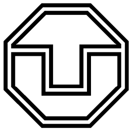


Fakultät Bauingenieurwesen Institut für Bauinformatik

Steuerung gängiger Industrieroboter mit ROS

Projektarbeit, Nr. 4649156

Shaowen Han

Dresden, den 21. Februar 2021

Abkürzungsverzeichnis

ROS	Robot Operating System
URDF	Unified Robot Description Format
XML	Extensible Markup Language
Makefile	Computer-Kompilierungsregeln
CMake	Regeln für die Kompilierung von Programmen auf Basis der ROS-Plattform
Rviz	3D visualization tool for ROS.
SLAM	Simultane Lokalisierung und Kartierung
RBPF	Rao-Blackwellised Particle Filter
SRI	formerly Stanford Research Institute
KDL	Kinematics Dynamics Library

Inhaltsverzeichnis

1 Einleitung 1

1.1 Motivation und Problemstellung der Arbeit 1

1.2 Aufbau der Arbeit 2

2 Stand der Technik 3

2.1 Robotik 3

2.2 Ubuntu Systems and Virtual Machines 6

2.3 Robot Operating System (ROS) 7

2.4 Rviz 3D-Visualisierungsplattform 8

3 Anwendung von ROS 10

3.1 Einige übliche Steuerbefehle in ROS 10

3.1.1 ROS-Shell-Befehle 10

3.1.2 ROS-Ausführungsbefehl 10

3.1.3 ROS-Meldungsbefehl 11

3.2 Catkin Arbeitsbereich 12

3.2.1 Initialisieren des Catkin-Arbeitsbereichs 12

3.2.2 Aufbau Vorstellung des Systems 13

3.3 Catkin Package-Software 16

3.3.1 Aufbau des Package 16

3.3.2 Package-Erstellung 17

3.4 Verwalten von CMakeLists.txt 18

3.4.1 CMakeLists.txt Funktion 18

3.4.2 CMakeLists.txt Schreibverfahren 18

3.5 Package.xml 19

3.5.1 Die Funktion von package.xml 19

3.5.2 Package.xml Schreibverfahren 20

3.6 ROS-Kommunikationsarchitektur 20
3.6.1 Node & Master 21
3.6.2 Start Master und Node 21
3.7 Launch Datei 22
3.8 Topic 23
3.9 Message 24
3.10 Service 24
4 Auslegung eines Robotermodells Roboter-Wagen 26
4.1 Erstellen eines Hardware-Beschreibungspakets im Arbeitsbereich .. 26
4.2 Erstellen die Beschreibung des Modells Robot.urdf 27
4.2.1 URDF-Beschreibungsdatei für Wagenkörper 28
4.2.2 Vorbereitung der Modelldaten 28
4.3 Launch-Datei erstellen 31
4.3.1 Der Node-Abschnitt der Launch-Datei 31
4.3.2 Unterschied und Zusammenhang zwischen joint_state_publisher und robot_state_publisher 32
4.3.3 Ausführen der Rviz-Visualisierungsoberfläche 33
4.4 Inbetriebnahme Durchführung 33
4.5 Tastatur-Steuerung 34
4.5.1 Publisher 35
4.5.2 Subscriber 35
4.5.3 Beschreibung robot_base_control.py 36
4.6 Node-Relationship-Diagramm 37
5 Auslegung des Roboterarms 38
5.1 Erstellung des Modells Roboterarm.urdf 39
5.1.1 URDF-Beschreibungsdatei für Roboterarm 39
5.1.2 URDF-Beschreibungsdatei für Robotergreifer 40

5.1.3 Importieren von SolidWorks fertigen *.dae-Dateien 42

5.2 Modifizierung der Launch-Datei 43

5.3 Automatische Steuerung mit Roboterarmwagen 44

5.3.1 Open-Loop-System 45

5.3.2 Closed-Loop (Feedback Control) Systems 47

5.4 Automatische Steuerung mit Roboterarm 50

6 Implementierung der Software 52

7 Analyse der Ergebnisse 56

7.1 Trajektorien des Roboterwagens 56

7.2 Übliche Slam-Algorithmen in ROS 60

7.3 Kinematische Modellierung und Simulationsanalyse eines Roboterarms 62

7.3.1 positive kinematische Lösung und Analyse 63

7.3.2 Inverse Kinematik lösen und analysieren 64

8 Schlussbetrachtung 64

8.1 Zusammenfassung 64

8.2 Ergebnisse der Arbeit 65

8.3 Ausblick 65

9 Literaturverzeichnis 67

10 Abbildungsverzeichnis 71

11 Tabellenverzeichnis 72

1 Einleitung

1.1 Motivation und Problemstellung der Arbeit

Mit dem Bevölkerungszuwachs werden immer mehr Bauprojekte zur Schaffung von Wohnraum benötigt [1]. Zusätzlich werden auch Infrastrukturen zur Modernisierung der Gesellschaft aufgrund der zunehmenden Nutzung der öffentlichen Verkehrsmittel nachgefragt. Die Umsetzung von Baumaßnahmen wird in Folge des Fachkräftemangels und der steigenden Lohnkosten für Bauarbeiter erschwert und verzögert. Begründet durch fehlenden Nachwuchs und der schweren körperlichen Arbeit, gehört der Beruf des Mauers zu den am schwersten zu besetzenden Positionen im Baugewerbe. [2] Mit der rasanten Entwicklung der Automatisierungstechnik in verschiedenen Bereichen, insbesondere im Bauwesen, hat dies dazu geführt, dass einige Unternehmen bei der Durchführung des Bauprozesses vollautomatisch arbeiten. In anderen Worten: Die Unternehmen gleichen den Mangel an Arbeitskräften durch die Einführung von Robotern aus.

Um die fehlende Arbeitskraft zu ersetzen bietet sich der Einsatz von autonomen Baurobotern an. Einsatzfelder wären z.B. der Transport von Baumaterial und die automatische Durchführung von Bauprozessen. Durch den Einsatz von Baurobotern können auch die Risiken von Personenschäden bei der Durchführung von Bauarbeiten und die Personalkosten reduziert werden. Außerdem wird die Effizienz des Bauens gesteigert und die Bauzeiten werden verkürzt. Umfangreiche Bauroboter wurden in den letzten Jahren mit Hilfe von Automatisierungstechniken intensiv weiterentwickelt und spielen heutzutage eine immer wichtigere Rolle. [3]

Im Rahmen dieser Arbeit sollen Industrieroboter verschiedener Hersteller Betriebssystem über das ROS gesteuert werden. Die Plattform wird von ROS (Robot Operating System) betrieben, einer Open-Source Plattform, die es ermöglicht Roboter unterschiedlicher Hersteller zu steuern. Dazu soll in einem ersten Schritt untersucht und dokumentiert werden, wie Industrieroboter mit ROS gesteuert werden können. Der zweite Schritt

besteht darin, die für die Robotersteuerung erforderliche ROS-Arbeitsumgebung auf einer virtuellen Maschine einzurichten.

Die ROS-Arbeitsumgebung soll zur Simulation von einzelnen Industrierobotern genutzt werden. Ideale Demonstratoren sind z.B. einfache "Pick and Place"-Aufgaben und Simulationen von mobilen Robotern.

1.2 Aufbau der Arbeit

Das Ziel dieser Arbeit ist es, eine Simulation eines generischen Roboters zu entwerfen, um die Simulationssteuerung eines Maschinenwagens mit Roboterarm zu realisieren. In diesem Kapitel wird eine Einführung in das Robot Operating System gegeben und anschließend die Verwendung des ROS Schritt für Schritt beschrieben. Außerdem werden einige der von ROS unterstützten Plugins, Gazebo und Rviz, eingeführt, die auch in der folgenden Programmierung verwendet werden müssen.

Für die Aufgabe muss ein Roboter in der Simulationsumgebung simuliert werden, damit er sich selbstständig bewegen und Greifaktionen durchführen kann.

Das Steuerungssystem basiert auf dem ROS-Betriebssystem. Bei der Verwendung dieses neuen Systems ist es notwendig, den Umgang mit der Roboterplattform zu erlernen, um die Simulation auf der ROS-Plattform zu entwerfen und das Steuerungssystem zu entwickeln, sowie Informationen auf der ROS-Plattform zu übertragen. Darüber hinaus soll gelernt werden, wie die Parameter des simulierten Objekts in der Simulation eingestellt werden können, einschließlich der Zielgeschwindigkeit und der Flugbahn des Wagens. Die Ergebnisse der Arbeit sollen mit Hilfe der Simulations-Plugins demonstriert werden.

Um die oben genannten Ziele zu erreichen, werden im Rahmen dieser Arbeit die technischen Grundlagen und Untersuchungen durchgeführt.

Nach dem einleitenden Teil wird jedes Programm einzeln detailliert erklärt. Die Konstruktion des Wagens erfolgt zunächst durch die Codierung von Unified Robotik Description Format <URDF>, wobei das Programm für die Karosseriestruktur des Wagens sowie für die Räder codiert wird. Der nächste Schritt ist, wie man den Wagen

dazu bringt, sich in eine bestimmte Richtung zu bewegen, indem man den Wagen über die Tastatur steuert und in der Lage ist, die Schleife automatisch zu schließen. Basierend auf der Untersuchung wird die Eigenschaft des Wagens durch das URDF-Modul so eingerichtet, dass er im Simulator gesteuert und getestet werden kann. Der Schwerpunkt der Steuerungsentwicklung liegt darin, den Wagen im Rviz-Simulator durch die Programmierung von Code-Paaren zum Vorwärts- und Rückwärtsfahren sowie zum Drehen über die Tastatur zu bringen. Das Steuerungssystem ermöglicht die Interaktion zwischen dem Roboter und dem Bediener. Die automatische omnidirektionale Bewegung des Roboters in einer Ebene wird durch Routenplanung erreicht.

Der nächste Schritt ist die Konstruktion des Roboterarms, der wie der Wagen in URDF modelliert und durch den Code gesteuert werden kann, um die Greifaktion auszuführen. Schließlich vergleichen und analysieren wir die Ergebnisse des Programms durch eine Demonstration in Rviz.

Abschließend wird die Funktionsfähigkeit des entworfenen Systems durch Tests überprüft.

2 Stand der Technik

2.1 Robotik

Seit der Geburt von Robotern hat der Mensch diese nach ihrem Intelligenzgrad in drei Generationen eingeteilt: Lehrroboter gehören zur ersten Generation von Robotern, die weniger intelligent sind, aber aufgrund ihres niedrigen Preises die meisten kommerziellen Anwendungen haben. Perzeptive Roboter mit eigenen Sensoren sind die zweite Generation von Robotern, die ihren eigenen Zustand und externe Umgebungsinformationen erfassen und eine rückgekoppelte Bewegungssteuerung realisieren können. Die dritte Generation intelligenter Roboter hat eine stärkere Wahrnehmungsfähigkeit und die eigenständige Lern- und Entscheidungsfähigkeit, die den ersten beiden Robotergenerationen fehlt, und sie enthält ein komplexes intelligentes Informationsverarbeitungssystem. [4]

Industrieroboter sind die mit Abstand am häufigsten eingesetzten Roboter. [5] Sie wurden entwickelt, um Menschen bei sich wiederholenden oder gefährlichen Produktionsaufgaben zu ersetzen, wie z. B. Karosserieschweißen, Karosserielackierung, Teilemontage usw. Die kontinuierliche Entwicklung der Robotik hat dazu geführt, dass Roboter nicht mehr nur in der industriellen Fertigung eingesetzt werden. [6] Immer mehr Roboter werden in das tägliche Leben der Menschen eingebunden, die unter dem Begriff Serviceroboter zusammengefasst werden. [7] Generell gehören Serviceroboter zur dritten Generation von Robotern, die über intelligente Systeme verfügen, die in den ersten beiden Robotergenerationen nicht zu finden sind. [8] Mobile Serviceroboter haben die Fähigkeit, sich frei in der menschlichen Lebensumgebung zu bewegen. **Abbildung 2-1** zeigt drei typische mobile Serviceroboter: Lieferroboter, Bodenreinigungsroboter und Unterhaltungsroboter, die bequem und komfortabel für das Leben der Menschen sind. [9] Diese drei Roboter gehören jedoch zu den traditionellen mobilen Servicerobotern, die menschlichen Anweisungen folgen können, um bestimmte Aufgaben zu erfüllen, aber die Nachteile haben, dass sie nur eine einzige

a) Lieferroboter   b) Kehrroboter   c) Unterhaltungsroboter

Abbildung 2-1 Drei traditionelle mobile Serviceroboter

Funktion haben, nicht genügend eigene Ausführungsfähigkeiten und eine geringe Intelligenz.

Der Care-O-bot ist ein spezieller Typ eines mobilen Serviceroboters [10], der die Vorteile von mobilen Robotern und fest montierten Robotern kombiniert, um den Arbeitsbereich des Roboters erheblich zu erweitern. Er kann Dinge tun, die herkömmliche mobile Serviceroboter nicht tun können, wie z. B. Gegenstände aushändigen, Kühlschranktüren öffnen und sich auf einer Fläche befindliche Gegenstände aufheben, wie in **Abbildung 2-2** demonstriert, die den mobilen Manipulator Arm des Care-O-bot beim Ausliefern von Bier an einen Benutzer zeigt. [11]

Die Forschung im Bereich mobiler Manipulator-Roboterarme wurde in den USA und Europa bereits früher durchgeführt, und auch der Stand der Technik im Zusammenhang mit Robotern ist weit voraus. Mit der Entwicklung der Robotik-Industrie sind immer mehr fortschrittliche mobile Manipulator-Roboterarme im Blickfeld des Menschen erschienen,

Abbildung 2-3 Care-O-Bot bringt Bier

Abbildung 2-3 HERB und Care-O-bot Roboter [38]

wie in **Abbildung 2-3** für einige der fortschrittlichen mobilen Manipulator-Roboterarme dargestellt.

