Pi zurückzuführen, was dazu führte, dass gebufferte Informationen falsch verarbeitet wurden. Die resultierende Karte kann direkt über das *SlamToolboxPlugin-Panel* in `RViz` gespeichert werden, welches in Abbildung 4.4 dargestellt ist. Dieses Panel lässt sich hinzufügen, indem man in `RViz` zu *Panels >Add New Panel >SlamToolboxPlugin* navigiert. Anschließend kann die Karte entweder als Rasterkarte über den Button *Save Map* oder als Pose-Graph über den Button *Serialize Map* gespeichert werden. Da `Nav2` mit der Rasterkarte und `slam_toolbox` mit der Pose-Graph-Karte arbeitet, sollte die Karte in beiden Formaten gespeichert werden. Dabei kann der gleiche Name für beide Kartenversionen verwendet werden, da sie als unterschiedliche Dateitypen gespeichert werden[2].

[2] `.pgm` und `.yaml` für die Rasterkarte / `.posegraph` und `.data` für den Pose-Graph



(a) synchrones Mapping (b) synchrones Mapping

Abbildung 4.3: Vergleich der resultierenden Karte mittels synchromen Mapping und asynchronem Mapping.

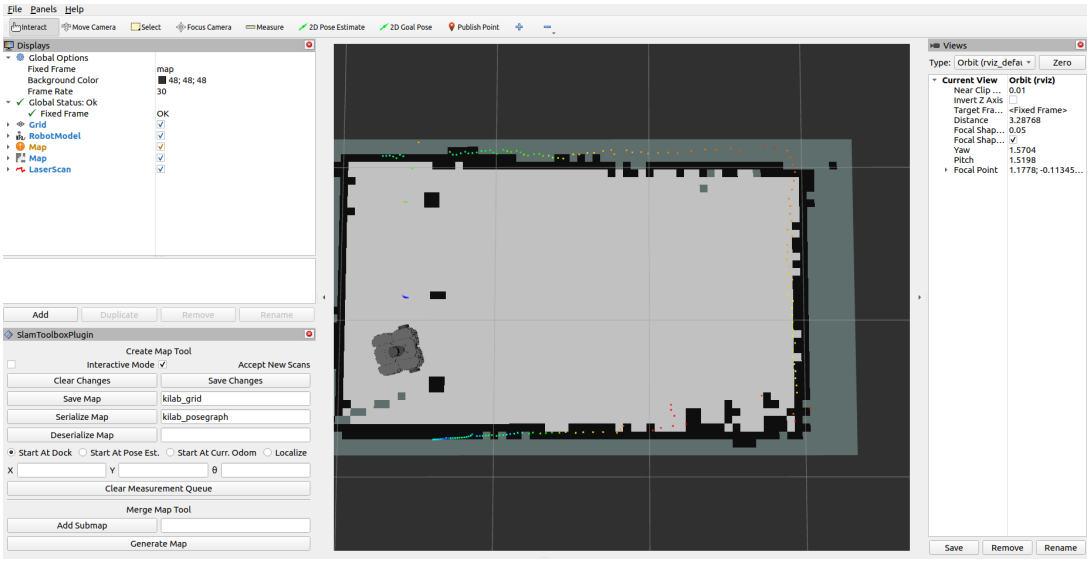


Abbildung 4.4: RViz UI mit hinzugefügtem *SlamToolboxPlugin-Panel*.

4.2.3 Verwendung von Nav2

Für die Verwendung von `Nav2` wurde in dieser Arbeit die Launch-Datei `navigation_launch.py` aus dem Paket `nav2_bringup` verwendet. Auch in diesem Kontext können spezifische Parameter über die Datei `nav2_params.yaml`, die sich im selben Package befindet, angepasst werden. Um die Parameteränderungen zu vereinfachen, sollten sowohl die Launch-Datei als auch die Parameterdatei in das eigene Package kopiert werden. Dabei ist darauf zu achten, dass der Pfad zur Parameterdatei in der Launch-Datei an das eigene Package angepasst wird.
`Nav2` kann verwendet werden, indem es parallel zur Lokalisierung von `slam_toolbox` gestartet wird. Die Navigationsfunktion kann getestet werden, indem in `RViz` manuell eine Zielposition über den *2D Goal Pose*-Button[3] auf der Karte markiert wird. Ist alles korrekt konfiguriert, sollte der Roboter zu dieser Position navigieren.
Indem `RViz` mit der von `nav2_bringup` bereitgestellten Parameterdatei `nav2_default_view.rviz` gestartet wird, können zusätzliche Informationen zur Navigation, wie der global geplante Pfad, angezeigt werden [10]. In der Praxis führte die Nutzung dieser Datei jedoch dazu, dass die VM abstürzte.
Um die Navigation an die spezifischen Bedingungen anzupassen, können sowohl Parameter als auch Plugins über die Datei `nav2_params.yaml` konfiguriert werden. In dieser Arbeit wurden keine Plugins angepasst, sodass im Folgenden ausschließlich auf wichtige Parameter und deren verwendete Werte eingegangen wird. Die Parameter werden für jeden verfügbaren Action Server und seine zugehörigen Plugins separat definiert, weshalb die Parameter in der Schreibweise `nav2_component/plugin/parameter` oder `nav2_component/parameter` angegeben werden. Alle Informationen zu Parametern stammen aus Steve Macenski [10].

- `controller_server/progress_checker/required_movement_radius`: `0.1` - Dieser Parameter definiert die minimale Distanz in Metern, die der Roboter zurücklegen muss, damit eine Bewegung als Fortschritt in Richtung des Ziels gewertet wird.

- `controller_server/precise_goal_checker/xy_goal_tolerance`: `0.25` - Dieser Parameter definiert die maximal zulässige Abweichung der Roboterposition zum Ziel in Metern, bei der die Navigation als erfolgreich abgeschlossen gilt. In der Praxis hat sich jedoch gezeigt, dass Werte unter 0,25 dazu führen, dass der Roboter seine Position kontinuierlich korrigiert und dadurch das Ziel nicht erreicht.

- `controller_server/precise_goal_checker/yaw_goal_tolerance`: `0.1` - Dieser Parameter legt die maximal zulässige Abweichung der Ausrichtung des Roboters zur Zielausrichtung in Radiant fest, bei der die Navigation als erfolgreich abgeschlossen gilt. Ähnlich wie bei `xy_goal_tolerance` führten Werte kleiner als 0,1 dazu, dass der Roboter seine Ausrichtung kontinuierlich korrigierte und das Ziel nicht erreichte.

- `local_costmap/resolution`: `0.02` - Auflösung der lokalen Costmap (Meter pro Rasterpunkt). Obwohl es nicht explizit empfohlen wird, wurde die Auflösung der Costmap der Auflösung der mit `slam_toolbox` erstellten Karte angepasst.

[3] Dieser befindet sich in der oberen Menüleiste und wird mit einem grünen Pfeil gekennzeichnet. Er ist in Abbildung 4.4 zu sehen.

- `global_costmap/resolution`: `0.02` - Auflösung der globalen Costmap (Meter pro Rasterpunkt). Auch hier wurde die Auflösung der zugrundeliegenden Karte von `slam_toolbox` angepasst.

- `local_costmap/inflation_layer/inflation_radius`: `0.3` - Dieser Parameter definiert den Radius in Metern, um den Hindernisse in der lokalen Costmap „aufgeblasen“ werden. Ein kleinerer Wert ermöglicht eine zuverlässigere Navigation in der Nähe von Hindernissen, erhöht jedoch gleichzeitig die Wahrscheinlichkeit, dass der Roboter diese berührt.

- `global_costmap/inflation_layer/inflation_radius`: `0.3` - Dieser Parameter definiert den Radius in Metern, um den Hindernisse in der globalen Costmap „aufgeblasen“ werden.

Für die Implementierung dieses Projekts wurde die `/navigate_to_pose`-Action von `Nav2` genutzt, um den Roboter zu definierten Zielpositionen zu navigieren. Diese Positionen entsprechen zuvor festgelegten Punkten vor den jeweiligen Produkten, die zur Laufzeit über Umgebungsvariablen eingelesen werden.
In der praktischen Umsetzung zeigte sich jedoch, dass die Navigation zwischen diesen Punkten teilweise problematisch war. Aufgrund des geringen Abstands von lediglich 0,5 m und der Nähe zu Hindernissen (Säulen) hatte der Roboter Schwierigkeiten, über diese kurzen Distanzen zuverlässig zu navigieren.
Um die Stabilität der Navigation zu verbessern, wurde daher eine zusätzliche Zwischenposition im zentralen Bereich der Karte eingeführt. Diese sollte es dem Roboter ermöglichen, zunächst einen weiter entfernten Punkt anzufahren, bevor er sich den Produkten nähert. Allerdings erwies sich diese Anpassung als ineffektiv, da sie hauptsächlich die Ausführungszeit verlängerte, ohne die Genauigkeit signifikant zu verbessern. Daher wird das Verfahren ohne Zwischenpositionen empfohlen, wenngleich beide Navigationsarten zu Demonstrationszwecken über den `begin_operation`-Service wählbar sind. Genaue Details zur Auswahl der entsprechenden Methode finden sich in Abschnitt 5.2.