In **Abbildung 2-3** ist der mobile Manipulator-Roboterarm HERB (Home Exploring Robotic Butler) dargestellt, der gemeinsam von Pittsburgh Laboratory und Inter, Inc. in den USA entwickelt wurde [12]. Der Roboter umfasst Komponenten wie die mobile Segway-Plattform, den überflüssigen Barrett WAM-Roboterarm mit 7 Freiheitsgraden, die Barrett-Drei-Finger-Handkralle mit 7 Freiheitsgraden und den SICK-Laserentfernungssensor. Er ist in der Lage, in Echtzeit Karten zur Positionierung zu erstellen und sich anhand der erstellten Karten frei im Raum zu bewegen. Außerdem kann der Roboter Aufgaben wie das Öffnen von Türen, das Aufnehmen von Gegenständen und das Reinigen von Tischen übernehmen. [13] Letzteres ist die vom Fraunhofer IPA entwickelte mobile Roboterplattform Care-O-bot, die wie der HERB dem Menschen im täglichen Leben dienen soll. Sie wird seit mehr als 15 Jahren vom Fraunhofer IPA entwickelt und befindet sich mittlerweile in der dritten Generation der Care-O-bot-Serie. Der neueste Care-O-bot Roboter verfügt über eine omnidirektionale Plattform mit 4-Rad-Antrieb, die dafür sorgt, dass der gesamte Roboter auch in engen Räumen flexibel bewegt werden kann. Der leichte Roboterarm mit sieben Freiheitsgraden und die Roboterhand mit drei Fingern erleichtern dem Roboter das Greifen von Alltagsgegenständen. Der Roboter Care-O-bot nutzt Kinect-, Kraft- und haptische

Sensoren, um Informationen über seine Umgebung zu erfassen. Er erkennt die Position von Hindernissen in der Umgebung und weicht ihnen flexibel aus. [11]

2.2 Ubuntu Systems and Virtual Machines

Ubuntu ist ein Desktop-basiertes Linux-Betriebssystem, dessen Name vom Zulu- oder Hausa-Wort "Ubuntu" im südlichen Afrika stammt, was "Menschlichkeit" und "Ich existiere, denn alle existieren" bedeutet. [14] Ubuntu basiert auf der Debian-Distribution und der Gnome-Desktop-Umgebung, die seit Version 11.04 zugunsten von Unity aufgegeben wurde. Die Wahrnehmung, dass Linux schwierig zu installieren und zu bedienen ist, gehört mit dem Aufkommen von Ubuntu der Vergangenheit an. Ubuntu hat auch eine große Community, von der Benutzer leicht Hilfe bekommen können. Seit Ubuntu 18.04 LTS verwendet die Ubuntu-Distribution wieder die Desktop-Umgebung

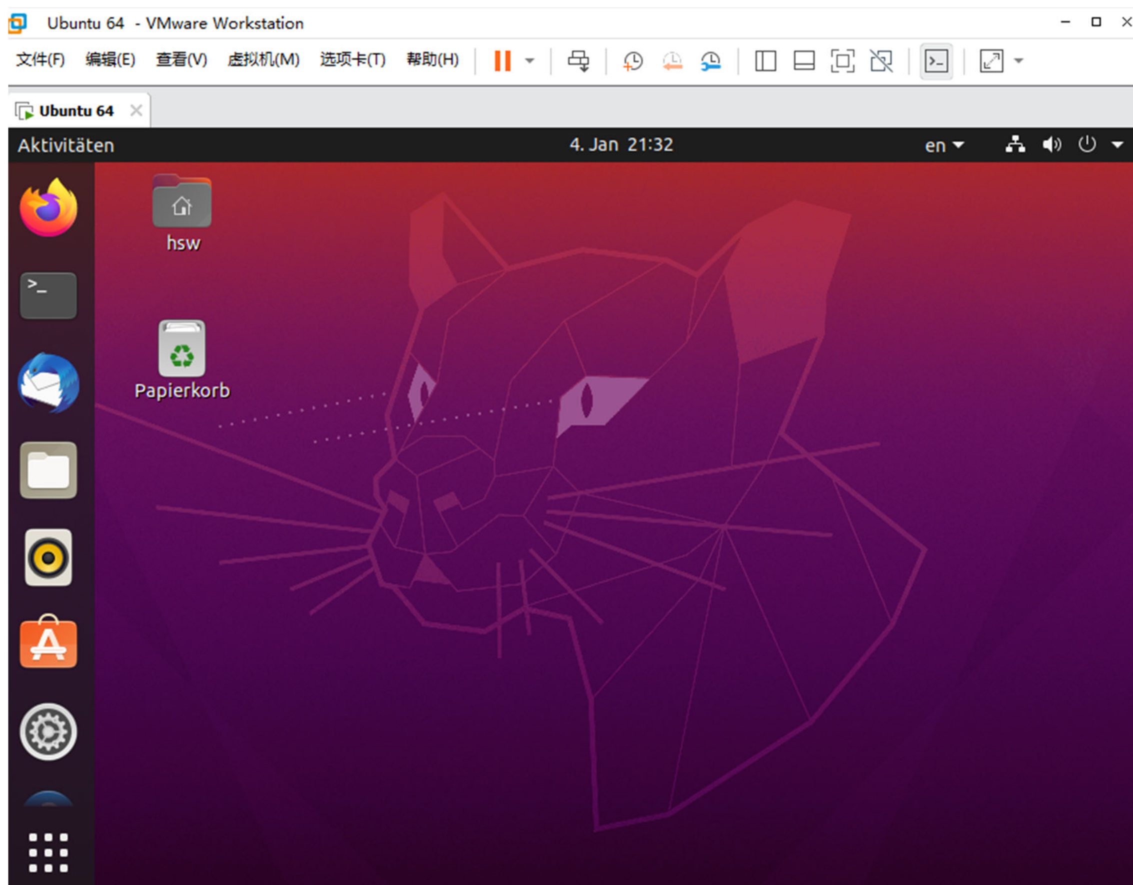


Abbildung 2-4 Ubuntu im VMware

GNOME3 [15]. Für den ROS Noetic, den wir jetzt verwenden, ist Ubuntu Version 20 die beste Version und er kann problemlos auf Win10-Systemen mit einer virtuellen Maschine laufen.

Ubuntu benötigt, wie Windows, einen physischen oder virtuellen Rechnerhost, um auf einem Computer zu laufen. Hier wird eine virtuelle VMware-Maschine verwendet, die von der Universität zur Verfügung gestellt wird und die auf den meisten modernen Computern laufen kann. Die Verwendung einer virtuellen Maschine ermöglicht es mehr als zwei Betriebssysteme auf demselben PC zu verwenden und jederzeit dazwischen zu wechseln, ohne dass eine Partitionierung oder ein Neustart erforderlich ist. Außerdem ist es möglich, die Betriebsumgebung der verschiedenen Betriebssysteme und alle darauf installierten Anwendungen und Daten vollständig zu isolieren und zu schützen sowie die Dateien der beiden Betriebssysteme zu kopieren und einzufügen. [16] Das über eine virtuelle VMware-Maschine geladene Ubuntu-System wird in der **Abbildung 2-4** gezeigt.

2.3 Robot Operating System (ROS)

Die Robotik kann nicht ohne die Unterstützung eines Betriebssystems entwickelt werden, genau wie die Entwicklung von Computern und Smartphones. In den letzten Jahren hat eine wachsende Zahl von Wissenschaftlern brauchbare Softwarepakete für ROS entwickelt, die auch Lösungen für spezifische Probleme enthalten. Viele Unternehmen bieten auch ROS-Schnittstellen in den neuen Robotern, die sie produzieren an. Mit dem weit verbreiteten Einsatz von ROS-Schnittstellen wird erwartet, dass sich ROS zu einem neuen Standard in der Robotik entwickelt und die Entwicklungsgeschwindigkeit erhöht [17] [18]. TurtleBot ist ein preiswerter individueller Roboter. Mit TurtleBot ist es möglich, einen Roboter zu bauen, der um ein Haus herumfahren kann, im 3D-Modus sehen kann und genug Leistung hat, um verschiedene Anwendungen zu erstellen. ROS bietet eine Standard-Betriebssystem-Umgebung, einschließlich der zugrunde liegenden Gerätesteuerung, Hardware-Abstraktion, Inter-Prozess-Kommunikation und Software-Management. ROS ist ein verteiltes Verarbeitungssystem, und jedes Funktionsmodul ist unabhängig voneinander anwendbar. Das erlaubt, dass Module beliebig hinzugefügt oder entfernt werden können, ohne das Gesamtsystem zu beeinflussen. Außerdem bietet

es eine Reihe von Werkzeugprogrammen und Bibliotheken zum Erfassen, Erstellen, Schreiben und Ausführen von Programmen für die Integration mehrerer Computer. Es ist auch das algorithmische Zentrum des Roboters und für Aufgaben wie den Empfang und die Analyse serieller Daten für die Fahrgestellsteuerung, die Verarbeitung und Zusammenführung von Sensordaten zur Erfassung von Umgebungsinformationen, die Synthese und Ausgabe von Befehlen zur Bewegungssteuerung des Roboterfahrgestells und die Implementierung von Algorithmen zur Roboterpositionierung, Navigation, Hindernisvermeidung und intelligenten Aufladung zuständig. Jedes ROS-Programm läuft als Knoten, der das Äquivalent einer ausführbaren Datei ist und dazu verwendet werden kann, bestimmte Daten oder Fähigkeiten bereitzustellen, wie z. B. das Lesen von Daten aus einem Sensor.

2.4 Rviz 3D-Visualisierungsplattform

Im Robotersystem gibt es eine große Menge an Daten, die bei der Berechnung oft in Datenform vorliegen, wie z. B. RGB-Werte von 0 bis 255 in Bilddaten. Aber solche Werte

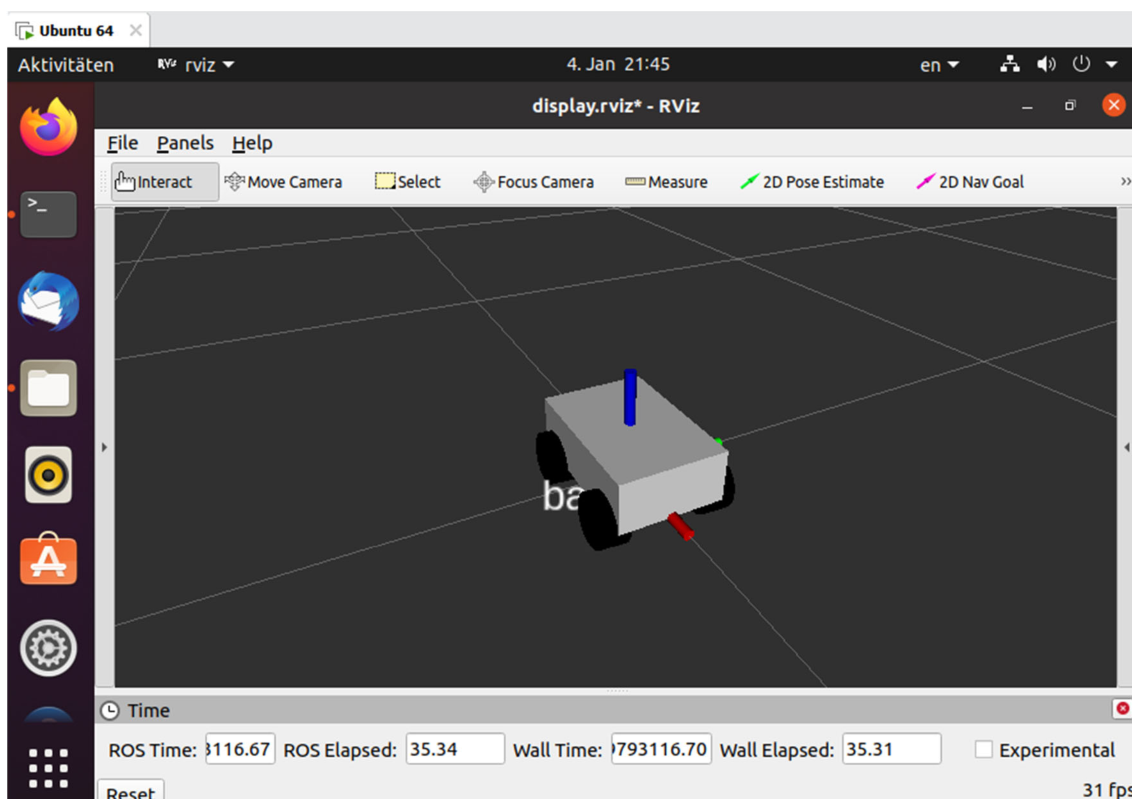


Abbildung 2-5 Rviz-Emulator

in Datenform sind für Entwickler oft nicht förderlich, um zu erkennen, was die Daten beschreiben. Daher ist es oft notwendig, die Daten zu visualisieren und darzustellen, z. B. durch Visualisierung von Robotermodellen, Visualisierung von Bilddaten, Visualisierung von Kartendaten usw. ROS erfüllt die Anforderungen an die Visualisierung von Robotersystemen, indem es den Benutzern Rviz zur Verfügung stellt, eine 3D-Visualisierungsplattform, die eine Vielzahl von Daten anzeigt. [19]

Rviz ist ein 3D-Visualisierungstool, das gut mit verschiedenen Robotik-Plattformen kompatibel ist, die auf dem Software-Framework ROS basieren. In Rviz können beliebige physische Objekte wie Roboter, umgebende Objekte usw. in Bezug auf Größe, Masse, Position, Material, Gelenke und andere Attribute mithilfe von XML beschrieben und in der Oberfläche dargestellt werden. Außerdem kann Rviz Informationen über die Sensoren des Roboters, den Bewegungsstatus des Roboters, Änderungen in der Umgebung usw. in Echtzeit grafisch darstellen.

Kurz gesagt, Rviz hilft den Entwicklern, alle überwachbaren Informationen grafisch darzustellen, und die Entwickler können auch das Verhalten des Roboters mit Schaltflächen, Schiebereglern, Werten usw. unter der Steuerungsoberfläche von Rviz steuern. Der Rviz-Emulator, der über die ROS-Plattform geöffnet wird, ist in der **Abbildung 2-5** dargestellt (s.o.).

3 Anwendung von ROS

3.1 Einige übliche Steuerbefehle in ROS

3.1.1 ROS-Shell-Befehle

Tabelle 3-1 ROS-Shell-Befehle

Befehle	Funktionalitäten
roscd	wechselt in das angegebene ROS-Paketverzeichnis
rosls	Anzeige der Dateien und Verzeichnisse des ROS-Pakets
rosed	Bearbeiten Sie die Dateien des ROS-Pakets

Zu den üblichen ROS-Shell-Befehlen gehören roscd, rosls und rosed. Der Befehl ist roscd [Name des Funktionspakets] wird verwendet, um in ein Funktionspaket zu gelangen, ohne es mit cd Schicht für Schicht finden zu müssen;

Rosls [Name des Funktionspakets]. Dieser Befehl ist gleichwertig zu ROSCD+LS. Es ist einfacher und schneller, die Liste der Dateien in ROS-Funktionspaketen anzuzeigen;

Rosed [Funktionspaketname] [Dateiname]. Es dient zur Bearbeitung einer bestimmten Datei in einem Funktionspaket und hat zudem den Vorteil, dass es schnell und einfach zu ändern ist.

3.1.2 ROS-Ausführungsbefehl

Tabelle 3-2 Ausführungsbefehl

Befehle	Funktionalitäten
roscore	Master einschalten (ROS-Namensdienst)
rosrun	Einzelne Node ausführen

roslaunch	Mehrere Node ausführen und Laufoptionen festlegen
rosclean	ROS-Protokolldateien prüfen oder löschen

Der Roscore ist die Verbindungsinformation in der Nachrichtenkommunikation zwischen dem laufenden Masterknoten, der die Node verwaltet. Wenn das Masterprogramm von ROS ausgeführt ist, erkennt dies den übergreifenden Satz von Befehlen, mit denen er arbeitet, damit der Computer den Satz aller Befehle von ROS ausführen kann.

Rosrun [Funktionspaketname] [Knotenname]. Führt den Befehl für einen der Node im angegebenen Funktionspaket aus (rosrun rqt_graph1 rqt_graph2 zur Anzeige des aktuellen Systembetriebs).