4.3 Implementierung einzelner Nodes

In diesem Abschnitt soll auf die Besonderheiten in der Implementierung einzelner Nodes eingegangen werden, welche maßgeblich an der Umsetzung der angestrebten Funktionalitäten beteiligt sind. Ausgewählt wurden hier für die `warehouse_bot_main`-, `product_manipulator`-, `product_aligner`-, `scan_filter`- und `product_info_provider`-Nodes da ihre Implementierungen nicht trivial sind.

4.3.1 warehouse_bot_main

Diese Node stellt die zentrale Steuereinheit des Roboters dar und wurde unter Verwendung der Python-Bibliothek `transitions` als State-Machine implementiert. Die Entscheidung für eine State-Machine basiert darauf, dass der Roboter in unterschiedlichen Szenarien, wie etwa während der Navigation, der Ausrichtung oder dem Greifen nach einem Produkt, unterschiedliche Funktionalitäten ausführen und Informationen verarbeiten muss. Durch die Implementierung als State-Machine wird sichergestellt, dass der Roboter nur solche Informationen verarbeitet, die für den aktuellen Zustand relevant

sind, und ausschließlich die Funktionalitäten ausführt, die in diesem Zustand erlaubt sind. Dadurch wird das Risiko unvorhergesehenen Verhaltens reduziert, da der Übergang zwischen Zuständen nur unter bestimmten, festgelegten Bedingungen erfolgen kann. Die verfügbaren Zustände sind folgende: `IDLE`, `NAVIGATING`, `ALIGNING_WITH_PRODUCT`, `GRABBING_PRODUCT`, `MOVING_BACK`, `PUTTING_DOWN_PRODUCT` und `ERROR`.
Dennoch sind mit dieser Implementierung auch gewisse Nachteile verbunden. So entsteht beispielsweise ein zusätzlicher Overhead durch die Definition der Zustände und Zustandsübergänge. Darüber hinaus generiert die `transitions`-Bibliothek implizit Funktionen für den Übergang zwischen Zuständen. Diese Funktionen werden an den entsprechenden Stellen aufgerufen, sind jedoch nicht explizit definiert, was potenziell zu Verwirrung führen kann. Trotz dieser Nachteile überwiegen die Vorteile der Implementierung in diesem Fall.
Die resultierende State-Machine inklusive Zuständen und Übergängen kann der Abbildung 4.5 entnommen werden. Beim Übergang in den entsprechenden Zustand schickt die `warehouse_bot_main`-Node Anfragen an insgesamt vier verschiedene Action Server. Die zugehörigen Actions sind folgende:

- `/navigate_to_pose`: Diese Action wird vom Package `nav2_bringup` definiert und navigiert den Roboter an eine bestimmte Position auf der Karte. Das Goal der Anfrage enthält die gewünschte Position des Roboters vom Typ `geometry_msgs/Point` und die Orientierung des Roboters vom Typ `geometry_msgs/Quaternion`. Ist die Action erfolgreich, befindet sich der Roboter an der korrekten Position um in den Zustand `ALIGNING_WITH_PRODUCT` zu wechseln.

- `/align_with_product`: Eine selbstdefinierte Action, welche von der Node `product_aligner` bereitgestellt wird. Ihr Ziel ist es, den Roboter so mittig vor dem Produkt auszurichten, dass der OpenManipulator auf dem TurtleBot sich nur an eine vorher festgelegte Position bewegen muss, um das Produkt erfolgreich zu greifen. Die Definition der Action kann im Anhang in Abbildung 7.1 nachgelesen werden. Die genaue Funktionsweise wird in Abschnitt 4.3.2 geklärt.

- `/manipulate_product`: Ebenfalls eine eigens definierte Action, welche von der Node `product_manipulator` bereitgestellt wird. Sie ist für die Steuerung des OpenManipulators verantwortlich und greift bzw. legt das Produkt ab, abhängig vom aktuellen Zustand des Roboters. Die Definition der Action kann im Anhang in Abbildung 7.2 nachgelesen werden. Die genaue Funktionsweise wird in Abschnitt 4.3.3 geklärt.

- `/move_back`: Eine selbstdefinierte Action, welche von der Node `move_back_server` bereitgestellt wird. Sie ist dafür Zuständig, den Roboter nach dem Greifen des Produkts in einer angegebenen Geschwindigkeit für eine bestimmte Zeit nach hinten fahren zu lassen, damit er danach problemlos navigieren kann. Ohne vorher Abstand vom Produkt zu nehmen befindet sich der Bot nämlich innerhalb des „aufgeblasenen“ Bereichs um das Produkt, welches von `Nav2` als Hindernis erkannt wird, was Schwierigkeiten bei der Navigation bereitet. Die Definition der Action kann im Anhang in Abbildung 7.3 nachgelesen werden.

Bei erfolgreicher Ausführung der Action wechselt die Node in den nächsten vorgesehenen Zustand. Sollte die Action unerwartet fehlschlagen, versetzt sich der Roboter in einen Fehlerzustand. Eine detaillierte Fehlerbehandlung wurde nicht implementiert, da es sich

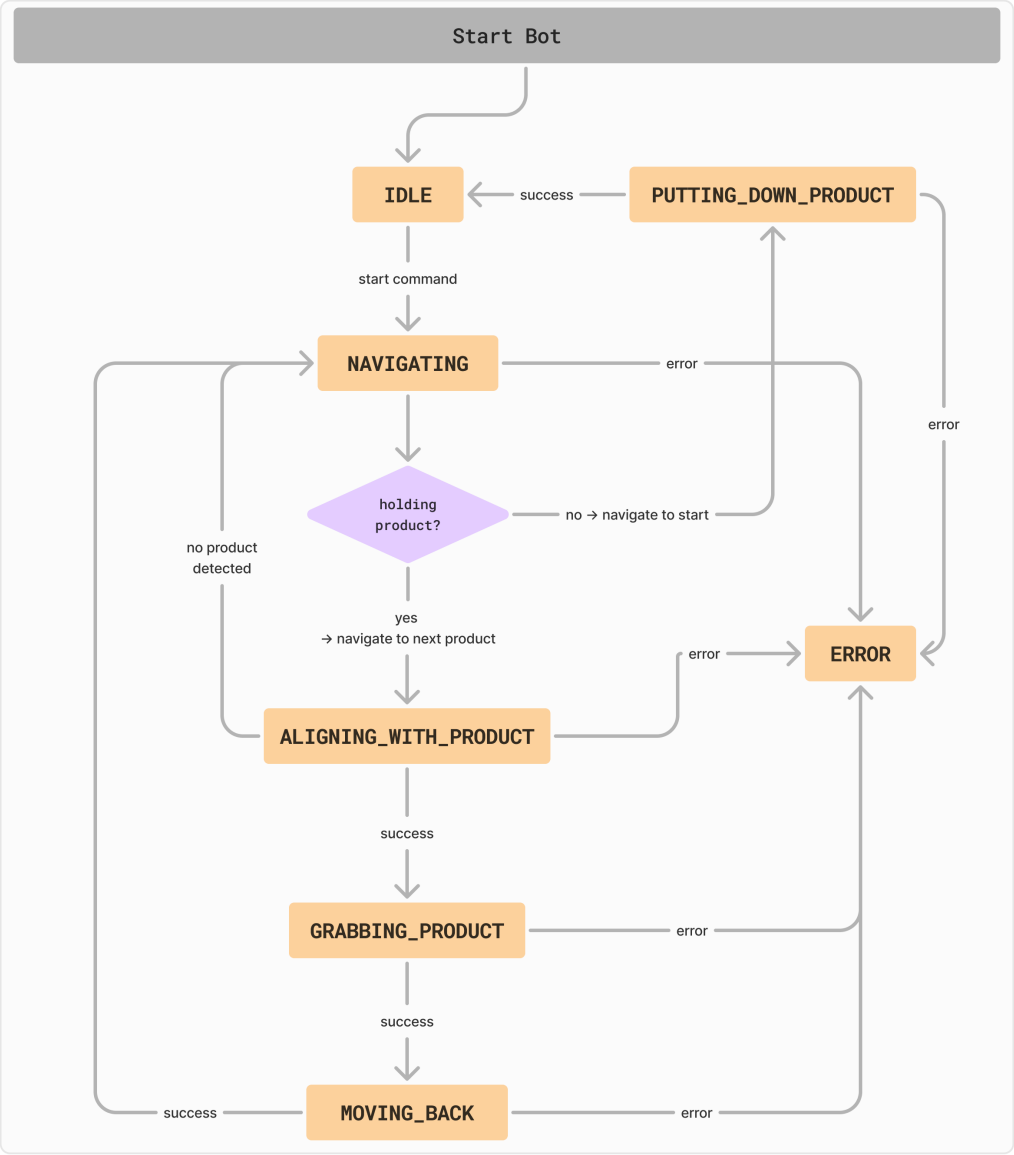


Abbildung 4.5: State-Machine der *warehouse_bot_main*-Node. Zustände werden als gelbe Rechtecke definiert, Pfeile zeigen Zustandsübergänge an.

um einen Prototyp für Demonstrationszwecke handelt, der im Fehlerfall einfach neu gestartet werden kann. Zudem zeigte sich in den Robustheitsexperimenten, dass kein einziger unerwarteter Fehler auftrat, der den Error-Zustand ausgelöst hätte.
Eine mögliche Zustandsabfolge während des Betriebs, bei dem sich das gesuchte Produkt auf der zweiten vom Roboter angefahrenen Säule befindet, könnte folgendermaßen aussehen:

- `IDLE`: Der Roboter ist bereit zur Ausführung und bekommt die Startanweisung über den Service `begin_operation`.
- `NAVIGATING`: Der Roboter navigiert zur ersten Säule.
- `ALIGNING_WITH_PRODUCT`: Der Roboter erkennt kein Produkt. Die Action `align_with_product` gibt `success: False` zurück.
- `NAVIGATING`: Der Roboter navigiert zur zweiten Säule.
- `ALIGNING_WITH_PRODUCT`: Der Roboter erkennt das Produkt und positioniert sich korrekt davor. Die Action `align_with_product` gibt `success: True` zurück
- `GRABBING_PRODUCT`: Der Roboter greift das Produkt erfolgreich mit dem OpenManipulator. Die Action `manipulate_product` gibt `success: True` zurück.
- `MOVING_BACK`: Der Roboter fährt rückwärts um einen ausreichenden Abstand von der Säule sicherzustellen. Die Action `move_back` gibt `success: True` zurück.
- `NAVIGATING`: Der Roboter fährt zurück zur Ausgangsposition.
- `PUTTING_DOWN_PRODUCT`: Der Roboter legt das Produkt mit dem OpenManipulator vor sich ab. Die Action `manipulate_product` gibt `success: True` zurück.
- `IDLE`: Der Roboter ist bereit zur Ausführung.

Die Implementierung der Node als State-Machine sowie das Auslagern der Funktionalitäten in verschiedene Action-Server gewährleisten, dass die Node als zentrales Steuerungselement des Roboters fungieren kann und dabei eine möglichst hohe Modularität beibehält. Dadurch wird eine spätere Erweiterung der Node um zusätzliche Funktionalitäten problemlos ermöglicht.

4.3.2 product_aligner

Die `product_aligner`-Node stellt die Action `/align_with_product` bereit. Bekommt sie eine Anfrage von einem Action-Client, werden als Goal folgende 4 Parameter übergeben:

- `float64 product_diameter`: Gibt den Durchmesser des Produkts im Bild an, gemessen in Pixeln (px).
- `float64 product_diameter_tolerance`: Gibt die maximale absolute Abweichung vom angestrebten Durchmesser in px an, damit der Parameter als optimal bewertet wird.
- `float64 product_center_offset`: Gibt den Abstand zwischen der Mitte des Produkts und der Bildmitte entlang der x-Achse in px an.

- `float64 product_center_offset_tolerance`: Gibt die maximale zulässige absolute Abweichung der Produktmitte von der Bildmitte in px an, damit der Parameter als optimal bewertet wird.

Während der Bearbeitung einer Anfrage abonniert die Node kontinuierlich die Topic `/product_info`, auf der der aktuelle Durchmesser des Produkts sowie dessen Entfernung von der Bildmitte veröffentlicht werden. Diese Informationen werden als Instanzvariablen in der Node gespeichert, um sie jederzeit abrufen zu können.
Um die gleichzeitige Ausführung des Action-Callbacks und des Subscription-Callbacks zu ermöglichen, wird die Node in einem `MultithreadedExecutor` betrieben. Dieser erlaubt mithilfe von `Multithreading` die parallele Ausführung mehrerer Callbacks innerhalb einer `CallbackGroup`. Ohne die Verwendung des `MultithreadedExecutor` wäre es nicht möglich, gleichzeitig Informationen von der abonnierten Topic zu beziehen, da das Action-Callback die Ausführung des Subscription-Callbacks blockieren würde. [3]
Während des Action-Callbacks optimiert der Roboter kontinuierlich die Parameter `product_diameter` und `product_center_offset`, bis ihre Abweichung vom Sollwert innerhalb des vorgegebenen Toleranzbereichs liegt. Die Optimierung erfolgt hierbei in einer `while`-Schleife, die beide Parameter parallel anpasst.
Um den Produktdurchmesser zu optimieren, nähert sich der Roboter dem Produkt an oder entfernt sich davon. Analog dazu wird der Abstand der Produktmitte zur Bildmitte durch eine Drehung des Roboters optimiert, entweder hin zum Produkt oder weg davon. Die Bewegungen – Vorwärts- und Rückwärtsbewegung sowie Drehung – werden realisiert, indem Nachrichten des Typs `geometry_msgs/Twist` auf der Topic `cmd_vel` veröffentlicht werden. Diese Nachrichten werden von der Aktorik des TurtleBot abonniert und entsprechend umgesetzt. Die `Twist`-Nachricht enthält zwei Parameter: einen dreidimensionalen Vektor `linear` für lineare Bewegungen und einen dreidimensionalen Vektor `angular` für Drehbewegungen. Für die Vorwärts- oder Rückwärtsbewegung wird der `x`-Wert des `linear`-Vektors genutzt, während die Drehung durch den `z`-Wert des `angular`-Vektors gesteuert wird.
Um eine schnelle und präzise Ausrichtung des Roboters auf das Produkt sicherzustellen, wird die Geschwindigkeit seiner Bewegung während des Optimierungsprozesses dynamisch mit der Abweichung vom Sollwert skaliert. Das bedeutet, dass der Roboter bei größeren Abweichungen schneller agiert, während er bei Annäherung an den Sollwert langsamer wird. Diese Strategie dient dazu, Probleme durch Latenzen in der asynchronen Kommunikation zu minimieren.
Wenn der Roboter sich mit hoher Geschwindigkeit dem Sollwert nähert, kann es aufgrund der Verzögerung zwischen der Veröffentlichung einer `cmd_vel`-Nachricht und deren Umsetzung durch die Aktorik des TurtleBot dazu kommen, dass der Roboter über das Ziel hinaus fährt. Dadurch könnte der Parameter erneut außerhalb des gewünschten Wertebereichs liegen. Andererseits können zu geringe Geschwindigkeitswerte dazu führen, dass diese nicht mehr korrekt von der Motorsteuerung des TurtleBot verarbeitet werden.
Um solchen Problemen entgegenzuwirken, werden die Bewegungswerte nicht nur skaliert, sondern zusätzlich mit minimalen und maximalen Grenzen versehen. Dies stellt sicher, dass die Bewegung einerseits effizient bleibt, andererseits aber auch zuverlässig im vorgesehenen Bereich verarbeitet werden kann.
Sollte das Produkt während des Optimierungsprozesses für einen bestimmten Zeitraum nicht im Bild zu sehen sein, wird der Vorgang nach einigen Sekunden abgebrochen die Action gibt ein Result mit `success=False` zurück.

4.3.3 product_manipulator

Diese Node stellt die Action `/manipulate_product` bereit und führt Befehle aus, um mithilfe des OpenManipulators das Produkt zu greifen oder abzulegen. Bei einer Anfrage eines Action-Clients wird folgender Parameter als Goal übergeben:

- `string task`: Nimmt die zwei Werte *grip* oder *release* an, welche bestimmen, welche Aufgabe der OpenManipulator ausführen soll.

Ähnlich wie die `warehouse_bot_main`-Node ist auch diese Node als State-Machine implementiert, um die Bewegungsabläufe des OpenManipulators zu steuern. Die Verwendung einer State-Machine gewährleistet, dass die Abläufe klar strukturiert und an die jeweilige Situation angepasst sind. Ein Fehler in einem Zustand, beispielsweise eine falsche Bewegung, könnte nicht nur den Erfolg der aktuellen Handlung gefährden, sondern auch technische Komponenten beschädigen.
Die komplette State-Machine ist in Abbildung 4.6 dargestellt. Neben der State-Machine speichert die Node ebenfalls Informationen darüber, ob der OpenManipulator gerade ein Produkt hält oder nicht.