Roslaunch [Name des Funktionspakets] [Dateiname des Starts]. Ähnlich wie der dat-Batch-Befehl ist dies ein Befehl, der eine oder mehrere Ausführungsoptionen im angegebenen Funktionspaket ausführt.

Rosclean [Option]. Beim Ausführen von roscore werden Aufzeichnungen für alle Node in die Protokolldatei geschrieben, die im Laufe der Zeit regelmäßig mit dem Befehl rosclean entfernt werden muss.

3.1.3 ROS-Meldungsbefehl

Tabelle 3-3 Meldungsbefehl

Befehle	**Funktionalitäten**
rostopic	Anzeige von ROS-Themeninformationen
rosservice	Anzeige der ROS-Dienstinformationen
rosnode	Anzeige von ROS-Knoteninformationen

Der Befehl rostopic liefert Informationen über ROS-Themen. Der Befehl rosservice dient dazu, Informationen über die Dienste in der aktuellen ROS-Umgebung anzuzeigen. Der Befehl rosnode ist eine Methode zur Anzeige von Informationen in Bezug auf einen ros-

Knoten und wird in der Regel verwendet, wenn ein Bot erstellt und das ordnungsgemäße Funktionieren eines bestimmten Knotens überwacht wird.

Wenn am Ende dieser Befehle list hinzugefügt wird, werden alle aktuell funktionierenden Knoten-Themen-Dienste aufgelistet. Der Befehl rostopic list wird z. B. häufig verwendet, um alle aktuell abonnierten und veröffentlichten Themen aufzulisten und um die für Unterbefehle erforderlichen Parameter in der Nachrichtenliste zu sehen.

3.2 Catkin Arbeitsbereich

Der Catkin-Arbeitsbereich ist ein Verzeichnis zum Erstellen, Ändern und Kompilieren von Catkin-Package. Jede ROS-Quelldatei muss auf dem Catkin Arbeitsbereich kompiliert werden, bevor sie auf dem Computer ausgeführt werden kann. [20] Die Compiler unter Linux sind gcc und g++. Als die Quelldateien zunahmen, wurden die direkten gcc/g++-Befehle ineffizient und es wurde begonnen, Makefile zum Kompilieren zu verwenden. CMake ist ein Generator für das Werkzeug make, ein übergeordnetes Werkzeug, das den Kompilier-Build-Prozess vereinfacht, große Projekte verwaltet und gut skalierbar ist. Für eine so große Plattform wie ROS wurde CMake verwendet, und ROS erweiterte CMake, wodurch das Catkin-Kompilierungssystem entstand. Der Catkin-Arbeitsbereich ist ein Repository für ROS-Projekte zur einfachen Systemorganisation und zum Wiederaufrufen von den Dateien. Er wird in der visuellen Benutzeroberfläche als Ordner dargestellt. Der ROS-Code, der vom Anwender selbst geschrieben wird, wird normalerweise im Arbeitsbereich platziert, und in diesem Abschnitt wird der Aufbau des Catkin-Arbeitsbereichs beschrieben, der die Installation der Catkin-Spaces und des Catkin-Paket umfasst.

3.2.1 Initialisieren des Catkin-Arbeitsbereichs

Nachdem das Catkin-Kompilierungssystem vorgestellt wurde, wird ein Catkin-Arbeitsbereich erstellt. Zunächst wird auf unserem Computer ein Anfangspfad zu *catkin_ws/* erstellt, der die höchste Gliederungsebene des catkin-Arbeitsbereichs darstellt. mit dem folgenden Befehl wird eingegeben, um die Ersterstellung abzuschließen:

```
1. $ mkdir -p ~/catkin_ws/src
2. $ cd ~/catkin_ws/
3. $ catkin_make                #Initialisierung des Arbeitsbereichs
```

Die erste Codezeile erzeugt direkt den Ordner src auf der zweiten Ebene, wo die ROS-Pakete abgelegt werden. Die zweite Codezeile sorgt dafür, dass der Prozess den Arbeitsbereich betritt, danach folgt catkin_make.

3.2.2 Aufbau Vorstellung des Systems

Auf den ersten Blick sieht der Workspace von Catkin extrem komplex aus, aber tatsächlich ist die Struktur des Catkin-Workspace sehr übersichtlich. Im Arbeitsbereich verwenden wir den Befehl *tree*, um die Dateistruktur anzuzeigen:

```
1. $ cd ~/catkin_ws
2. $ sudo apt install tree
3. $ tree
```

Die Ergebnisse sind nachstehend aufgeführt:

```
1.  ─ build
2.  |   ├── catkin
3.  |   |   └── catkin_generated
4.  |   |       └── version
5.  |   |           └── package.cmake
6.  |   ├──
7.  ......
8.
9.  |   ├── catkin_make.cache
10. |   ├── CMakeCache.txt
11. |   ├── CMakeFiles
12. |   |   ├──
13. ......
14.
```

```
15.  ├── devel
16.  |   ├── env.sh
17.  |   ├── lib
18.  |   ├── setup.bash
19.  |   ├── setup.sh
20.  |   ├── _setup_util.py
21.  |   └── setup.zsh
22.  └── src
23.  └── CMakeLists.txt -> /opt/ros/Noetic/share/catkin/cmake/toplevel.cmake
```

Der *Tree-Befehl* zeigt die Struktur des Catkin-Arbeitsbereichs an, der aus drei Pfaden besteht: *src*, *build* und *devel*. [21] Diese drei Ordner sind der Standard für das Catkin-Build-System. Ihre spezifischen Abfolgen sind in **Abbildung 3-1** dargestellt.

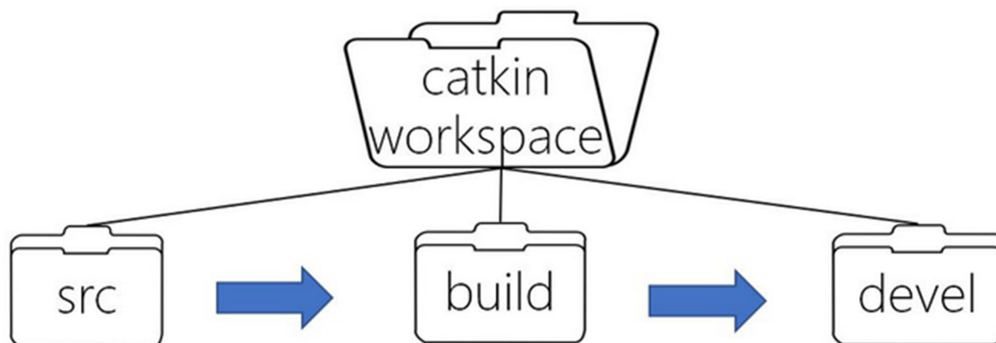


Abbildung 3-1 Arbeitsablauf bei der Kompilierung [21]

Die beiden letzten Pfade werden automatisch vom Catkin-System generiert und verwaltet, und wir beschäftigen uns normalerweise nicht mit der täglichen Entwicklung. Stattdessen verwenden wir hauptsächlich den src-Ordner, in dem die von uns geschriebenen ROS-Programme und ROS-Quellcode-Pakete gespeichert werden.

Zur Kompilierzeit findet das Catkin-Build-System rekursiv jedes Quellpaket unter src/ mit dem entsprechenden Kompiliervorgang. Daher können mehrere Quellcode-Pakete in denselben Ordner gelegt werden, wie nachfolgend in **Abbildung 3-2** abgebildet.

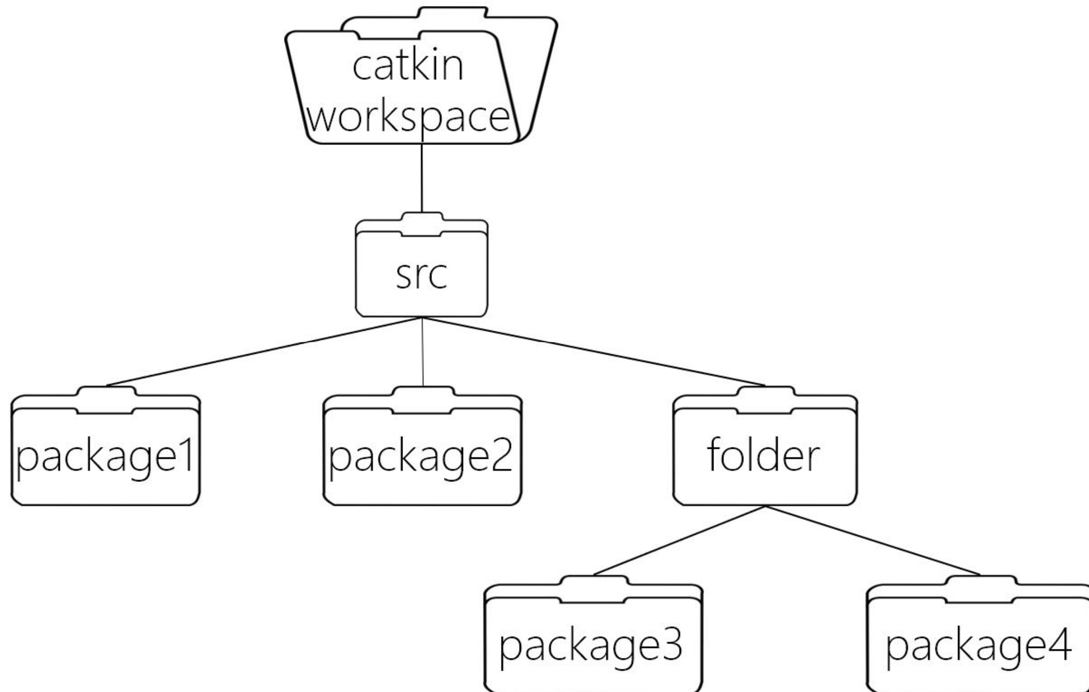


Abbildung 3-2 Quellcode-Paketverzeichnis [21]

Zusammenfassend dargestellt, ist der Catkin-Arbeitsbereich im Grunde die obige Struktur. Paket ist die Grundeinheit des Catkin-Arbeitsbereichs, das heißt wir schreiben den Code in ROS-Entwicklung, dann catkin_make, und das System wird die ganze Arbeit der Kompilierung und Paketerstellung selbst erledigen.

3.3 Catkin Package-Software

Die Definition von Paket in ROS ist spezifischer, es ist nicht nur das Paket unter Linux, sondern auch die Grundeinheit der Catkin-Kompilierung. Die Aufgabe, die catkin_make zum Kompilieren aufruft, ist ein ROS-Paket, was bedeutet, dass jedes ROS-Programm nur kompiliert werden kann, wenn es in einem Paket organisiert ist. Package ist folglich auch der Ort, an dem der ROS-Quellcode gespeichert wird. Jeder ROS-Code, egal ob C++ oder Python, muss in ein Package abgelegt werden, damit er kompiliert werden kann und korrekt läuft.

```
1.   ├── CMakeLists.txt      #Paketkompilierungsregeln (erforderlich)
2.   ├── package.xml         #Paketbeschreibungsinformationen (erforderlich)
3.   ├── src/                #Quellcode-Dateien
4.   ├── include/            #C++ Kopfzeilen
5.   ├── Skripte/            #Ausführbare Skripte
6.   ├── msg/                #Custom messages
7.   ├── srv/                #custom services
8.   ├── Modelle/            #3D-Modelldateien
9.   ├── urdf/               #urdf-Dateien
10.  ├── Launch/             #Launch-Dateien
```

3.3.1 Aufbau des Package

Das Paket wird durch CMakeLists.txt und package.xml definiert, daher sind diese beiden Teile für die Paketkompilierung unerlässlich. Das Catkin-Kompilierungssystem analysiert zunächst diese beiden Dateien, bevor es kompiliert.

- **CMakeLists.txt:** Definieret Paketname, Abhängigkeiten, Quell- und Zieldateien und andere Kompilierregeln, die unverzichtbare Bestandteile des Pakets sind.
- **package.xml:** Beschreibt Paketname, Versionsnummer, Autor, Abhängigkeiten und andere Informationen. Es ist ein unverzichtbarer Teil des Pakets.
- **src/:** enthält den Quellcode von ROS, einschließlich C++-Quellcode und (.cpp) und Python-Modul (.py)

- **include/:** enthält die Header-Dateien für den C++-Quellcode
- **scripts/:** enthält ausführbare Skripte, z. B. Shell-Skripte (.sh), Python-Skripte (.py)
- **msg/:** enthält benutzerdefinierte formatierte Nachrichten (.msg)
- **srv/:** enthält einen benutzerdefinierten formatierten Dienst (.srv)
- **models/:** enthält 3D-Modelle von Robotern oder Simulationsszenen (.sda, .stl, .dae, usw.)
- **urdf/:** enthält die Modellbeschreibung des Roboters (.urdf oder .xacro)
- **launch/:** enthält die Startdatei (.launch oder .xml)

Normalerweise sind ROS-Dateien in der obigen Form organisiert, dies ist eine konventionell akzeptierte Namenskonvention und es wird empfohlen, sie zu befolgen. Von den obigen Pfaden sind nur CMakeLists.txt und package.xml erforderlich, der Rest der Pfade wird dadurch bestimmt, ob das Paket sie benötigt oder nicht.

3.3.2 Package-Erstellung

Um ein Paket zu erstellen, muss über den Befehl catkin_create_pkg unter catkin_ws/src die folgenden Schritte ausgeführt werden.

catkin_create_pkg ist ein Befehl, der automatisch ein Funktionspaket erstellt, beispielsweise (catkin_create_pkg [Funktionspaketname] [abhängiges Funktionspaket 1] [abhängiges Funktionspaket 2]). Dies ist die Dateierstellungsmethode, die wir am häufigsten für das ROS verwenden, und sie ermöglicht es uns, ein leeres Projektpaket auf einmal zu erstellen, um die nächsten Schritte beim Schreiben unseres Roboterprogramms zu erleichtern.

```
1.    $ catkin_create_pkg package depends
```

Beispielsweise erstellen wir ein neues Paket namens test_pkg, und fügen die üblichen Abhängigkeiten für roscpp, rospy und std_msgs hinzu.

```
1.    $ catkin_create_pkg test_pkg roscpp rospy std_msgs
```

Dadurch wird ein neues test_pkg-Paket unter dem aktuellen Pfad erstellt, das Folgendes enthält:

```
1.    ├── CMakeLists.txt
```

```
2.    ├── include
3.    │   └── test_pkg
4.    ├── package.xml
5.    └── src
```

catkin_create_pkg hilft bei der Initialisierung des Pakets, indem es CMakeLists.txt und package.xml auffüllt und die Abhängigkeiten in beide Dateien einträgt.

3.4 Verwalten von CMakeLists.txt

3.4.1 CMakeLists.txt Funktion

CMakeLists.txt war ursprünglich eine Regeldatei für das CMake-Kompilierungssystem, und das Catkin-Kompilierungssystem folgt im Wesentlichen dem CMake-Kompilierungsstil, wobei ein paar Makrodefinitionen für ROS-Projekte hinzugefügt wurden. Vom Schreibstil her ist die CMakeLists.txt von catkin also im Grunde die gleiche wie die von CMake.

Diese Datei gibt direkt an, von welchen Paketen das Paket abhängt, welche Ziele kompiliert werden sollen, wie kompiliert werden soll und so weiter. Die CMakeLists.txt ist also sehr wichtig, sie legt die Regeln für den Weg von den Quell- zu den Zieldateien fest. Das Catkin-Build-System findet bei der Arbeit zunächst die CMakeLists.txt unter jedem Paket und kompiliert dann den Build nach den Regeln.

3.4.2 CMakeLists.txt Schreibverfahren

Die grundlegende Grammatik von CMakeLists.txt folgt immer noch der von CMake, und Catkin hat sie um eine Handvoll Makros erweitert, mit der folgenden Gesamtstruktur.

```
1.    cmake_minimum_required() #CMake's Versionsnummer
2.    project() # Projektname
3.    find_package() #Findet andere CMake/Catkin-Pakete, die zum Kompilieren ben
      ötigt werden
4.    catkin_python_setup() #catkin's neues Makro, das die Unterstützung für das Py
      thon-Modul von catkin einschaltet
```

```
5.  add_message_files() #catkin's neues Makro zum Hinzufügen von benutzerdefin
    ierten Message/Service/Action-Dateien
6.  add_service_files()
7.  add_action_files()
8.  generate_message() #catkin neues Makro zur Erzeugung verschiedener Sprach
    versionen der msg/srv/action-Schnittstelle
9.  catkin_package() #catkin fügt ein neues Makro hinzu, um die cmake-Konfigura
    tion des aktuellen Pakets für andere Pakete, die davon abhängen, zu generiere
    n
10. add_library() # Bibliotheken erzeugen
11. add_executable() #Erzeugen von ausführbaren Binärdateien
12. add_dependencies() #Zieldateien definieren, die von anderen Zieldateien abhä
    ngen, um sicherzustellen, dass andere Ziele gebaut wurden
13. target_link_libraries() #link
14. catkin_add_gtest() #catkin fügt neue Makros zum Erzeugen von Tests hinzu
15. install() #Installation auf lokalem Rechner
```

3.5 Package.xml

Package.xml ist ebenfalls eine für Catkin-Pakete erforderliche Datei. Es ist die Beschreibungsdatei für das Paket, in früheren Versionen von ROS (rosbuild build system) heißt diese Datei manifest.xml und wird verwendet, um die grundlegenden Informationen über das Paket zu beschreiben.

3.5.1 Die Funktion von package.xml

Die Datei pacakge.xml enthält den Paketnamen, die Versionsnummer, den Inhalt, den Betreuer, die Softwarelizenz, das Kompilierwerkzeug, die Kompilierabhängigkeiten, die Ausführungsabhängigkeiten und weitere Informationen.

Tatsächlich können rospack search, rosdep und andere Befehle Informationen wie Paketabhängigkeiten schnell finden und analysieren, da sie die Datei package.xml in jedem Paket direkt lesen können. Dies bietet dem Benutzer einen schnellen Zugriff auf ein Paket.

3.5.2 Package.xml Schreibverfahren

package.xml folgt der Art und Weise, wie XML-Tag-Text geschrieben wird, und aufgrund von Versionsänderungen existieren nun zwei Formate nebeneinander (format1 und format2), wobei der Unterschied nicht signifikant ist. Wir verwenden nun generell das neueste Verfahren wie folgt:

1. <Package> Wurzel-Tag-Datei
2. <Name> Name des Pakets
3. <Version> Versionsnummer
4. <Deskription> Inhalt Beschreibung
5. <Maintainer> Verwalter
6. <Lizenz> Software-Lizenz
7. <buildtool_depend> Build-Tool kompilieren, normalerweise catkin
8. <depend> Legt die Abhängigkeiten bezüglich der Kompilierung, des Exports und der Ausführung fest.
9. <build_depend> Abhängigkeiten kompilieren
10. <build_export_depend> Abhängigkeiten exportieren
11. <exec_depend> Abhängigkeiten ausführen
12. <test_depend> Abhängigkeiten von Testfällen
13. <doc_depend> Abhängigkeiten der Dokumentation

3.6 ROS-Kommunikationsarchitektur

Die Kommunikationsarchitektur von ROS ist die Seele des ROS und der Schlüssel zum ordnungsgemäßen Betrieb des gesamten ROS. Die ROS-Kommunikationsarchitektur umfasst die Verarbeitung verschiedener Daten, den Betrieb von Prozessen, die Weitergabe von Nachrichten und so weiter. In diesem Kapitel werden die grundlegenden Kommunikationsmethoden und die zugehörigen Konzepte der Kommunikationsarchitektur vorgestellt. Er führt die kleinste Prozesseinheit, den Knoten, und den Knotenmanager, den Knotenmaster, ein, der versteht, dass die Prozesse in ROS aus vielen Node bestehen und dass der Knotenmaster diese Node verwaltet.

3.6.1 Node & Master

a) Node

In der ROS-Welt ist der Node die kleinste Prozesseinheit. Ein Paket hat mehrere ausführbare Dateien, die, wenn sie ausgeführt werden, zu einem Prozess zusammengefasst werden, und dieser Prozess wird in ROS ein Node genannt.

Aus programmiertechnischer Sicht ist ein Node eine ausführbare Datei (normalerweise eine in C++ kompilierte ausführbare Datei oder ein Python-Skript), die ausgeführt und in den Speicher geladen wird; aus funktionaler Sicht ist ein Node normalerweise für eine einzelne Funktion des Roboters verantwortlich. Da die Funktionsmodule des Roboters sehr komplex sind, konzentrieren wir nicht alle Funktionen auf einen Node, sondern verwenden einen verteilten Ansatz, um die Eier in verschiedene Körbe zu legen.

b) Master

Aufgrund der großen Anzahl von Komponenten und Funktionen eines Roboters gibt es in der Praxis oft viele Knoten, die für die Erfassung der Welt, die Steuerung der Bewegung, das Treffen von Entscheidungen und die Berechnung zuständig sind. Der Master "verkabelt" die Node auch, damit sie miteinander kommunizieren können, bevor sie Peer-to-Peer kommunizieren können. Wenn das ROS-Programm startet, wird als erstes der Master gestartet und der Knotenmanager kümmert sich um das sequenzielle Starten der Node.

3.6.2 Start Master und Node

Wenn wir ROS starten wollen, müssen wir zunächst den Befehl eingeben:

```
1.    $ roscore
```

An diesem Punkt startet der ROS-Master, zusammen mit rosout und dem Parameter Server, wobei rosout ein Node ist, der für die Protokollausgabe zuständig ist. Seine Aufgabe ist es, den Benutzer über den aktuellen Systemstatus zu informieren, inklusive der Ausgabe des Systemfehlers, der Warnung usw., dies wird in der Protokolldatei protokolliert. Der Parameter Server ist der Datenserver für die Einstellungen, er ist kein Knoten, sondern ein Server, der die Konfiguration der Parameter speichert. Jedes Mal,

wenn wir den ROS-Node starten, müssen wir den Master aktivieren, damit der Node starten und sich registrieren kann.

Nach dem Master beginnt der Node-Manager, spezifische Nodes in Abstimmung mit der Anordnung des Systems zu starten. Ein Node ist ein Prozess, nur dass er in ROS einen speziellen Namen erhält – Node.

Genau wie im Abschnitt 2.1.2 über ROS allgemeine Grundbefehle wird hier ein Knoten mit dem Befehl rosrun gestartet.

```
1.    $ rosrun pkg_name node_name
```

Normalerweise führen wir ROS aus und starten es in dieser Reihenfolge, manchmal gibt es zu viele Knoten und wir entscheiden uns, das Programm mit dem Befehl Roslaunch zu

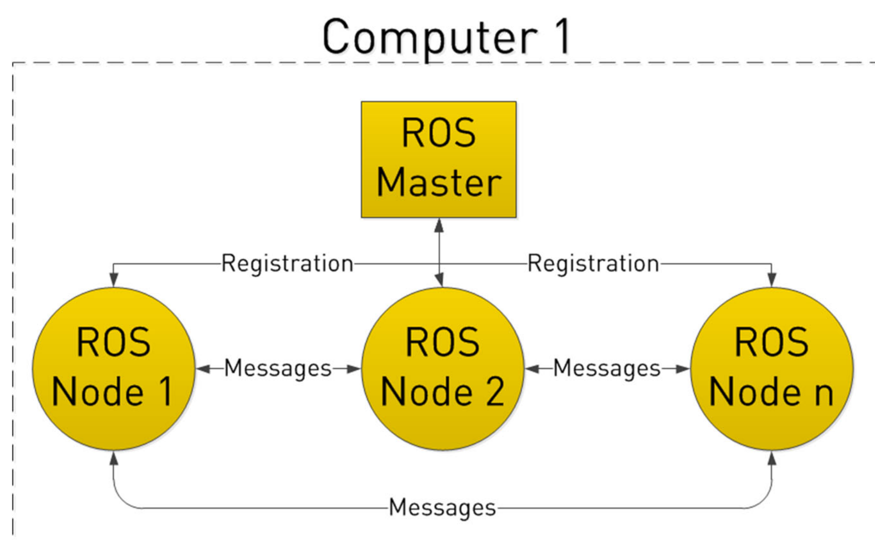


Abbildung 3-3 Diagramm der Knotenbeziehungen [21]

starten. Die Beziehung zwischen Master, Knoten und zwischen den Knoten ist in der obigen **Abbildung 3-3** dargestellt.

3.7 Launch Datei

Robotik ist ein Systemprojekt. Normalerweise läuft ein Roboter mit vielen geöffneten Knoten, allerdings brauchen wir nicht jeden Knoten nacheinander zu rosrun, ROS stellt uns einen Befehl zur Verfügung, um Master und mehrere Knoten auf einmal zu starten. Der Befehl dazu lautet:

```
1.    $ roslaunch pkg_name file_name.launch
```

Der Befehl roslaunch erkennt zunächst automatisch, ob roscore auf dem System läuft. Wenn die Master nicht laufen kann, dann startet roslaunch zuerst die Master und führt sie dann nach den Regeln von launch aus, die bereits in der launch-Datei konfiguriert sind. Roslaunch ist also wie ein Startup-Tool, das mehrere Nodes auf einmal gemäß unserer vorkonfigurierten Konfiguration starten kann, wodurch die mühsame Eingabe von Befehlen nacheinander im Terminal entfällt.

Die Launch-Datei folgt ebenfalls der XML-Formatspezifikation und ist ein getaggter Text, der so formatiert ist, dass er die folgenden Tags enthält:

```
1.  <launch> <! --root Tag -->
2.  <Knoten> <! --zu startender Knoten und seine Parameter -->
3.  <include> <! -Andere Starts einbeziehen -->
4.  <Maschine> <! -Angabe der Maschine, die ausgeführt werden soll -->
5.  <env-loader> <! -- Umgebungsvariablen setzen -->
6.  <param> <! -- Parameter an den Parameter-Server definieren -->
7.  <rosparam> <! --launch yaml-Datei Argumente für den Parameterserver -->
8.  <arg> <! -- Variablen definieren -->
9.  <remap> <! -- Parameter-Mapping einstellen -->
10. <group> <! --> -- Namespace setzen -->
11. </launch> <! --root Tag -->
```

3.8 Topic

Unter den Kommunikationsmethoden in ROS spielt das Topic eine häufige Rolle. Für Echtzeit-, periodische Nachrichten ist ein Topic die beste Wahl. Topic ist eine Punkt-zu-Punkt-Einweg-Kommunikationsmethode, wobei sich der "Punkt" auf den Node bezieht, d. h. die Nodes können sich gegenseitig Informationen über die Topic-Methode zukommen lassen. Zunächst müssen sich sowohl der Publisher-Node als auch der Subscriber-Node beim Node-Manager registrieren, dann veröffentlicht der Publisher das Topic, und der Subscriber abonniert das Topic unter dem Befehl des Masters, wodurch die Kommunikation zwischen den Sub-Pubs hergestellt wird. Bitte beachten, dass der gesamte Prozess einseitig ist. ROS ist eine verteilte Architektur, bei der ein Topic von

mehreren Nodes gleichzeitig veröffentlicht werden kann und von mehreren Nodes gleichzeitig empfangen werden kann.

3.9 Message

Wenn ein Topic strenge Formatierungsanforderungen hat, muss es einem bestimmten Format folgen, z. B. Farbe, Länge, Größe, Position usw. Dieses Datenformat wird als Message bezeichnet. Message ist definitionsgemäß dem Datentyp des Topic-Inhalts nachempfunden, auch Formatstandard des Topics genannt.

Gängige Nachrichtentypen beinhalten *std_msgs, sensor_msgs, nav_msgs, geometry_msgs*, etc.

a) **Vector3.msg**

```
1. # Dateispeicherort: geometry_msgs/Vector3.msg

2. float64 x
3. float64 y
4. float64 z
```

b) **Accel.msg**

```
1. # Definieren Sie die Begriffe der Beschleunigung, einschließlich der linearen un
   d der Winkelbeschleunigung
2. #Dateispeicherort: geometry_msgs/Accel.msg
3. Vektor3 linear
4. Vektor3 winkelig
```

3.10 Service

Im vorigen Abschnitt haben wir die Topic-Kommunikation in ROS vorgestellt, und wir wissen, dass Topic eine einseitige asynchrone Kommunikationsmethode in ROS ist. Es gibt jedoch Zeiten, in denen die Einweg-Kommunikation die Kommunikationsanforderungen nicht erfüllt, z. B. wenn einige Nodes nur vorübergehend einige Daten benötigen, aber nicht regelmäßig oder wenn die Topic-

Kommunikationsmethode eine Menge unnötiger Systemressourcen verbraucht, was zu einem ineffizienten Betrieb und hohen Stromverbrauch des Systems führt.

In diesem Fall wird ein alternatives Anfrage-Abfrage-Kommunikationsmodell benötigt. In diesem Abschnitt stellen wir Service vor, eine weitere Art der Kommunikation in der ROS-Kommunikation.

Um das obige Problem zu lösen, macht der Service-Ansatz einen Unterschied im Kommunikationsmodell zum Topic. Die Service-Kommunikation ist bidirektional, sie sendet nicht nur Nachrichten, sondern hat auch Rückmeldungen. Ein Dienst besteht also aus zwei Teilen, zum einen aus der anfragenden Partei (Client) und zum anderen aus der antwortenden Partei/Dienstanbieter (Server). Zu diesem Zeitpunkt sendet die anfragende Partei (Client) eine Anfrage, um auf die Verarbeitung des Servers zu warten, um eine Antwort zurückzusenden, so dass durch einen ähnlichen "Anfrage-Antwort"-Mechanismus die gesamte Dienstkommunikation abgeschlossen wird.

Ein Diagramm dieser Kommunikationsart sieht folgendermaßen aus.