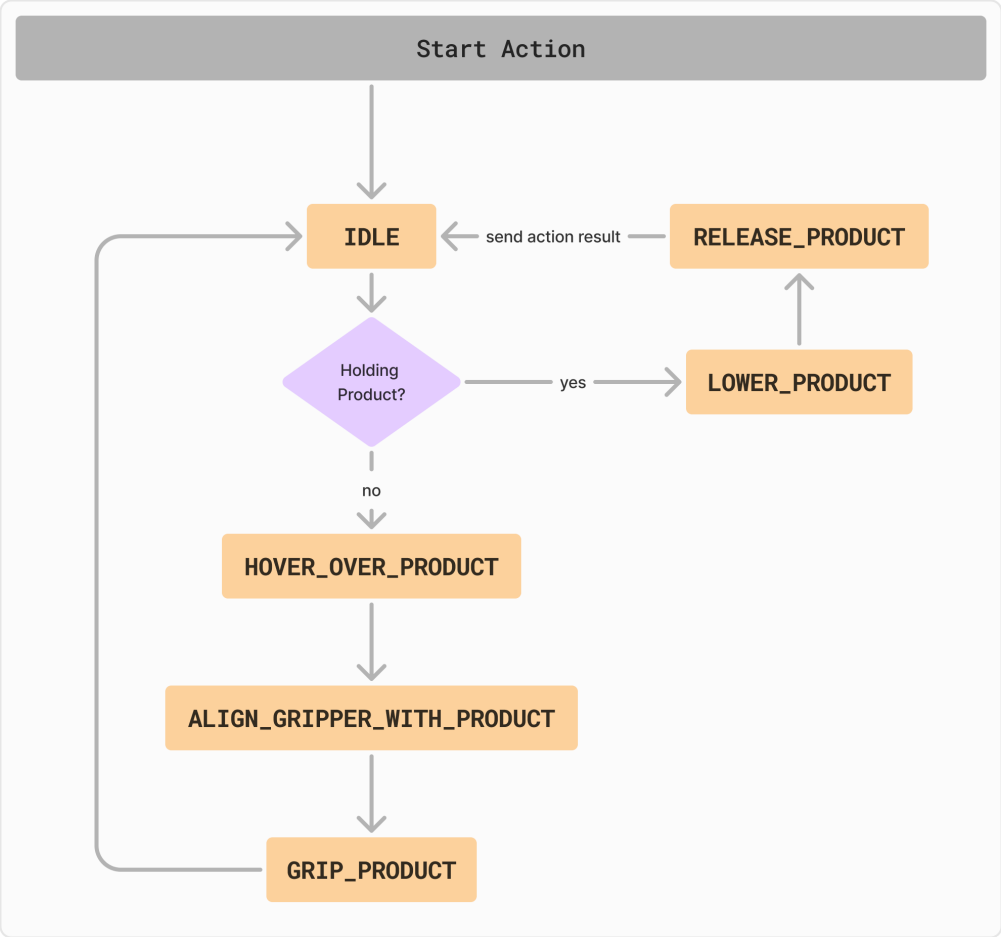


Abbildung 4.6: State-Machine der *product_manipulator*-Node. Zustände werden als gelbe Rechtecke definiert, Pfeile zeigen Zustandsübergänge an.

Die Logik der Node ist relativ einfach strukturiert. Sobald eine Anfrage eines Action-Clients eingeht, beginnt die Verarbeitung im Action-Callback mit der Überprüfung, ob die Voraussetzungen für die ausgeführte Handlung, die im `task`-Attribut des Goals definiert sind, erfüllt sind. Für die Aktion *grip* besteht die Voraussetzung darin, dass der OpenManipulator derzeit kein Produkt hält, während für *release* die Voraussetzung ist, dass er ein Produkt hält.
Nach der Überprüfung der Voraussetzungen wird eine Abfolge von Zuständen ausgewählt, die vom OpenManipulator durchlaufen wird. Bei jedem Zustandsübergang führt der Manipulator eine vordefinierte Bewegung aus. Diese Bewegungen werden über zwei verschiedene Services gesteuert: Für Bewegungen des gesamten Arms wird der Service `/open_manipulator/goal_joint_space_path` genutzt, während für das Öffnen oder Schließen des Grippers der Service `/open_manipulator/goal_tool_control` zum Einsatz kommt.
Der Grund, warum der OpenManipulator nur im Joint Space und nicht im Task Space bewegt wird, liegt darin, dass Bewegungen im Task Space nicht erfolgreich ausgeführt werden konnten. Der Grund hierfür ist, dass die notwendige Berechnung der inversen Kinematik in diesem Kontext nicht durchgeführt werden konnte, was jedoch nicht weiter untersucht wurde.
Bei der Arbeit mit den beiden Services im Joint Space wird stets eine Liste von Gelenken (Servomotoren) des Arms zusammen mit den gewünschten Winkelwerten sowie einem Attribut `path_time` übergeben. Letzteres gibt die Zeit an, die der OpenManipulator benötigt, um von der Ausgangsposition die gewünschte Zielposition zu erreichen. Hierbei ist zu beachten, dass der Service bereits eine positive Rückmeldung sendet, sobald festgestellt wurde, ob die gewünschte Position erreicht werden kann und nicht sobald der Arm die Zielposition erreicht hat. Aus diesem Grund wird nach der positiven Bestätigung manuell mit `time.sleep()` für die Dauer der Bewegung (`path_time`) gewartet. Diese Maßnahme dient dazu, mögliche Fehler zu vermeiden, die durch parallele Aufrufe der Services entstehen könnten.
Sobald die Reihenfolge der Zustandsübergänge abgeschlossen ist, sendet die Node ein Result an den aufrufenden Action-Client. In diesem Result wird mit dem Attribut `is_holding_product` festgehalten, ob der Arm derzeit ein Produkt hält oder nicht.

4.3.4 scan_filter

Die Node `scan_filter` führt keine komplexe Logik aus, sondern dient dazu, die vom LDS gelieferten Informationen in einem bestimmten Bereich zu filtern.
Sie abonniert die aktuellen Daten des LDS, die als Nachrichten vom Typ `sensor_msgs/msg/LaserScan` auf dem Topic `/scan` veröffentlicht werden. Diese Nachricht enthält unter anderem ein Array `float32[] ranges`, das 360 Entfernungswerte beinhaltet, die vom LDS aufgezeichnet werden. Dabei beschreibt der erste Eintrag die Entfernung zu einem Punkt direkt vor dem Roboter, während alle weiteren Einträge (insgesamt 360) im Uhrzeigersinn die Entfernungen zu den entsprechenden Punkten im 360°-Bereich rund um den Roboter angeben.
Da der LDS auf dem TurtleBot vor dem OpenManipulator platziert ist, enthält das `ranges`-Array auch Werte, die die Entfernung vom LDS zum OpenManipulator darstellen. Diese Entfernungswerte werden von `slam_toolbox` fälschlicherweise als Hindernisse interpretiert und müssen daher bei der Verarbeitung ignoriert werden. Im Rahmen dieses Projekts werden daher alle Werte im `ranges`-Array, die Entfernungen unter 0,1 Metern repräsentieren, ignoriert, da es sich bei diesen mit hoher Wahrscheinlichkeit nicht um

echte Hindernisse, sondern um den OpenManipulator selbst handelt.
Durch die Entfernung von Informationen des LDS wird die Lokalisierungs- und Navigationsgenauigkeit des Roboters negativ beeinträchtigt. Allerdings wäre ohne diese Informationen die Lokalisierung und Navigation gar nicht erst möglich gewesen.

4.3.5 product_info_provider

Diese Node arbeitet mit `OpenCV` und ist so konfiguriert, dass sie blaue Bereiche im Bild erkennen kann. Dies wird durch die Erstellung einer Maske im Bild erreicht, die ausschließlich Pixel eines ausgewählten Farbbereichs enthält. Zur Definition dieser Farbbereiche wird der HSV-Farbraum verwendet, obwohl Bilder in `OpenCV` typischerweise im BGR-Farbraum dargestellt werden. Der Einsatz des HSV-Farbraums ist sinnvoll, da er eine intuitivere und präzisere Definition von Farbbereichen ermöglicht.
HSV steht für *Hue* (Farbton), *Saturation* (Sättigung) und *Value* (Helligkeit). Hue beschreibt die eigentliche Farbe eines Pixels und wird als Winkel auf einem Farbkreis angegeben. Dieser Winkel reicht von 0° bis 360°, wird jedoch in `OpenCV` auf einen Bereich von 0 bis 180 skaliert [15]. Die Auswahl eines spezifischen Winkelbereichs ermöglicht eine klare Definition einer gewünschten Farbe, was die Farbauswahl im Vergleich zum BGR-Farbraum erheblich vereinfacht. [16]
Nachdem die Maske des Bildes erstellt wurde, werden unerwünschte Artefakte durch *Erosion* entfernt. Anschließend werden die Konturen der in der Maske identifizierten Objekte ermittelt. In dieser Arbeit wird angenommen, dass die größte gefundene Kontur dem gesuchten Produkt entspricht, weshalb ausschließlich diese weiter analysiert wird. Diese wird danach dazu verwendet, zu bestimmen, ob sich ein Produkt im Sichtfeld der Kamera befindet, wie zentral es im Bild platziert ist und welche Größe es hat. Die entsprechenden Informationen werden auf dem Topic `product_info` veröffentlicht.