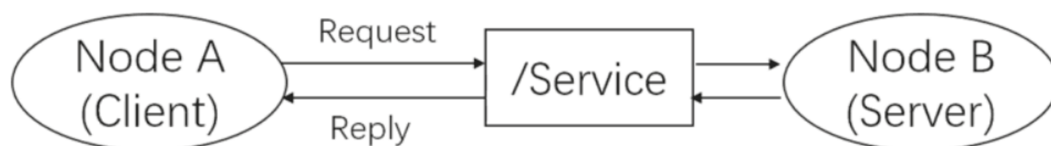


Abbildung 3-4 Kommunikationsart [37]

Service ist eine synchrone Kommunikationsmethode. Wenn Node A eine Anfrage übersendet, so wird gewartet, bis Node B die Bearbeitung der Anfrage fertiggestellt hat und eine Antwort übersendet, das heißt Node A wird weiterhin ausgeführt. Ein solches Kommunikationsmodell hat keine häufige Nachrichtenübergabe, keine Konflikte und keine hohe Systemressourcennutzung und nimmt nur Anfragen an, bevor es Dienste ausführt, was einfach und effizient ist.

4 Auslegung eines Robotermodells Roboter-Wagen

Im Robot Operating System (ROS) ist es notwendig, Roboterbewegungen zu simulieren, daher muss zunächst ein neuer Roboter erstellt werden.

Die Schritte zur Erstellung eines grundlegenden Robotermodells sind wie folgt.

- Erstellen des Hardware-Beschreibungspakets
- Erstellen der urdf-Datei
- Erstellen der Launch Command-Datei
- Inbetriebnahme Durchführung
- Tastatur-Steuerung
- Node-Relationship-Diagramm
- Demonstrieren der Wirkung

4.1 Erstellen eines Hardware-Beschreibungspakets im Arbeitsbereich

Das Hardwarebeschreibungspaket wird im Arbeitsbereich angelegt und die erforderlichen Dateiverzeichnisse sowie die Dateipakete werden über die Systembefehle von Linux hinzugefügt.

```
1. mkdir ~/robot_model_ws/src -p
2. cd ~/ robot_model_ws/src
3. catkin_create_pkg robot_model std_msg roscpp rospy
```

Bei der Erstellung des ROS-Projektpakets ist es wichtig zu beachten, dass die vom Programm benötigten Abhängigkeiten im Paket hinzugefügt werden müssen. Diese Abhängigkeiten werden automatisch durch das ROS hinzugefügt und ermöglichen es, dass das geschriebene Programm vom System gelesen werden kann. Dies ist eine der Eigenschaften der ROS-Plattform.

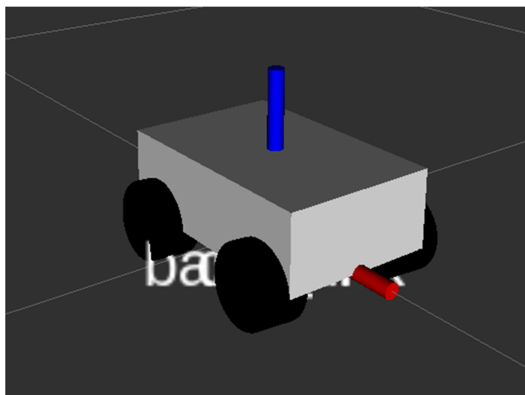
Wenn ein Projektpaket erstellt wird, erzeugt das System automatisch eine Datei namens package.xml in seinem Stammverzeichnis. Diese Datei definiert einige Eigenschaften des

Pakets, beispielsweise den Namen, die Paketversionsnummer, den Autor, den Betreuer und die Abhängigkeiten von den Paketen.

4.2 Erstellen die Beschreibung des Modells Robot.urdf

Der vollständige Name von URDF ist Unified Robot Description Format. Diese Datei ist eine Beschreibung des Roboters, eine Beschreibung der Größe und Form des Roboterteils in den Simulationsbedingungen und die Kombination der Daten für jedes Teil. Die Datei verwendet die XML-Sprache, um die entsprechende Beschreibung zu vervollständigen.

Der folgende Code stellt einen Roboter mit einem Körper und vier Rädern dar. Da es sich um einen simulierten Roboter handelt, werden grundlegende Elemente verwendet, wie eine rechteckige Karosserie anstelle eines Automodells und ein Zylinder anstelle von Rädern, mit den folgenden Abmessungen:



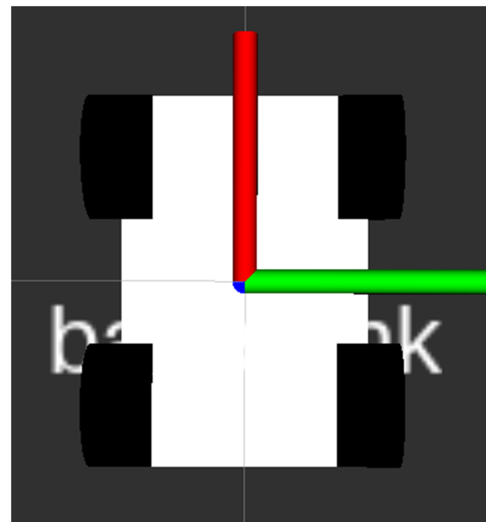


Abbildung 4-1 Wagenkörper

Ein rechteckiger Körper mit einer Länge, Breite und Höhe von 0,3, 0,2 bzw. 0,1 Metereinheiten wird als Hauptteil des Wagens angesetzt und die Räder sind als vier Zylinder mit einem Durchmesser von 0,05 Meter und einer Höhe von 0,05 Meter ausgelegt. Dies ist in **Abbildung 4.1** dargestellt.

4.2.1 URDF-Beschreibungsdatei für Wagenkörper

Die URDF-Beschreibungsdatei des Wagenkörpers wird in Form einer XML-Datei ausgedrückt, im Ubuntu-System können die URDF-Datei direkt bearbeitet werden, hier wird die XML-Version 1,0 verwendet, das heißt es muss in der ersten Zeile der Datei Platz eingefügt werden.

4.2.2 Vorbereitung der Modelldaten

Dieses Roboter-Chassis-Modell hat 5 Modul-Links, die das Karosserieteil und 4 Räder enthalten, zusätzlich werden 4 Joint definiert, um die Verbindungen zwischen den Gelenken zu beschreiben. Nach der anfänglichen Spracheinstellung wird der Name jedes Moduls des Roboters festgelegt, damit es später einfach über Gelenke verbunden werden kann.

Anschließend wird das Basismodell des gesamten Roboters erstellt. Dabei wird mit der Gesamtstruktur des Roboters angefangen, ohne zu viele Details zu berücksichtigen, die durch die folgende URDF dargestellt werden kann:

```
1.  <robot name="origins">
2.    <link name="base_link" />
3.    <link name="wheel_1" />
4.    <link name="wheel_2" />
5.    <link name="wheel_3" />
6.    <link name="wheel_4" />
7.
8.    <joint name="joint1" type="continuous">
9.      <parent link="base_link"/>
10.     <child link="wheel_1"/>
11.   </joint>
12.
13.   <joint name="joint2" type="continuous">
14.     <parent link="base_link"/>
15.     <child link="wheel_2"/>
16.   </joint>
```

```
17.
18.     <joint name="joint3" type="continuous">
19.       <parent link="base_link"/>
20.       <child link="wheel_3"/>
21.     </joint>
22.
23.   <joint name="joint4" type="continuous">
24.       <parent link="base_link"/>
25.       <child link="wheel_4"/>
26.     </joint>
```

Zusätzlich zum Basismodell werden dem Roboter Größenangaben hinzugefügt. Da sich das Bezugssystem für jedes Glied, wie auch die Gelenke, am unteren Ende des Gliedes befindet, reicht es aus, bei der Darstellung der Größe die relative Position zu den angeschlossenen Gelenken zu beschreiben. Das Feld <origin> in der URDF wird verwendet, um diese relative Beziehung darzustellen.

Nach der Bestimmung des Hauptteils und des Verbindungsteils müssen zunächst die Größe und Position des Hauptteils sowie weitere Eigenschaften wie Farbmaterial usw. festgelegt werden. Die Länge, Breite und Höhe des rechteckigen Körperteils des Wagens wird mit dem Code *<box size = "0.3 0.2 0.1"/>* auf 0,3, 0,2 bzw. 0,1 festgelegt.

Nachdem die geometrischen Eigenschaften des Körperteils ermittelt wurden, wird nun seine räumliche Position bestimmt, indem der Abschnitt *<origin XYZ>* des Codes verwendet wird. Aufgrund der Höhe des Rades muss die z-Achsenhöhe des Karosserieteils um einen Radius erhöht werden. Der Code lautet wie folgt:

```
1.  <link name="base_link">
2.          <visual>
3.                  <geometry>
4.                          <box size="0.3 0.2 0.1"/>
5.                  </geometry>
6.                  <origin rpy="0 0 0" xyz="0 0 0.05"/>
7.                  <material name="white">
8.                          <color rgba="1 1 1 1"/>
```

```
9.                  </material>
10.         </visual>
11.     </link>
```

Als nächstes müssen die 4 Räder codiert werden und die Räder durch Verschieben und Drehen der Achsen mit dem Körper zusammengefügt werden. Zuerst wird ein Link zum linken Rad und ein Joint zwischen dem linken Rad und dem Gehäuse hinzugefügt. Das *r* im Code *rpy* bedeutet, dass er in einen Winkel umgerechnet wird. Da das linke Rad vertikal platziert ist, müssen wir seine Winkelumwandlung auf 90 setzen.

```
1.  <link name="wheel_1">
2.          <visual>
3.                  <geometry>
4.                          <cylinder length="0.05" radius="0.05"/>
5.                  </geometry>
6.              <origin rpy="1.5 0 0" xyz="0.1 0.1 0"/>
7.              <material name="black">
8.                      <color rgba="0 0 0 1"/>
9.              </material>
10.         </visual>
```

Ähnliches gilt für die anderen 3 Räder, die um den Körper gedreht und geschwenkt werden müssen und schließlich durch den Joint miteinander verbunden werden. Hier ist ein Beispiel mit dem linken Vorderrad mit folgendem Code:

```
1.  <joint name="base_to_wheel1" type="fixed">
2.          <parent link="base_link"/>
3.          <child link="wheel_1"/>
4.          <origin xyz="0 0 0"/>
5.   </joint>
```

Hier kann zur realistischen Demonstration der Code *<axis xyz="0 1 0"*> hinzugefügt werden, um anzuzeigen, dass sich das Rad um die Y-Achse dreht. Der endgültige Code sieht also wie folgt aus:

```
1.  <joint name="base_to_wheel1" type="fixed">
2.          <parent link="base_link"/>
3.          <child link="wheel_1"/>
```

```
4.        <origin xyz="0 0 0"/>
5.        <axis xyz="0 1 0">
6.    </joint>
```

4.3 Launch-Datei erstellen

Die Antwort auf die Frage, warum eine Launch Datei benötigt wird, liegt darin, dass es viele Knoten gibt, wenn ein Roboter fertiggestellt werden soll. Um eine Funktion zu implementieren, müssen wir im Terminal einen Knoten nach dem anderen starten, und das ist sehr ineffizient. Daher wird eine Launch Datei vorgeschlagen, die mehr als einen Knoten auf einmal starten kann. Die gleiche Launch-Datei wird auch in der gleichen XML-Sprache erstellt.

Erstellen eines Launch-Ordners im Ordner *robot_model_ws\src\robot_model*, Erstellen der Beschreibungsdatei *robot_model.launch* für dem Wagenkörper (entspricht dem Öffnen von Robot.urdf).

Das Format der Launch-Datei ist wie folgt:

```
1.    <launch>
2.        <node>...
3.    </launch>
```

4.3.1 Der Node-Abschnitt der Launch-Datei

Node ist der zu beginnende ROS-Knoten und enthält die folgenden Parameter:

1. pkg="mypackage”: Funktionspaket für den Knoten
2. type="nodetype”: der Typ des Knotens
3. name="nodename”: der Name des Knotens, der jeden Namen überschreibt, der dem Knoten durch den Aufruf von ros:init gegeben wurde
4. args="arg1 arg2 arg3" (optional): die Argumente, die an den Knoten übergeben werden sollen
5. respawn="true" (optional): startet den Knoten automatisch neu, wenn er stehen bleibt
6. ns="foo" (optional): Startet den Knoten im Namensraum "foo".
7. output="log | screen" (optional): wenn screen ausgewählt ist, werden die Meldungen *stdout* (Standardausgabe) und *stderr* (Standardfehler) des Knotens im T

> erminalfenster angezeigt; wenn log ausgewählt ist, wird *stdout* und *stderr* eine Protokolldatei gesendet und *stderr* wird ebenfalls im Terminalfenster angezeig t

Hier ist ein Beispiel dafür, wie ein Wagen zu einem der Start-Knoten von Rviz gestartet wird:

```
1.  <?xml version="1.0"?>
2.  <launch>
3.      <arg name="model" default="$(find robot_model) /urdf/robot.urdf"/>
4.      <arg name="gui" default="false" />
5.      <param name="robot_description" textfile="$(arg model)" />
6.      <param name="use_gui" value="$(arg gui)"/>
7.  </launch>
```

Diese Datei hilft, Rviz zu starten, um den Roboter zu kontrollieren. Es gibt drei Knoten, einer ist Rviz, die anderen beiden joint_state_publisher und robot_state_publisher sind erforderliche Knoten. Der erste Eingabeparameter Model ist der Weg zu der zu startenden urdf-Datei. Der zweite Eingabeparameter <gui> gibt an, ob das Fenster des Bedienfelds für die Gelenkdrehung aktiviert werden soll.

4.3.2 Unterschied und Zusammenhang zwischen joint_state_publisher und robot_state_publisher

Wenn die Dateien in der Launch-Datei fertiggestellt sind, muss als nächstes überlegt werden, wie der Roboter in Bewegung gesetzt werden kann. Zu diesem Zeitpunkt muss die Bewegung des Roboters über joint_state_publisher und robot_state_publisher erfolgen. Dies sind zwei Publisher, die mit dem ROS bereitgestellt werden. Außerdem helfen sie dabei, Daten zu senden, die die Bewegung des Roboters ermöglichen. Hier sind die Aufrufe zu ihnen in der Launch-Datei:

```
1.  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_s
    tate_publisher" />
2.  <node name="robot_state_publisher" pkg="robot_state_publisher" type="rob
    ot_state_publisher" />
3.  <node name="robot_base_state_publisher" pkg="robot_model" type="robot_
    state_publisher" />
```

Der Node joint_state_publisher erhält Informationen über alle nicht fixierten Joints des Roboters durch Ablesen von */robot_description* auf dem Parameterserver und holt sich den Gelenkzustand aus dem Topic */move_group/fake_controller_joint_states*. Seine Hauptfunktion ist für die Ausgabe von gemeinsamen Nachrichten *(sensor_msgs/JointState)* und die Veröffentlichung von */joint_state*-Themen an die Nodes */move_group* und */robot_state_publisher* zuständig.

Der Node robot_state_publisher ist der Node, der die von joint_state_publisher veröffentlichten Topic-Nachrichten empfängt und die Ergebnisse über **tf** an den Node */move_group* veröffentlicht.

4.3.3 Ausführen der Rviz-Visualisierungsoberfläche

Nachdem die benötigten Einstellungen und die Datei zur Veröffentlichung der Nachricht an Launch geschrieben wurden, müssen der Simulator verwendet werden, um unser Programm greifbar zu machen.