4.4 Optimierungspotenziale

Das größte Potenzial zur Verbesserung liegt derzeit in der Greifgenauigkeit des OpenManipulators. In den praktischen Tests zeigte sich, dass das Greifen des Produkts zwar zuverlässig funktioniert, jedoch können im praktischen Betrieb potenziell Ungenauigkeiten auftreten. Zur Verbesserung dieses Prozesses gibt es zwei mögliche Ansätze: Die Verbesserung der Positionierung des TurtleBots oder die Optimierung der Zusammenarbeit zwischen TurtleBot und OpenManipulator.
Im ersten Ansatz wäre es notwendig, die Möglichkeiten zur Erfassung von Umgebungsinformationen zu erweitern, um eine präzisere Positionierung des TurtleBots zu gewährleisten. Die aktuelle Kamera ist stark von guten Lichtverhältnissen abhängig, was jedoch nicht immer gegeben ist. Daher könnte eine Verbesserung der Kamera, möglicherweise in Kombination mit einer besseren Beleuchtung oder einer Kamera mit Tiefenmessung, eine Lösung darstellen. Eine Unabhängigkeit von Lichtbedingungen könnte zudem durch die zusätzliche Verarbeitung von Informationen aus dem LDS erreicht werden. Allerdings haben sich einzelne Distanzwerte in Tests als relativ unzuverlässig erwiesen, sodass hier weiterer Testaufwand erforderlich ist, um die optimale Lösung zu finden.
Eine elegantere Lösung als die manuelle Positionierung des TurtleBots wäre die direkte Zusammenarbeit zwischen TurtleBot und OpenManipulator. Dies könnte mithilfe des `turtlebot3_manipulation` Pakets realisiert werden, welches speziell für die Zusammenarbeit von TurtleBot und OpenManipulaotor entwickelt wurde [5]. Obwohl dieses in der

vorliegenden Arbeit nicht näher untersucht wurdee, ermöglicht es vermutlich eine präzise Erkennung der Position des Produkts relativ zum OpenManipulator durch den TurtleBot. So könnte der Arm automatisch zur erkannten Position des Produkts ausgerichtet werden. Auch in diesem Fall kann der Einsatz einer verbesserten Kamera oder zusätzlicher Sensoren zur Erkennung des Produkts eine sinnvolle Erweiterung darstellen.
Zusätzlich wäre es sinnvoll, die Erkennung weiterer Objekte, wie unterschiedlich farbiger Bälle, zu implementieren. Derzeit kann der Roboter ausschließlich blaue Bälle identifizieren. Wenn er beispielsweise rote Bälle direkt als solche erkennen könnte, würde dies eine schnellere Navigation zwischen den einzelnen Säulen ermöglichen.
Weiterhin könnte eine tiefere Analyse der `Nav2`-Plugins und -Parameter dazu beitragen, die Effizienz und Stabilität der Navigation zu verbessern. Zudem wäre die Integration des *Waypoint Following*-Features eine sinnvolle Erweiterung, um eine zuverlässigere Navigation zwischen den einzelnen Positionen zu ermöglichen.
Zusätzliche Verbesserungsmöglichkeiten umfassen den Einsatz von *ROS-Parametern* anstelle von Umgebungsvariablen, da diese besser in das ROS-Ökosystem integriert sind und zur Laufzeit modifiziert werden können, sowie eine differenziertere Fehlerbehandlung. Die Verwendung von *Behaviour Trees* anstelle von State Machines für die Nodes `warehouse_bot_main` und `product_manipulator` stellt ebenfalls eine sinnvolle Verbesserungsmöglichkeit dar. Aufgrund der begrenzten Zeit im Rahmen dieser Arbeit konnte diese Technologie nicht eingehend untersucht werden, jedoch könnte sie zu einem übersichtlicheren und leichter verständlichen Code beitragen.

5 Experimente und Robustheit

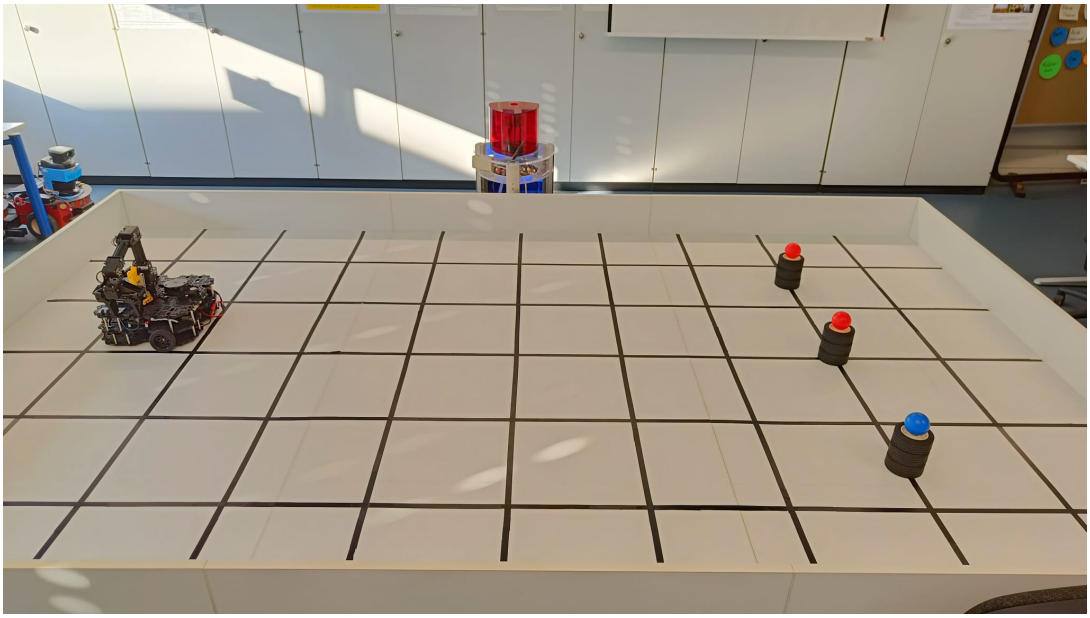
In diesem Kapitel wird die Einrichtung der Testumgebung für den Roboter beschrieben sowie der Ablauf zum erfolgreichen Start des Roboters erläutert. Im Anschluss werden die Ergebnisse der durchgeführten Tests präsentiert, um die Robustheit des Roboters bei der Ausführung seiner Aufgabe zu evaluieren.

5.1 Aufbau der Testumgebung

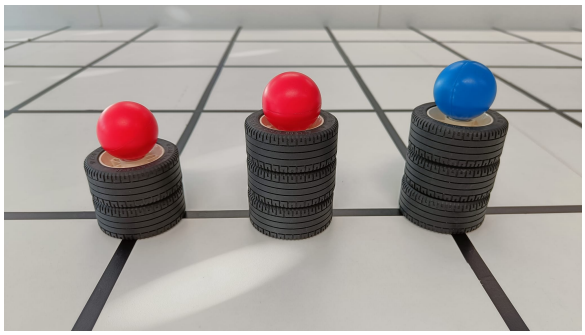
Die Testumgebung wird auf der Arbeitsfläche im KI-Labor eingerichtet und besteht aus drei Säulen, auf denen sich zwei rote und ein blauer Ball befinden. Der vollständige Aufbau der Testumgebung ist in Abbildung 5.1a visualisiert.
Jede Säule setzt sich aus drei übereinander gestapelten Reifen zusammen, wobei der oberste Reifen so ausgerichtet ist, dass er dem darauf platzierten Ball eine erhöhte Position bietet. Abbildung 5.1b zeigt die Unterschiede zwischen falsch aufgebauten Säulen (links, mittig) und korrekt aufgebauten Säulen (rechts). Es ist unbedingt zu beachten, dass sich alle Säulen an den vorgesehenen Positionen befinden, da der Roboter an diese vorher festgelegten Positionen navigieren wird.
Außerdem sollte sich der OpenManipulator in der vorgeschriebenen Position befinden, welche in Abbildung 5.1c gezeigt wird. Eine falsche Startposition kann dazu führen, dass er nicht korrekt verwendet werden kann.
Um eine konsistente Umgebungserfassung durch den LDS zu gewährleisten, sollte die Arbeitsfläche – mit Ausnahme des TurtleBots und der Säulen – vollständig freigehalten werden, sodass die erfassten Sensordaten mit der zuvor erstellten Karte übereinstimmen. Zudem ist darauf zu achten, dass die Säulen in Richtung der Tafel ausgerichtet werden. Die Raumbeleuchtung muss eingeschaltet sein, mit Ausnahme der Tafelbeleuchtung, da diese ein Gegenlicht erzeugt, das die Produkterkennung negativ beeinflussen kann. Ebenso können große blaue Objekte hinter den Säulen die Erkennung beeinträchtigen, weshalb sie entfernt werden sollten.

5.2 Start des Lagerroboters

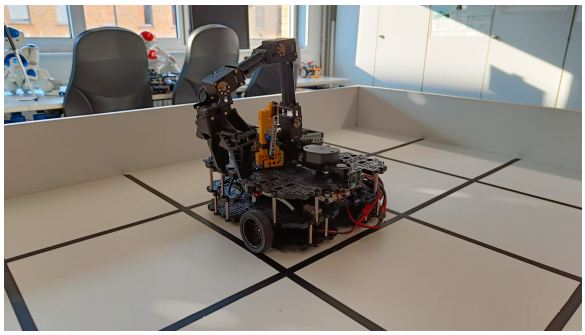
Vor dem Start ist sicherzustellen, dass die Testumgebung gemäß Abschnitt 5.1 korrekt aufgebaut wurde und alle erforderlichen Packages installiert sind. Diese sind in der Tabelle 4.1 aufgelistet. Eine übersichtlichere Installationsanleitung inklusive Befehle zum installieren der benötigten Packages findet sich außerdem in der README des GitHub Repositories dieses Projekts [17].