An diesem Punkt müssen die Node-Datei von Rviz gestartet werden, der Visualisierungssoftware, die mit dem ROS mitgeliefert wird, um das programmierte Modell zur Fehlersuche lediglich zu simulieren. Es wird aufgerufen, indem ein Knoten namens Rviz gestartet wird und die erforderlichen Modelldaten über diesen Node geladen werden. Der Code ist unten dargestellt:

```
1.    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find robot_model)/rviz/
      display.rviz" />
```

Abschließend wird dieser Satz in die zuvor geschriebene Launch-Datei eingefügt und mit dem Befehl kann das Programm Rviz gestartet und das zuvor programmierte Robotermodell darin angezeigt werden. Damit ist die Programmierung der Launch-Datei für unseren Roboter abgeschlossen.

4.4 Inbetriebnahme Durchführung

Die zuvor eingerichtete Launch-Datei wird über den Befehl *roslaunch robot_model robot_model.launch* ausgeführt. Bevor das System gebootet wird, müssen die von uns geschriebenen Dateien von ROS kompiliert werden, also wird im Verzeichnis Files ein

Terminal geöffnet und der Befehl catkin_make ausgeführt. Der Befehl zum Starten der Launch-Datei lautet wie folgt:

1. cd robot_model_ws
2. catkin_make
3. source devel/setup.bash
4. roslaunch robot_model robot_model.launch

An diesem Punkt wird Rviz direkt gestartet, aber die Rviz-Datei muss so eingerichtet werden, dass die geschriebene Datei beim ersten Start geladen werden kann. Wie im Bild gezeigt, muss das Topic und base_link in der oberen linken Ecke geändert werden, dann wird robot_model über ***add*** hinzugefügt, so dass unser Modell in Rviz wie in **Abbildung 4-2** angezeigt werden kann.

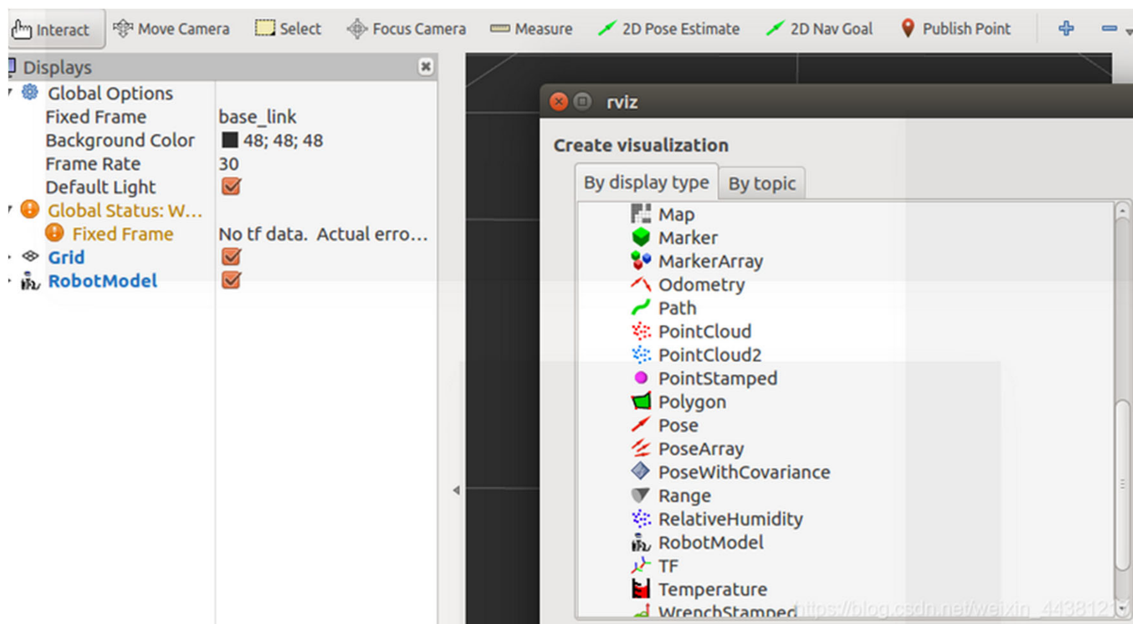


Abbildung 4-2 Durchführung in Rviz

4.5 Tastatur-Steuerung

An diesem Punkt wird das Roboterwagenmodell angezeigt, das in Rviz mit der URDF-Datei geschrieben wurde, und nun können Befehle eingegeben werden, um es in Bewegung zu setzen. Da der ROS-Code hochgradig portabel ist, kann der Code für die Steuerung mit der Tastatur tatsächlich direkt von anderen Bot-Paketen portiert werden,

in diesem Fall ist die Referenz der Turtlesim, wo der Knoten *turtlesim_node* und der Knoten *turtle_teleop_key* aufgerufen werden können.

4.5.1 Publisher

Das Programm veröffentlicht und empfängt Daten über die ROS-Plattform, während jeder Roboter läuft, und für unseren Roboterwagen verweist der Publisher des Programms auf den Turtlesim_node und turtle_teleop_key. Diese beiden Knoten gehören zum Basispaket von ROS. Die turtlesim_node enthält Folgendes,

Node [/turtle_node]

Publications:

* /turtle1/color_sensor [turtlesim/Color]

* /rosout [rosgraph_msgs/Log]

* /turtle1/pose [turtlesim/Pose]

Subscriptions:

* Für Subscriber, den Empfänger des Roboterwagens, muss auf den Knoten */turtle1/cmd_vel* von Turtlesim verwiesen werden, aber da die Daten des Wagens sich vom Turtle unterscheiden, wird eine Datei erstellt, die den Verweis auf den *Turtlesim-Knoten* ausdrückt. *[geometry_msgs/Twist]*

Wie hier dargestellt, wird dieser Knoten ein Topic mit */turtle1/pose* veröffentlichen.

Um den bereits geschriebenen Roboter zu bewegen wie Turtle im ROS-Tutorial, ist es ausreichend, wenn sich der Wagen auf dieses Topic einstellt.

Der Publisher kann die Standortinformationen der Wagen den Empfängern von Turtlesim mitteilen, indem er dieses Topic abonniert.

4.5.2 Subscriber

Für Subscriber, den Empfänger des Roboterwagens, muss auf den Knoten */turtle1/cmd_vel* von Turtlesim verwiesen werden, aber da die Daten des Wagens sich vom Turtle unterscheiden, wird eine Datei erstellt, die den Verweis auf den Turtlesim-

Knoten ausdrückt. Im Roboterverzeichnis wird der Ordner */scripts* angelegt und die Datei *robot_base_control.py* erstellt, um die Fahrgestellsteuerung des Wagens zu gewährleisten:

```
1.  mkdir ~/robot_model_ws/ scripts -p
2.  cd ~/robot_model_ws/ scripts
3.  touch robot_base_control.py
```

4.5.3 Beschreibung robot_base_control.py

Der erste Schritt besteht darin, die Abhängigkeiten einzurichten, die das Programm möglicherweise verwendet, hauptsächlich *rospy, tf, math,* usw.

```
1.  import rospy
2.  from geometry_msgs.msg import Twist, Pose2D, TransformStamped
3.  import tf
4.  import math
```

Im zweiten Teil werden mit *def __init__(self)* die Anfangswerte der Variablen definiert, die für den Robot_state_publisher benötigt werden, um die Anfangswerte und den Anfangszustand zu bestimmen.

```
1.  class RobotStatePublisher:
2.      def __init__(self):
3.          self.robot_pose_ = Pose2D()
4.          self.robot_pose_.x = 0.0
5.          self.robot_pose_.y = 0.0
6.          self.robot_pose_.theta = 0.0
7.  ......
```

Im dritten Teil werden dann zwei Callback-Funktionen aufgerufen, die in der Lage sind, die eingehenden Informationen zu lesen und zu erkennen und die Anfangswerte der Variablen oben entsprechend der unterschiedlichen Informationen zu aktualisieren. Ziel ist es, durch diesen Prozess die Informationen des Roboters in der ROS-Umgebung auszulesen und Befehle zu erteilen, damit er selbst erkennen kann, ob er die Bewegung erhält. Ein Abschnitt des Codes ist unten dargestellt:

```
1.  def cmd_callback(self, cmd):
2.      if(cmd.angular.z == 0):
```

```
3.                  delta_dis = cmd.linear.x * self.cmd_delta_time_
4.                  ......
5.          else:
6.                  R = cmd.linear.x / cmd.angular.z
7.                  delta_theta = cmd.angular.z * self.cmd_delta_time_
8.                  delta_x = R * (1 - math.cos(delta_theta))
9.                  ......
10.         if(self.robot_pose_.theta > math.pi*2):
11.                 self.robot_pose_.theta -= math.pi*2
12.                 elif(self.robot_pose_.theta < 0):
13.         self.robot_pose_.theta += math.pi*2
```

Schließlich gibt es noch den Standard-Einstiegscode von Python, der den */cmd*-Teil des ROS-Pakets aufrufen kann. Mit diesen wenigen Schritten ist der Teil der Tastatursteuerung des Wagens abgeschlossen. Während dieser Zeit wird die Tastatursteuerung über das Topic turtle_teleop_key realisiert. Der Aufrufcode wird unten angezeigt:

```
1.      rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/cmd_vel
```

Wenn dieser Befehl eingegeben wird, kann der Wagen mit den Tasten Auf, Ab, Links und Rechts der Tastatur gesteuert werden.

4.6 Node-Relationship-Diagramm

Das Knotenbeziehungsdiagramm für die Tastatursteuerung des Wagens ist in der Abbildung unten dargestellt. Darin ist zu erkennen, dass der Wagen den Turtle simuliert, um Befehle über */cmd_vel* zu veröffentlichen, die dann über den Knoten

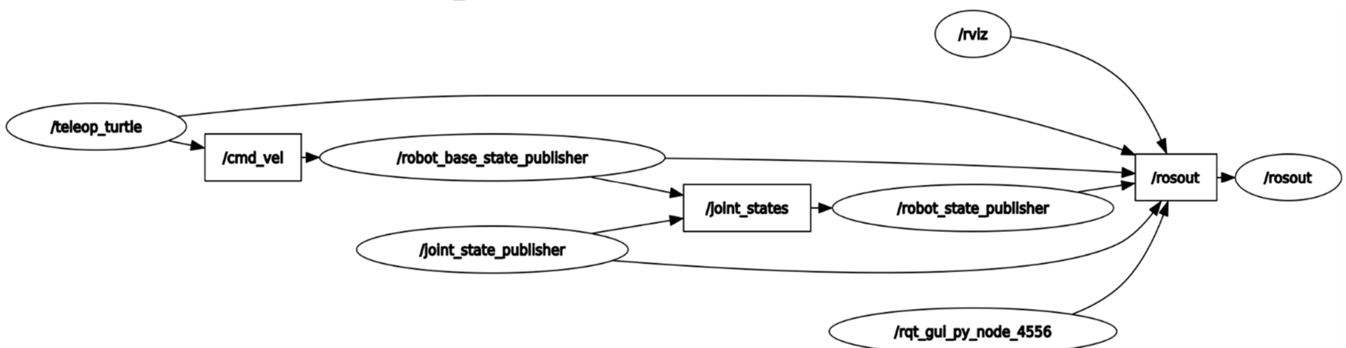


Abbildung 4-3 Node-Relationship-Diagramm

/robot_state_publisher empfangen und an den Knoten */rviz* für die Ausgabe von Rosout weitergegeben werden.

5 Auslegung des Roboterarms

Im vorherigen Abschnitt wurde beschrieben, wie eine Simulation eines Roboterwagens auf der ROS-Plattform entworfen und implementiert wird. Im Folgenden wird ein Roboterarm zu dem zuvor entworfenen Roboterwagen hinzugefügt. Ein grundlegender Roboterarm mit 3 Gelenken ist entworfen worden.

Die Schritte zur Erstellung eines grundlegenden Robotermodells sind wie folgt.

- Erstellung des Modells Roboterarm.urdf
- Modifizierung der Launch-Datei
- Automatische Steuerung mit Roboterarmwagen
- Automatische Steuerung mit Roboterarm

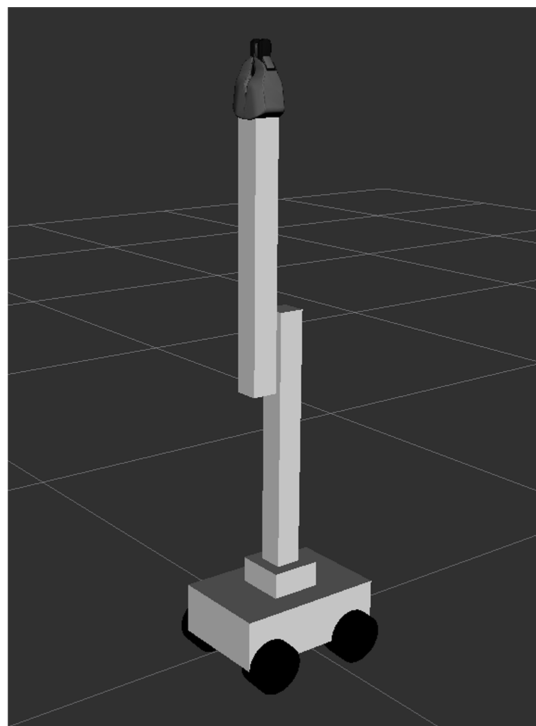


Abbildung 5-1 Entwurf des Roboterarms

5.1 Erstellung des Modells Roboterarm.urdf

Da die URDF-Datei für den Wagen bereits fertiggestellt wurde, muss nun ein Roboterarm mit Gelenken auf dem oberen Teil des Wagens gebaut werden. Auch hier wird für die Beschreibung des Roboterarmteils die XML-Sprache verwendet. Der neue benötigte Abschnitt wird nach der vorherigen Datei eingefügt.

Da es sich wiederum um ein Modelldesign handelt, werden einfache Objekte anstelle eines physischen Modells verwendet. So wird ein kleines rechteckiges Objekt verwendet, um den Drehpunkt an der Unterseite des Roboterarms darzustellen, und zwei rechteckige Objekte werden anstelle des Roboterarms eingesetzt, die mechanischen Greifer werden von SolidWorks Design erzeugt. Der endgültige Entwurf ist in **Abbildung 5.1** (s.o.) dargestellt.

5.1.1 URDF-Beschreibungsdatei für Roboterarm

Auch hier wird nach dem Modulprinzip vorgegangen, um den Code für den Roboterarm zu schreiben. Der gesamte Arm ist in 3 Teile unterteilt, die das Gehäuse und zwei Arm-Teile umfassen, da die mechanischen Greifer in SolidWorks erstellt werden muss, werden hier zuerst die restlichen Komponenten besprochen.