(a) Aufbau der Arbeitsfläche



(b) Korrekter Aufbau der Säulen (rechts)



(c) Startposition des OpenManipulators

Abbildung 5.1: Korrekter Aufbau der Testumgebung im KI-Labor

Anschließend kann die Launch-Datei `warehouse_bot_tb_launch` direkt auf dem TurtleBot mit folgendem Befehl gestartet werden:

```
ros2 launch warehouse_bot warehouse_bot_tb_launch.py
```

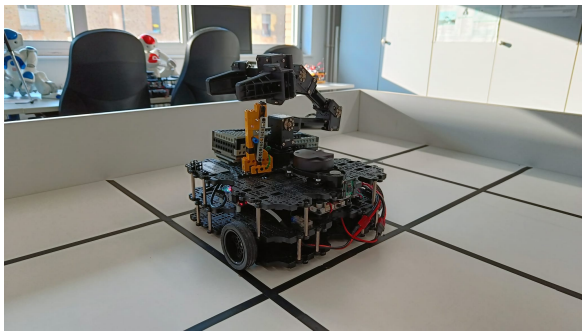
Diese Launch-Datei startet den TurtleBot inklusive seiner Sensorik sowie den OpenManipulator. Nach erfolgreicher Initialisierung sollte die letzte sichtbare Meldung `image_provider successfully initialized!` lauten.
Falls stattdessen die Fehlermeldung `failed to open camera by index` erscheint, konnte die Kamera nicht gestartet werden, sodass die Launch-Datei erneut ausgeführt werden muss.
In seltenen Fällen kann es vorkommen, dass der OpenManipulator und der LDS des TurtleBots an den falschen USB-Ports registriert werden. Dies äußert sich in der Fehlermeldung `open_manipulator_x_controller-4: process has died`, welche sich weiter oben in den Logs befindet. In diesem Fall sollte die Launch-Datei mit expliziter Angabe der USB-Ports neu gestartet werden:

```
ros2 launch warehouse_bot warehouse_bot_tb_launch.py usb_port_open_
manipulator:=/dev/ttyUSB0 usb_port_lds:=/dev/ttyUSB1
```

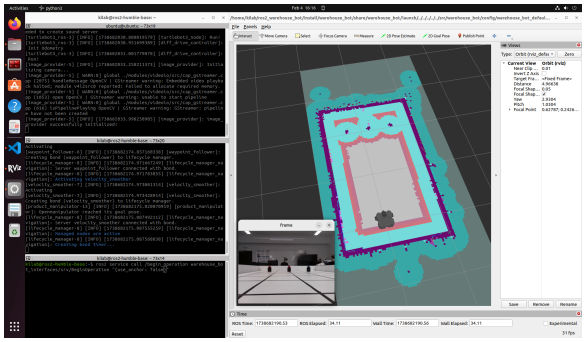
Danach kann die `warehouse_bot_main_launch` Launch-Datei in der VM auf einem der Laborrechner gestartet werden:

```
ros2 launch warehouse_bot warehouse_bot_main_launch.py
```

Dieser Befehl startet sowohl `Nav2` und `slam_toolbox` als auch nötigen Nodes der `warehouse_bot_pilots` und `warehouse_bot_world` Pakete sowie die `warehouse_bot_main` Node. Es kann einige Sekunden dauern, bis alle Befehle ausgeführt wurden. Danach befindet sich der Roboter im `IDLE` Zustand und wartet auf seinen Startbefehl. Wenn sich der OpenManipulator während des Startvorgangs nicht eigenständig in seine Ruheposition bewegt, liegt höchstwahrscheinlich ein Fehler vor und beide Launch-Dateien sollten erneut gestartet werden. Abbildung 5.2 zeigt die korrekte Ruheposition des OpenManipulators sowie den Zustand aller Fenster nach einem erfolgreichen Start.



(a) Ruhezustand OpenManipulator



(b) Zustand der Fenster

Abbildung 5.2: Zustand des OpenManipulator sowie aller Fenster auf der VM nach einem korrekten Start.

Anschließend kann die Ausführung der Aufgabe über den Service `begin_operation` mit folgendem Befehl initiiert werden:

```
ros2 service call /begin_operation warehouse_bot_interfaces/srv/
BeginOperation "{use_anchor: false}"
```

Er navigiert daraufhin nacheinander zu allen Säulen und sucht nach dem blauen Ball. Sobald er diesen gefunden hat, kehrt er zur Startposition zurück und legt den Ball dort ab. Danach ist der Roboter wieder betriebsbereit und kann über den Service `begin_operation` erneut gestartet werden. Der Parameter `use_anchor` kann alternativ auch auf `true` gesetzt werden. Dann werden die Zwischenpositionen in die Navigation integriert.
Sollten beim Start oder der Ausführung des Roboters unvorhergesehene Probleme auftreten, können potenzielle Lösungen im Abschnitt 7.2 im Anhang dieser Arbeit eingesehen werden.

5.3 Robustheit

Um die Robustheit des Roboters zu ermitteln wurde er in drei unterschiedlichen Szenarien getestet. Diese unterscheiden sich lediglich durch die Position des blauen Balls auf den Säulen:

1. *Szenario 1*: Ball befindet sich auf der linken Säule.
2. *Szenario 2*: Ball befindet sich auf der mittleren Säule.
3. *Szenario 3*: Ball befindet sich auf der rechten Säule.

Jedes Szenario beginnt mit dem Start des Lagerroboters, wie in Abschnitt 5.2 beschrieben. Der Roboter wird anschließend zehnmal hintereinander über den Service `begin_operation` gestartet. Danach wird ausgewertet, wie oft er seine Aufgabe korrekt ausgeführt hat und ob er sie vorher abbrechen musste.
Für jedes Szenario wird die Abfolge der Zustände analysiert, die der Roboter durchläuft. Dabei wird erfasst, wie häufig der Roboter in einem bestimmten Zustand seine Aufgabe abgebrochen hat. Die Anzahl der Abbrüche pro Zustand wird anschließend in eine Tabelle eingetragen. Erreicht der Roboter den letzten Zustand, wird auch dort die entsprechende Anzahl notiert. Zustände, die in einem bestimmten Szenario nicht durchlaufen wurden, werden mit einem „-“ markiert.[1].
Das Resultat dieses Experiments ist sowohl ein Maß für die Zuverlässigkeit des Roboters als auch eine Möglichkeit, potenzielle Schwachstellen zu identifizieren. Die Ergebnisse finden sich in Tabelle 5.1. Ein Video für einen exemplarischen Durchlauf von Szenario 3 findet sich im GitHub-Repository dieses Projekts[2] im Verzeichnis `static/videos`.

[1] Da der Roboter in Szenario 1 direkt nach der Ankunft an der ersten Säule ein Produkt greifen kann, fallen die Zustände NAVIGATING und ALIGNING_WITH_PRODUCT, welche in Szenario 2 genutzt werden, um zur nächsten Säule zu navigieren, weg. Das gleiche gilt für Szenario 2 verglichen mit Szenario 3.
[2] [17]

Zustand	**Szenario 1**	**Szenario 2**	**Szenario 3**
IDLE (Start)	0	0	0
NAVIGATING	0	0	0
ALIGNING_WITH_PRODUCT	0	0	0
NAVIGATING	-	0	0
ALIGNING_WITH_PRODUCT	-	0	0
NAVIGATING	-	-	0
ALIGNING_WITH_PRODUCT	-	-	0
GRABBING_PRODUCT	0	0	0
MOVING_BACK	0	0	0
NAVIGATING	1	0	0
PUTTING_DOWN_PRODUCT	0	0	0
IDLE (Erfolg)	9	10	10

Tabelle 5.1: Häufigkeit der erreichten Zustände bis zum Auftreten eines Fehlers oder bis die Aufgabe erfolgreich abgeschlossen wurde.

Die Resultate des Experiments deuten darauf hin, dass der Roboter seine Aufgabe zuverlässig und reproduzierbar ausführt. Von insgesamt 30 Durchläufen, die auf 3 Szenarien verteilt wurden, waren 29 erfolgreich. Nur ein einmaliger Fehler trat auf, bei dem der Navigationsprozess zur Startposition durch einen unbekannten Fehler unterbrochen wurde. Da nach dem Vorfall weder die Motoren noch das Terminal-Fenster auf Befehle reagierten, wird vermutet, dass es sich um einen internen Fehler des TurtleBots handelte.
Die Ausführungszeit des Roboters lässt jedoch noch Verbesserungspotential erkennen. Zwei Hauptfaktoren tragen hier zu Verzögerungen bei: Erstens versucht der Roboter nicht, in geraden Linien zu navigieren, sondern fährt oft in Bögen. Es wird vermutet, dass das verwendete `Nav2`-Plugin für den `planner_server` dazu neigt, sich an den "aufgeblasenen" Hindernissen der Costmap zu orientieren. Dieses Verhalten konnte durch das Anpassen verschiedener Parameter allerdings nicht behoben werden. Zweitens kommt es in seltenen Fällen während der Navigation zu kurzen Verzögerungen, bei denen der Roboter für einige Sekunden an der aktuellen Position verharrt. Dies könnte auf ein Problem im `controller_server` von `Nav2` hinweisen, tritt jedoch in weniger als 10% der Fälle auf.