Auch hier wird zuerst der Hauptteil der URDF des Roboterarms geschrieben, wie unten gezeigt:

```
1.  <link name="arm_base">
2.  .....
3.  <link>
4.  <joint name="base_to_arm_base" type="continuous">
5.  ......
6.  </joint>
7.  <link name="arm_1">
8.  ......
9.  <joint name="arm_1_to_arm_base" type="revolute">
10. ......
11. </joint>
```

```
12.    <link name="arm_2">
13.    ......
14.    <joint name="arm_2_to_arm_1" type="revolute">
15.    ......
16.    </joint>
```

Nachdem das Körperteil und das Anschlussteil bestimmt wurden, ist der nächste Schritt die Bestimmung der Größe und Lage der einzelnen Unterteile. Die Länge, Breite und Höhe des rechteckigen Anschlussbereichs werden auf 0,1, 0,1 bzw. 0,1 gesetzt, und zwar mit dem Code <box="0,1 0,1 0,1"/>.

Nachdem die Art des angeschlossenen Teils bestimmt wurde, sollten auch die Informationen im angeschlossenen Teil wie folgt ausgedrückt werden:

```
1.    <joint name="base_to_arm_base" type="continuous">
2.        <parent link="base_link"/>
3.        <child link="arm_base"/>
4.        <axis xyz="0 0 1"/>
5.        <origin xyz="0 0 0"/>
6.    </joint>
```

Dann wird der erste Roboterarm mit einer Länge, Breite und Höhe von jeweils 0,05, 0,05 und 0,5 mit dem vorherigen Teil verbunden. Dieselben Überlegungen gelten dementsprechend für den zweiten Roboterarm.

5.1.2 URDF-Beschreibungsdatei für Robotergreifer

Die Teile der mechanischen Greifer sind in zwei Gruppen unterteilt, wobei jede Gruppe eine Hauptstruktur und eine Zubehörstruktur enthält. Die Hauptstruktur der URDF-Datei ist also unten dargestellt:

```
1.    <joint name="left_gripper_joint" type="revolute">
2.    ......
3.    </joint>
4.    <link name="left_gripper">
5.    ......
6.    </link>
```

```
7.    <joint name="left_tip_joint" type="fixed">
8.    ......
9.    </joint>

10.   <joint name="right_gripper_joint" type="revolute">
11.   ......
12.   </joint>
13.   <link name="right_gripper">
14.   ......
15.   </link>
16.   <joint name="right_tip_joint" type="fixed">
17.   .......
18.   </joint>
19.   <link name="right_tip">
20.   ......
21.   </link>
```

Ein Strukturdiagramm wird verwendet, um den Zusammenhang zwischen den einzelnen Teilen darzustellen (siehe **Abbildung 5-2)**:

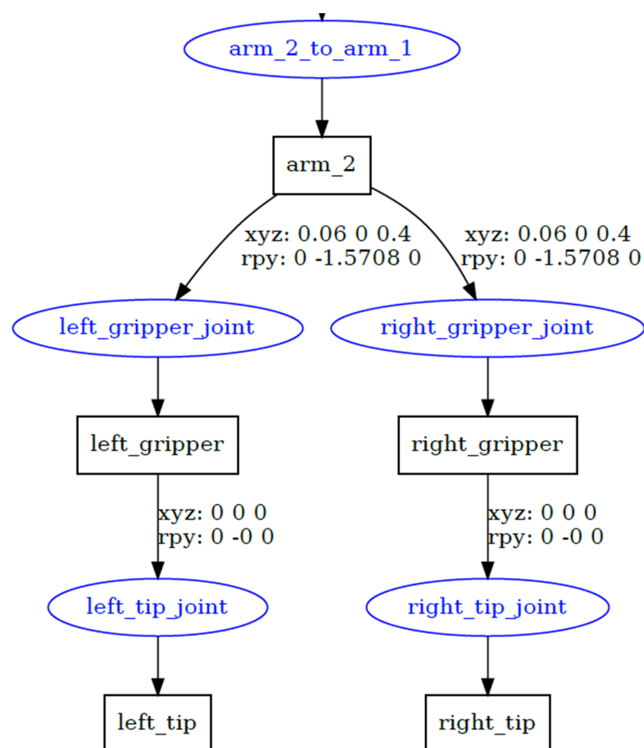


Abbildung 5-2 Strukturdiagramm

.

5.1.3 Importieren von SolidWorks fertigen *.dae-Dateien

Nachdem die mechanischen Backen in SolidWorks konstruiert wurden, wird das Plug-In SolidWorks to URDF Exporter verwendet. [22] Dessen Zweck ist es, die von SolidWorks erstellten Dateien in ein Dateiformat zu konvertieren, das von der ROS-Plattform gelesen werden kann.

Nach der Konvertierung wird eine *.dae-Datei erzeugt, die aus der URDL-Datei kompiliert und mit dem zuvor erstellten Roboterarmteil verknüpft werden kann. Die von SolidWorks erzeugten Dateien werden in einem bestimmten Ordner namens "Meshes" abgelegt, der sich im ROS-Projektpaket befindet. Die Datei könnte also in der folgenden Form geschrieben werden:

```
1.  <link name="left_gripper">
2.      <visual>
3.          <origin rpy="0 0 0" xyz="0 0 0"/>
4.          <geometry>
5.          <mesh filename="package://robot_model/meshes/l_finger.dae"/>
6.          </geometry>
7.          </visual>
8.      <collision>
9.              <geometry>
10.             <box size="0.1 0.1 0.1"/>
11.             </geometry>
12.     </collision>
13.     <inertial>
14.          <mass value="1"/>
15.          <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
16.     </inertial>
17. </link>
```

Der Code in der fünften Zeile bezieht sich auf die *.dae-Datei für den mechanischen Greifer. Die mechanischen Greifer sind in zwei Teile geteilt, links und rechts, der obige

Code beschreibt die linke Hälfte und kann für den rechten Teil in gleicher Weise verwendet werden. Damit ist der gesamte URDF-Teil unseres Roboters abgeschlossen. Die Verteilungsstruktur des gesamten Abschnitts ist in **Abbildung 5-3** dargestellt:

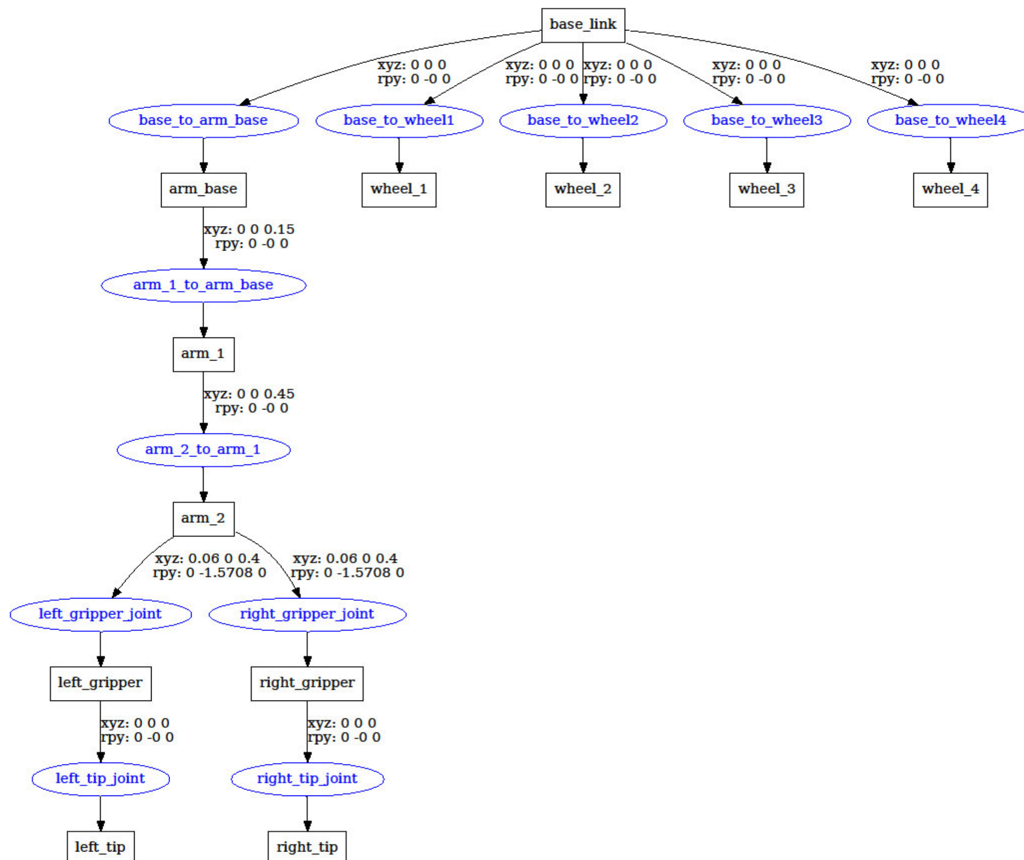


Abbildung 5-3 Verteilungsstruktur des Roboters

5.2 Modifizierung der Launch-Datei

Das Beladungsprinzip für einen Wagen mit einem Roboterarm ist das gleiche wie für einen separaten Wagen. Jetzt muss es nur noch durch die Rviz-Simulation mit dem Befehl Roslaunch angezeigt werden. Da die folgenden Aufrufe der Fahrgestellsteuerungsdatei benötigt werden, um die Bewegung des Roboterarms und des Wagens über das Python-Programm zu steuern, wird diese Datei direkt in die Launch-Datei geladen. Der Code des Programms wird wie folgt geändert:

```
1.    <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" >
```

```
2.        <param name="rate" value="100"/>
3.        <rosparam param="source_list">["/robot1/joint_state"] </rosparam>
4.    </node>
5.    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
6.    <node name="robot_base_control" pkg="robot_model" type="robot_base_control.py" />
```

Unter diesen Bedingungen führt die Eingabe der Befehle in Abschnitt 4.5.2 dazu, dass der Wagen mit dem Roboterarm im Simulator RVIZ erscheint.

```
1.    roslaunch robot_model robot_model.launch
```

Anschließend dann wird das Modell in RVIZ jedoch mit einem Roboterarm anstelle des ursprünglichen Basisfahrzeugs gezeigt. Dies ist in **Abbildung 5-4** dargestellt.

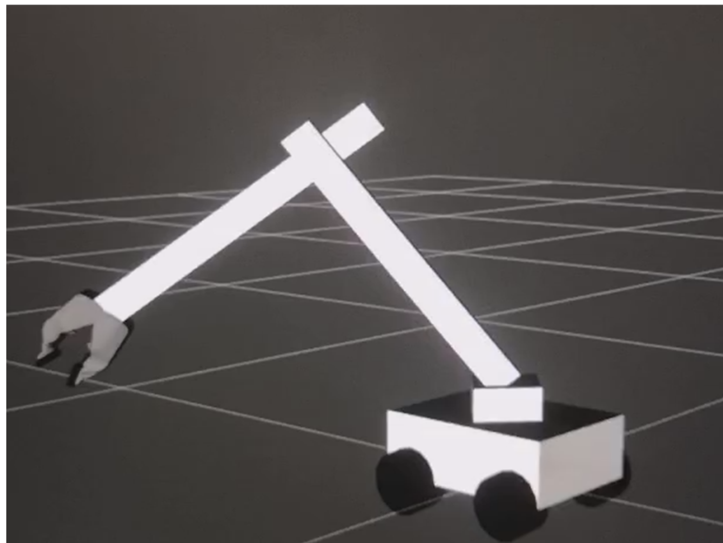


Abbildung 5-4 Modell mit einem Roboterarm

5.3 Automatische Steuerung mit Roboterarmwagen

In Abschnitt 4.5 wurde besprochen, wie der Fahrweg des Wagens über die Tastatur gesteuert werden kann. Diese Tastatursteuerung basiert auf der Bedienung der Wagenchassis, so dass die Steuerung auch für Wagen mit Roboterarmen funktioniert.

Jetzt muss diskutiert werden, wie der Prozess der Automatisierung des Roboterwagens abläuft. Ein Steuerungssystem besteht aus Teilsystemen und Prozessen (oder Anlagen), die zu dem Zweck zusammengesetzt sind, eine gewünschte Ausgabe mit gewünschter

Leistung bei einer vorgegebenen Eingabe zu erhalten. [23, p. 2] Es gibt zwei Grundformen von Steuerungen, das Open-Loop-System und das Close-Loop-System.

5.3.1 Open-Loop-System

Ein generisches Open-Loop-System ist in **Abbildung 5-5** dargestellt. Es beginnt mit einem Teilsystem, dem so genannten Eingangswandler, der die Form des Eingangs in die vom Regler verwendete Form umwandelt. Der Regler steuert einen Prozess oder eine Anlage. Der Eingang wird manchmal als Sollwert bezeichnet, während der Ausgang als Regelgröße bezeichnet werden kann. [23, p. 7]

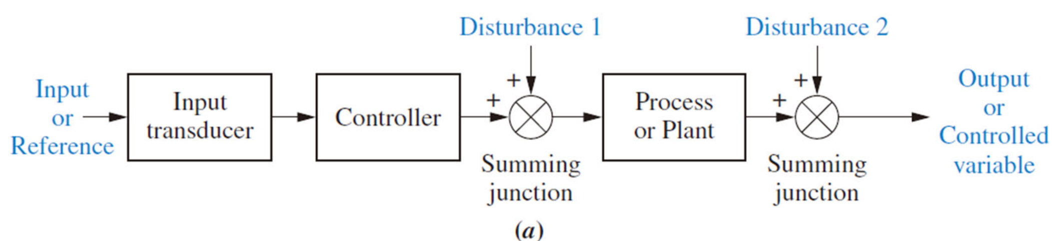


Abbildung 5-5 Open-Loop-System [23]

Daher ist die offene Steuerung eine Systemsteuerungsmethode ohne Rückkopplungsinformationen. Wenn der Bediener das System startet und in Betrieb nimmt, überträgt das System die Befehle des Bedieners sofort an den Roboterwagen. Danach kann der Bediener keine weitere Kontrolle über die Änderung des gesteuerten Objekts vornehmen. Auf diese Weise kann der Roboter gesteuert werden, um Bewegungen zu simulieren.