6 Fazit und Ausblick

In dieser Arbeit wurde der prototypische Entwurf eines autonomen Lagerroboters analysiert und realisiert. Hierbei wurde eine moderne, modulare Systemarchitektur entwickelt, die auf ROS2 als Middleware basiert. Diese umfasst die Implementierung spezifischer Nodes, Topics, Services und Actions, welche die verschiedenen Funktionalitäten des Roboters ermöglichen. Darüber hinaus wurden die theoretischen Grundlagen der ROS2-Packages `slam_toolbox` und `Nav2` systematisch erarbeitet und angewandt, um eine Karte der Umgebung zu erstellen, die erfolgreich zur Navigation genutzt werden konnte.
Zudem wurden umfangreiche praktische Tests durchgeführt, deren Ergebnisse sorgfältig dokumentiert wurden. Für den Einsatz in der Lehre oder als Demonstrator wurden der entsprechende Code sowie detaillierte Anleitungen zur Inbetriebnahme bereitgestellt. Es wurde beleuchtet, welche typischen Fehler bei dem Umgang mit den verwendeten Technologien auftreten können und wie man diese lösen kann, um Studierenden und Lehrenden einen einfachen Einstieg in die Arbeit mit `slam_toolbox` und `Nav2` zu ermöglichen.
In einer möglichen zukünftigen Weiterentwicklung des Projekts sollte der Schwerpunkt insbesondere auf der Umsetzung der in Abschnitt 4.4 beschriebenen Verbesserungsmöglichkeiten liegen, um die Schnittstelle zwischen dem OpenManipulator und dem TurtleBot weiter zu verbessern.
Darüber hinaus könnte der Aspekt der Navigation und Kartenerstellung intensiver untersucht werden, indem die Betriebsumgebung des Roboters erweitert wird – beispielsweise auf einen Flurbereich des Informatikgebäudes. Durch die Integration von (dynamischen) Hindernissen könnten zudem Szenarien wie der Betrieb des Roboters in Menschenmengen analysiert werden.

Literatur

[1] Kento Kawaharazuka u. a. „Real-world robot applications of foundation models: a review“. In: *Advanced Robotics* 38.18 (Sep. 2024). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/01691864.2024.2408593, S. 1232–1254. ISSN: 0169-1864. DOI: 10.1080/01691864.2024.2408593. URL: https://doi.org/10.1080/01691864.2024.2408593 (besucht am 06. 02. 2025).

[2] E. Coste-Maniere und R. Simmons. „Architecture, the backbone of robotic systems“. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Bd. 1. ISSN: 1050-4729. Apr. 2000, 67–72 vol.1. DOI: 10.1109/ROBOT.2000.844041. URL: https://ieeexplore.ieee.org/abstract/document/844041 (besucht am 06. 02. 2025).

[3] *ROS 2 Documentation — ROS 2 Documentation: Humble documentation*. URL: https://docs.ros.org/en/humble/index.html (besucht am 12. 01. 2025).

[4] Steven Macenski u. a. „Robot Operating System 2: Design, architecture, and uses in the wild“. In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[5] *ROBOTIS e-Manual*. en. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/ (besucht am 12. 01. 2025).

[6] *ROBOTIS e-Manual*. en. URL: https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/ (besucht am 12. 01. 2025).

[7] Steve Macenski und Ivona Jambrecic. „(PDF) SLAM Toolbox: SLAM for the dynamic world“. en. In: *ResearchGate* (Dez. 2024). DOI: 10.21105/joss.02783. URL: https://www.researchgate.net/publication/351568967_SLAM_Toolbox_SLAM_for_the_dynamic_world (besucht am 12. 01. 2025).

[8] Sebastian Thrun und Michael Montemerlo. „The Graph SLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures“. en. In: *The International Journal of Robotics Research* 25.5-6 (Mai 2006). Publisher: SAGE Publications Ltd STM, S. 403–429. ISSN: 0278-3649. DOI: 10.1177/0278364906065387. URL: https://doi.org/10.1177/0278364906065387 (besucht am 13. 01. 2025).

[9] Steve Macenski u. a. „The Marathon 2: A Navigation System“. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. arXiv:2003.00368 [cs]. Okt. 2020, S. 2718–2725. DOI: 10.1109/IROS45743.2020.9341207. URL: http://arxiv.org/abs/2003.00368 (besucht am 14. 01. 2025).

[10] Joshua Wallace Steve Macenski Ruffin White. *Nav2 Documentation*. Zugriff am 06. Februar 2025. Open Navigation. 2023. URL: https://docs.nav2.org/about/index.html.

[11] Razan Ghzouli u. a. „Behavior Trees and State Machines in Robotics Applications“. In: *IEEE Transactions on Software Engineering* 49.9 (Sep. 2023). Conference Name: IEEE Transactions on Software Engineering, S. 4243–4267. ISSN: 1939-3520. DOI: 10.1109/TSE.2023.3269081. URL: https://ieeexplore.ieee.org/abstract/document/10106642 (besucht am 06. 02. 2025).

[12] Veit Kilian Siebert. „Entwicklung eines Demonstrators im Szenario eines mobilen Pflückroboters auf Basis eines TurtleBots und OpenMANIPULATORs“. Bachelorarbeit. Technische Hochschule Brandenburg, 2022.

[13] *About*. en-US. URL: https://opencv.org/about/ (besucht am 16. 01. 2025).

[14] SteveMacenski. *slam_toolbox*. https://github.com/SteveMacenski/slam_toolbox. Zugriffsdatum: 12.01.2025. 2018.

[15] OpenCV. *OpenCV Documentation*. Zugriff am 06. Februar 2025. OpenCV. 2025. URL: https://docs.opencv.org/4.x/index.html.

[16] Hyun-Chul Kang u. a. „HSV Color-Space-Based Automated Object Localization for Robot Grasping without Prior Knowledge“. en. In: *Applied Sciences* 11.16 (Jan. 2021). Number: 16 Publisher: Multidisciplinary Digital Publishing Institute, S. 7593. ISSN: 2076-3417. DOI: 10.3390/app11167593. URL: https://www.mdpi.com/2076-3417/11/16/7593 (besucht am 06. 02. 2025).

[17] Arne Allwardt. *ros2_warehouse_bot*. https://github.com/arneallwardt/ros2_warehouse_bot. Zugriffsdatum: 05.02.2025. 2025.

7 Anhang

7.1 Definition von selbsterstellten Actions

```
float64 product_diameter
float64 product_diameter_tolerance
float64 product_center_offset
float64 product_center_offset_tolerance
---
bool success
float64 product_diameter
float64 product_center_offset
---
float64 product_diameter
float64 product_center_offset
```

Abbildung 7.1: Definition der *AlignProduct*-Action.

```
string task
---
bool success
bool is_holding_product
---
bool is_holding_product
```

Abbildung 7.2: Definition der *ManipulateProduct*-Action.

```
float64 duration
float64 speed
---
bool success
---
float64 time_remaining
```

Abbildung 7.3: Definition der *MoveBack*-Action.

7.2 Probleme und Lösungen

In diesem Abschnitt werden Herausforderungen aufgeführt, die während der Arbeit mit dem TurtleBot, dem OpenManipulator und den in dieser Arbeit verwendeten Packages aufgetreten sind. Zudem werden potenzielle Lösungsvorschläge für die identifizierten Probleme vorgestellt.

Die Installation des OpenManipulators auf dem TurtleBot schlägt fehl.

Installationsanleitung im Anhang unter Abschnitt 7.3 folgen.

Der Controller des OpenManipulators (open_manipulator_x_controller) kann nicht gestartet werden.

Installationsanleitung im Anhang unter Abschnitt 7.3 folgen. Launch-Datei anpassen.

Die interne Kamera des TurtleBots kann nicht geöffnet werden.

In der Datei `/boot/firmware/config.txt` die Zeilen `camera_auto_detect=1` und `display_auto_detect=1` entfernen und stattdessen die Zeile `dtoverlay=imx708` einfügen. So wird manuell die passende Kamera ausgewählt, da sie nicht automatisch vom Raspberry Pi gefunden wird.

Bei der Kartenerstellung / Lokalisierung mit slam_toolbox wird die Karte nicht gefunden. (Frame [map] does not exist.)

Hier handelt es sich wahrscheinlich um ein Problem des LDS. Ist dieser an den falschen USB-Port angeschlossen, sendet der `/scan`-Topic keine Daten. Ob das der Fall ist, kann durch das loggen des Outputs auf dem Topic `/scan` überprüft werden. Wenn der Befehl `ros2 topic echo /scan` keinen Output liefert, muss der LDS an einen anderen USB-Port angeschlossen werden. Die Konfiguration der Anschlüsse für diese Arbeit war folgende:

- oben links: leer
- oben rechts: OpenManipulator
- unten links: OpenCR
- unten rechts: LDS

Es kann ebenfalls daran liegen, dass der Start von `warehouse_bot_tb_launch.py` nicht erfolgreich war. Hier kann es passieren, dass der LDS auf den falschen USB-Port registriert wurde. Dann muss die Launch-Datei mit folgendem Befehl neu gestartet werden:

```
ros2 launch warehouse_bot warehouse_bot_tb_launch.py usb_port_open_
manipulator:=/dev/ttyUSB0 usb_port_lds:=/dev/ttyUSB1
```

Der Roboter erkennt während der Navigation Hindernisse hinter sich, obwohl die Informationen des LDS manuell gefiltert wurden.

Das Problem konnte in der vorliegenden Arbeit nicht eingehend untersucht werden, da es sich von selbst löste. Es wird jedoch vermutet, dass ein vollständiger Neustart des Systems, einschließlich des TurtleBots und der VM, eine mögliche Lösung darstellen könnte. Darüber hinaus wurde in Siebert [12] festgestellt, dass der LDS empfindlich auf Sonneneinstrahlung und Reflektionen reagiert. Eine mögliche Maßnahme zur Verbesserung der Messgenauigkeit könnte das Schließen der Jalousien sein, um störende Lichtquellen zu minimieren.

Die Karte liegt bei der Erstellung „schief“ in RViz und wird nicht im Mittelpunkt initialisiert (Siehe Abbildung 4.2a.)

Sicherstellen, dass der TurtleBot erst an der gewünschten Position eingeschalten wird. Wird er vorher an einer anderen Position eingeschalten und dann an die Zielposition, gestellt wird die Ausrichtung bereits durch die Sensorik beeinflusst.

Der OpenManipulator bewegt sich nicht.

Hier kann entweder der OpenManipulator nicht bewegt werden, weil er sich in der falschen Ausgangsposition befindet oder er liegt auf dem falschen USB-Port. Entweder den Arm in die korrekte Ausgangsstellung bringen oder die Launch-Datei auf dem TurtleBot neu starten mit fest konfigurierten USB-Ports:

```
ros2 launch warehouse_bot warehouse_bot_tb_launch.py usb_port_open_
manipulator:=/dev/ttyUSB0 usb_port_lds:=/dev/ttyUSB1
```

Der Bot erkennt die Produkte nicht.

Die Kamera erkennt die Produkte nur zuverlässig bei guten Lichtbedingungen. Bitte das Licht an der Tafel ausschalten um Gegenlicht zu vermeiden. Möglicherweise hilft es außerdem, die Jalousien leicht zu öffnen, um mehr Licht in den Raum zu lassen.

Konfigurationsdateien werden nach dem Build nicht gefunden.

Konfigurationsdateien werden standardmäßig im `share`-Verzeichnis gesucht. Dieses wird nach dem Build des Packages erstellt. Um sicherzustellen, dass alle Konfigurationsdateien dort verfügbar sind, müssen der `setup.py`-Datei des eigenen Packages folgende Zeilen aus Abbildung 7.4 hinzugefügt werden: (angenommen, die Konfigurationsdateien befinden sich im Verzeichnis `config`)

```
data_files=[
    ('share/ament_index/resource_index/packages', ['resource/'+package_name]),
    ('share/'+package_name, ['package.xml']),
    (os.path.join('share', package_name, 'config'), glob('config/*.yaml')),
    (os.path.join('share', package_name, 'launch'), glob('launch/*.py')),
],
```

Abbildung 7.4: Änderung in der *setup.py*-Datei um Konfigurationsdateien einzubinden.

7.3 Installationsanleitung OpenManipulator auf dem TurtleBot

Folgende Schritte sind notwendig, um den Workspace für den OpenManipulator einzurichten und die benötigten Pakete zur Verfügung zu stellen:

```
mkdir robotis_ws
cd robotis_ws
```

Quellcode der erforderlichen Pakete herunterladen (Evtl. Fehlt das Dynamixel SDK, es sollte im Normalfall aber auf der VM installiert sein):

```
git clone -b ros2 https://github.com/ROBOTIS-GIT/dynamixel-workbench.git
git clone -b humble-devel https://github.com/ROBOTIS-GIT/open_manipulator.
git
git clone -b ros2 https://github.com/ROBOTIS-GIT/open_manipulator_msgs.git
git clone -b ros2 https://github.com/ROBOTIS-GIT/open_manipulator_
dependencies.git
git clone -b ros2 https://github.com/ROBOTIS-GIT/robotis_manipulator.git
```

Nun müssen die Pakete im `robotis_ws`-Verzeichnis gebaut werden:

```
cd ~/robotis_ws
colcon build --symlink-install
```

Es können einige Warnungen auftreten, diese können jedoch ignoriert werden.
Als nächstes müssen die udev-Regeln erstellt werden:

```
ros2 run open_manipulator_x_controller create_udev_rules
```

Die Ausführung ist erfolgreich, wenn der Befehl `cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer` den Wert `1` ausgibt.
Gibt der Befehl das gewünschte Ergebnis zurück, befindet sich der OpenManipulator auf ttyUSB0. Sollte die Datei nicht existieren, liegt der OpenManipulator (wie im Fall für diese Arbeit) auf ttyUSB1. Dann sollte der Befehl `cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer` den Wert `1` ausgeben.
Falls der OpenManipulator auf `ttyUSB1` liegt, muss der korrekte USB-Port beim Start angegeben werden (Standardwert: `/dev/ttyUSB0`):

```
ros2 launch open_manipulator_x_controller open_manipulator_x_controller.
launch.py usb_port:=/dev/ttyUSB1
```

Zusätzlich sollte in der `open_manipulator_x_controller_modified.launch.py` Datei ein eigener Namespace für den OpenManipulator definiert werden und der Default-Wert für den OpenManipulator auf `'/dev/ttyUSB1'` geändert werden (falls er auf ttyUSB1 liegt). Der Code der veränderten Datei ist in Abbildung 7.5 dargestellt.

```python
def generate_launch_description():
    # Parameters
    usb_port = LaunchConfiguration('usb_port', default='/dev/ttyUSB1') # change
    baud_rate = LaunchConfiguration('baud_rate', default='1000000')
    param_dir = LaunchConfiguration(
        'param_dir',
        default=os.path.join(
            get_package_share_directory('open_manipulator_x_controller'),
            'param',
            'open_manipulator_x_controller_params.yaml'))

    return LaunchDescription([
        DeclareLaunchArgument(
            'usb_port',
            default_value=usb_port,
            description='Connected USB port'),

        DeclareLaunchArgument(
           'baud_rate',
           default_value=baud_rate,
           description='Set Baudrate'),

        DeclareLaunchArgument(
            'param_dir',
            default_value=param_dir,
            description='Specifying parameter direction'),

        Node(
            namespace='open_manipulator', # new
            package='open_manipulator_x_controller',
            executable='open_manipulator_x_controller',
            name='open_manipulator_x_controller',
            arguments=[usb_port, baud_rate],
            parameters=[param_dir],
            output='screen',
            remappings=[ # new
                ('/joint_states', '/open_manipulator/joint_states'),
            ])
    ])
```

Abbildung 7.5: Modifizierte Launch-Datei für *open_manipulator_x_controller.launch.py*.

7.4 Anpassen der Position des LDS in der URDF Datei des TurtleBots

Diese Anleitung beschreibt, wie die Position des LDS auf dem TurtleBot in der entsprechenden URDF-Datei angepasst werden kann.
Der einfachste Ansatz besteht darin, sowohl die URDF-Datei als auch die zugehörigen Launch-Dateien in ein eigenes Package zu kopieren und dort anzupassen. In diesem Fall betrifft dies die `turtlebot3_state_publisher.launch`-Datei, welche jedoch selbst von der `robot.launch.py` aufgerufen wird. Daher muss auch diese kopiert und angepasst werden.
Beide Dateien gehören zum Package `turtlebot3_bringup` und befinden sich dort im Verzeichnis `launch`. Die URDF-Datei selbst ist Bestandteil des Packages `turtlebot3_description` und kann im Verzeichnis `urdf` gefunden werden. Die modifizierten Launch-Dateien befinden sich im Github Repository dieser Arbeit unter `warehouse_bot/urdf`.
Um die URDF-Datei im eigenen Package verwenden zu können, muss sie in ein separates Verzeichnis namens `urdf` kopiert werden. Außerdem muss in der `setup.py`-Datei des Packages die Zeile aus 7.6 hinzugefügt werden. Diese beiden Schritte stellen sicher, dass URDF-Dateien während des Build-Prozesses korrekt behandelt werden.

```
data_files=[
    ('share/ament_index/resource_index/packages', ['resource/'+package_name]),
    ('share/'+package_name, ['package.xml']),
    (os.path.join('share', package_name, 'urdf'), glob('urdf/*.urdf')), # new
    (os.path.join('share', package_name, 'launch'), glob('launch/*.py')),
],
```

Abbildung 7.6: Änderung in der *setup.py* Datei um URDF-Dateien einzubinden.

In der URDF-Datei ist es notwendig, die Position des `scan_joint` anzupassen. Um den erforderlichen Versatz des LDS zu bestimmen, kann die Differenz zwischen der korrekten Position des LDS auf einem TurtleBot ohne OpenManipulator und der Position des LDS auf dem aktuellen Roboter berechnet werden. In Fall dieser Arbeit beträgt die Differenz +6,6 cm. Somit muss der LDS um +0,066 entlang der x-Achse verschoben werden. Die für diese Arbeit notwendige Änderung ist in 7.7 dargestellt.

```
<joint name="scan_joint" type="fixed">
  <parent link="base_link"/>
  <child link="base_scan"/>
  <origin xyz="0.002 0 0.122" rpy="0 0 0"/>  <!--new-->
  <!--<origin xyz="-0.064 0 0.122" rpy="0 0 0"/> original-->
</joint>
```

Abbildung 7.7: Änderung in der URDF-Datei um die Position des LDS auf dem TurtleBot anzupassen.