Auch im Python-Code für die Steuerung muss die Steuerinformation in einem Zug vom Roboterwagen zum System übertragen werden. Für diese Informationen werden Variablen eingeführt, wie im folgenden Code gezeigt:

```
1.  class MoveRobotOpen():
2.      def __init__(self):
3.          rospy.init_node('move_robot_open', anonymous=False)
4.          rospy.on_shutdown(self.shutdown)
5.          self.cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
```

Danach müssen die Laufliniengeschwindigkeit und Winkelgeschwindigkeit des Roboterwagens eingestellt werden:

```
1.      rate = 50
2.      r = rospy.Rate(rate)
3.      linear_speed = 0.2      # Laufliniengeschwindigkeit
4.      goal_distance = 1.0
5.      linear_duration = goal_distance / linear_speed
6.      angular_speed = 2.0     #Winkelgeschwindigkeit
7.      goal_angle = pi
```

Als nächstes wird die Regelperiode anhand der linearen Geschwindigkeit und des Zielabstands berechnet, dann wird der Roboter im Open Loop geregelt. Auch für die Drehung des Roboterwagens ist es notwendig, mit Hilfe der Winkelgeschwindigkeit und des Zielwinkels die Regelperiode zu berechnen und dann die Open Loop Regelung durchzuführen. Im Folgenden wird ein Teil des Codes gezeigt:

```
1.   for i in range(4):
2.       move_cmd = Twist()
3.       ove_cmd.linear.x = linear_speed
4.   ......
5.       for t in range(steps):
6.           self.cmd_vel.publish(move_cmd)
7.           r.sleep()
8.       print("duration: " , rospy.Time.now().to_sec() - t1)
9.
10.      move_cmd = Twist()            # Erst stoppen, dann drehen
11.      self.cmd_vel.publish(move_cmd)
12.      rospy.sleep(0.3)
13.
14.      move_cmd.angular.z = angular_speed          #Schritte zählen
15.  ......
16.          r.sleep()
17.      move_cmd = Twist()
18.      self.cmd_vel.publish(move_cmd)
```

```
19.         rospy.sleep(0.3)
20.     ......
```

5.3.2 Closed-Loop (Feedback Control) Systems

Die Nachteile von Systemen mit offenem Regelkreis, nämlich die Empfindlichkeit gegenüber Störungen und die Unfähigkeit, diese Störungen zu korrigieren, können in Systemen mit geschlossenem Regelkreis überwunden werden. Die generische Architektur eines geregelten Systems ist in **Abbildung 5-6** dargestellt. [23, p. 8]

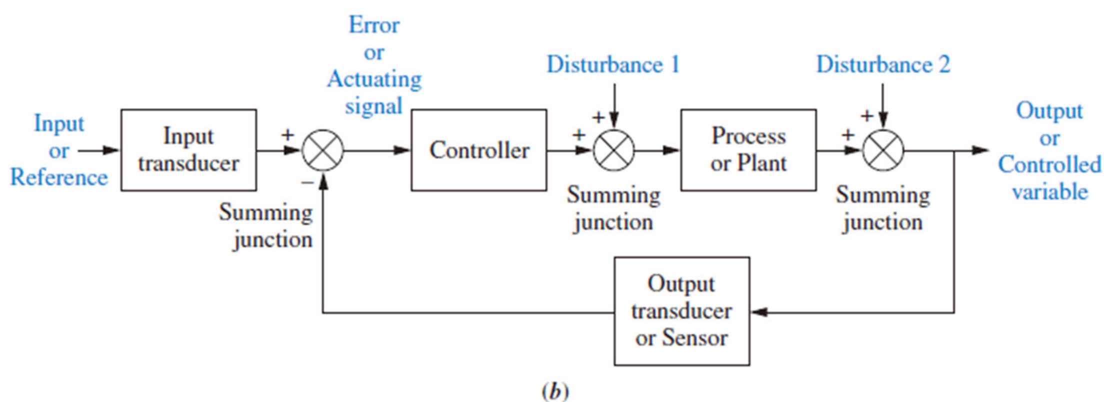


Abbildung 5-6 Closed-Loop (Feedback Control) Systems [13]

Im Gegensatz zu Systemen mit offenem Regelkreis sind bei geschlossenen Regelkreisen bei jedem Schritt der Vorwärtsbewegung des Roboterschlittens Rückwertberechnungen für die Punkte der Relativbewegung erforderlich. Das heißt, die Berechnungen und Kombinationen müssen erneut durchgeführt werden, während die Knoten Informationen freigeben. Im Folgenden wird ein Teil des Codes gezeigt:

```
1.  class MoveRobotClose():
2.      def __init__(self):
3.          rospy.init_node('move_robot_close', anonymous=False)
4.
5.          rospy.on_shutdown(self.shutdown)
6.          self.cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=5)
7.
8.          rate = 50
9.          r = rospy.Rate(rate)
10.         linear_speed = 0.5
```

```
11.         goal_distance = 1.0
12.         angular_speed = 2.0
13.         angular_tolerance = radians(1.0)     # Winkeltoleranz
14.         goal_angle = pi/2
15.         self.tf_listener = tf.TransformListener() # tf-Zuhörer
16.         rospy.sleep(0.1)
17.         self.odom_frame = '/odom'
```

Nachdem die Umgebungsvariablen und der Anfangswert für die Geschwindigkeit des Wagens gesetzt wurden, muss eine Funktion namens waitForTransform aufgerufen werden. Diese Funktion muss deshalb aufgerufen werden, weil bei einer Regelung normalerweise der Empfänger Informationen beim Sperren seiner eigenen entsprechenden Bewegung zurück an den Hörer empfängt, jeder Hörer jedoch einen Puffer hat, der alle Koordinatentransformationen der verschiedenen tf-Sender speichert. Wenn ein Sender eine Transformation sendet, dauert es einige Zeit (in der Regel ein paar Millisekunden), bis die Transformation in den Puffer gelangt. Wenn also zum Zeitpunkt "jetzt" eine Koordinatensystemtransformation angefordert wird, sollte einige Millisekunden gewartet werden, um diese Information zu erhalten. [20, pp. 60-61]

Das tf-Toolkit für die ROS-Plattform bietet ein großartiges Werkzeug, das diesen Warteschritt direkt in die Schleife der Funktion einfügt und die Überwachung des Rückgabewerts der Koordinatentransformation einfach durch Verwendung dieser waitForTransform-Funktion beim Aufruf der Funktion ermöglicht. Im Folgenden wird ein Teil des Codes gezeigt:

```
1.     try:
2.         self.tf_listener.wai        orTransform(self.odom_frame, '/base_footprint', ro
       spy.Time(), rospy.Duration(1.0))
3.         self.base_frame = '/base_footprint'
4.         except (tf.Exception, tf.ConnectivityException, tf.LookupException):
5.         try:
6.             self.tf_listener.waitForTransform(self.odom_frame, '/base_link', rospy.Ti
       me(), rospy.Duration(1.0))
7.             self.base_frame = '/base_link'
8.         except (tf.Exception, tf.ConnectivityException, tf.LookupException):
```

```
9.            rospy.loginfo("Cannot find transform between /odom and /base_link or /base_footprint")
10.           rospy.signal_shutdown("tf Exception")
```

Danach muss der Roboterwagen gesteuert werden, um sich zu bewegen. Mit Hilfe einer *for*-Schleife und einer *While*-schleife wird die Entfernung von der aktuellen Position zum Startpunkt berechnet, indem die aktuelle Position abgerufen wird und so der Satz festgelegt wird, der zur Steuerung der Bewegung des Wagens benötigt wird. Im Folgenden wird ein Teil des Codes gezeigt:

```
1.  for i in range(4):
2.      move_cmd = Twist()
3.      move_cmd.linear.x = linear_speed
4.      (position, rotation) = self.get_odom()
5.      x_start = position.x
6.      y_start = position.y
7.      distance = 0
8.      while distance < goal_distance and not rospy.is_shutdown():
9.          self.cmd_vel.publish(move_cmd)          # Steuerbefehle ausgeben
10.         r.sleep()
11.         (position, rotation) = self.get_odom()      # Aktuelle Position ermitteln
12.             # Berechnen der Entfernung zum Startpunkt
13.         distance = sqrt(pow((position.x - x_start), 2) +
14.                         pow((position.y - y_start), 2))
```

Die Wagen sollen in einer geschlossenen, quadratischen Schleife fahren. Wenn der Roboterwagen jede variable Länge abfährt, wird für die zu drehenden Teile die Planungsidee verwendet, zu stoppen und dann zu drehen. Wenn der Wagen die Route beendet hat, muss die Information über den Kilometerstand durch *rospy.loginfo("TF Exception")* exportiert werden, ein Teil des Codes lautet wie folgt:

```
1.  def get_odom(self):
2.    try:
3.      (trans, rot)  = self.tf_listener.lookupTransform(self.odom_frame, self.base_frame, rospy.Time(0))
4.    except (tf.Exception, tf.ConnectivityException, tf.LookupException):
```

```
5.          rospy.loginfo("TF Exception")
6.          return
7.      return (Point(*trans), quat_to_angle(Quaternion(*rot)))
8.  def shutdown(self):
9.      rospy.loginfo("Stopping the robot...")
10.     self.cmd_vel.publish(Twist())
11.     rospy.sleep(1)
```

5.4 Automatische Steuerung mit Roboterarm

Nachdem die Programmierung des Wagens abgeschlossen ist, soll jetzt die Steuerung der Manipulation des Roboterarms erfolgen. Da das Steuersystem des Computers diskontinuierlich ist, sind die simulierten Bewegungen eine Kombination aus jedem einzelnen Schritt, so dass die Echtzeit-Inkremente für jede Bewegung eingestellt werden müssen. Der Code wird unten gezeigt:

```
1.  inc= 0.005
2.  base_arm_inc = 0.005
3.  arm1_armbase_inc= 0.005
4.  arm2_arm1_inc= 0.005
5.  gripper_inc= 0.005
6.  tip_inc= 0.005
7.  base_arm = 0
8.  arm1_armbase = 0
9.  arm2_arm1 = 0
10. gripper = 0
11. tip = 0
```

Anschließend werden alle Gelenknamen des Roboterarms in die Liste aufgenommen und die Position der einzelnen Gelenke initialisiert.

```
1.  self.joint_state = JointState()
2.  self.joint_state.name.append("base_to_arm_base")
3.  self.joint_state.name.append("arm_1_to_arm_base")
4.  self.joint_state.name.append("arm_2_to_arm_1")
5.  ......
```

```
6.   self.joint_state.position = [base_arm, arm1_armbase, arm2_arm1, gripper, tip, g
     ripper, tip ,0,0,0,0]
```

Nachdem die einzelnen Joint-Positionen initialisiert worden sind, muss der Joint-Position-Publisher erstellt und die angepasste Funktion für den Joint-Abschnitt aktualisiert werden. Im Folgenden wird ein Teil des Codes gezeigt:

```
1.       self.joint_pub =  rospy.Publisher("/robot1/joint_state", JointState, queue_s
     ize=10)
2.   def update(self):
3.       global inc, base_arm_inc, arm1_armbase_inc, arm2_arm1_inc
4.       global gripper_inc, tip_inc, base_arm, arm1_armbase, arm2_arm1, gripper,
     tip
5.       self.joint_state.header.stamp = rospy.Time.now()
```

Anschließend muss jede Information zum Bewegungsjoint über den zuvor erstellten Joint Position Publisher veröffentlicht werden. Der Wert jedes Teils um ein Inkrement gestiegen wird, um die Einzelschrittbewegung des Roboterarms zu steuern. Wenn die Position außerhalb des Bereichs liegt, wird die Rückwärtsbewegung eingeleitet.

```
1.   self.joint_pub.publish(self.joint_state)
2.       while not rospy.is_shutdown():
3.           arm2_arm1 += arm2_arm1_inc
4.           if (arm2_arm1<-1.5 or arm2_arm1>1.5):
5.               arm2_arm1_inc *= -1
6.   arm1_armbase += arm1_armbase_inc
7.           if (arm1_armbase>1.2 or arm1_armbase<-1.0):
8.               arm1_armbase_inc *= -1
9.   ......
10.          self.joint_state.position = [base_arm, arm1_armbase, arm2_arm1, grippe
     r, tip, gripper, tip ,0,0,0,0]
11.          self.joint_state.header.stamp = rospy.Time.now()
12.          self.joint_pub.publish(self.joint_state)
13.
14.          rospy.sleep(0.03)
```

Schließlich wird der aktualisierte Zustandswert über den Publisher des Knotens veröffentlicht, so dass die Steuerung unseres Roboterarms abgeschlossen ist.

6 Implementierung der Software

In den beiden vorangegangenen Kapiteln wurde gezeigt, wie der Programmierprozess zur Entwicklung eines Roboters in der ROS-Umgebung unter Verwendung der Sprache Python durch die ROS-Plattform umgesetzt wird. In diesem Kapitel wird beschrieben, wie der Roboterwagen im Rviz-Simulator für die Ros-Plattform funktioniert.

Da die Ros-Plattform Python-Dateien direkt lesen kann, brauchen diese nicht wie *C++-Dateien* im Voraus kompiliert zu werden, aber die Voraussetzung ist, dass die Eigenschaften der Python-Datei geändert werden müssen, damit sie ausführbar ist, wie in der **Abbildung 6-1** gezeigt.

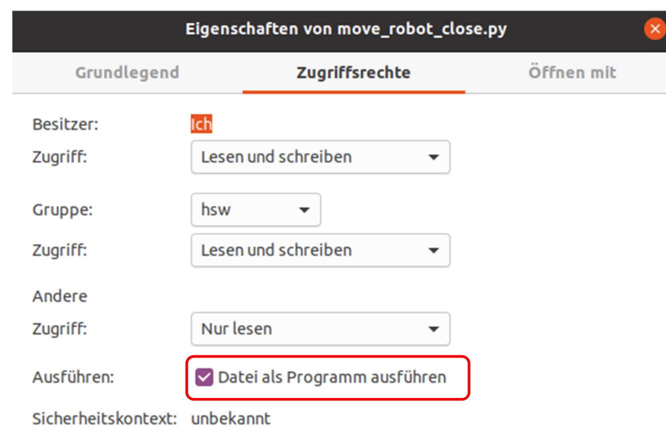


Abbildung 6-1 Eigenschaften der Python-Datei

Als Nächstes werden die Abhängigkeiten für das Programm hinzugefügt, und auch hier kann Ros System das Programm erkennen und korrekt ausführen. Also werden die vom Programm benötigten Abhängigkeiten in package.xml mit dem unten gezeigten Code hinzugefügt:

```
1. <depend>fake_localization</depend>
2. <depend>laser_filters</depend>
3. <depend>robot_state_publisher</depend>
4. <depend>roscpp</depend>
5. <depend>rospy</depend>
6. <depend>std_msgs</depend>
```

```
7.    <depend>tf</depend>
8.    <depend>urdf</depend>
9.    <depend>xacro</depend>
```

Da diese Abhängigkeiten vom Ros-System mitgebracht werden, sollten die Abhängigkeiten außerdem in der *CMakeLists.txt* im Programm referenziert werden, damit das System beim Kompilieren des Programms die Abhängigkeiten für jede Datei genau identifizieren kann.

```
1.  find_package(catkin REQUIRED COMPONENTS
2.    laser_filters
3.    robot_state_publisher
4.    geometry_msgs
5.    roscpp
6.    rospy
7.    std_msgs
8.    tf
9.    urdf
10.   xacro
11. )
```

Als nächstes muss unser zuvor kompiliertes Programm durch Catkin kompiliert werden, und die Dateiabhängigkeiten müssen zu *.Bashrc* hinzugefügt werden. Wir tragen den Befehl wie unten gezeigt ein:

```
1.  cd robot_model_ws
2.  catkin_make
3.  source devel/setup.bash
```

An diesem Punkt ist die von uns geschriebene Datei erfolgreich auf unserem System kompiliert und wird nun über den Befehl roslaunch in **Abschnitt 5.2** ausgeführt.

```
1.  roslaunch robot_model robot_model.launch
```

Nach dem Ausführen von Roslaunch wird die Rviz-Software automatisch geöffnet und die zuvor geschriebene URDF-Modelldatei wird visualisiert. Dies ist in **Abbildung 6-2** dargestellt:

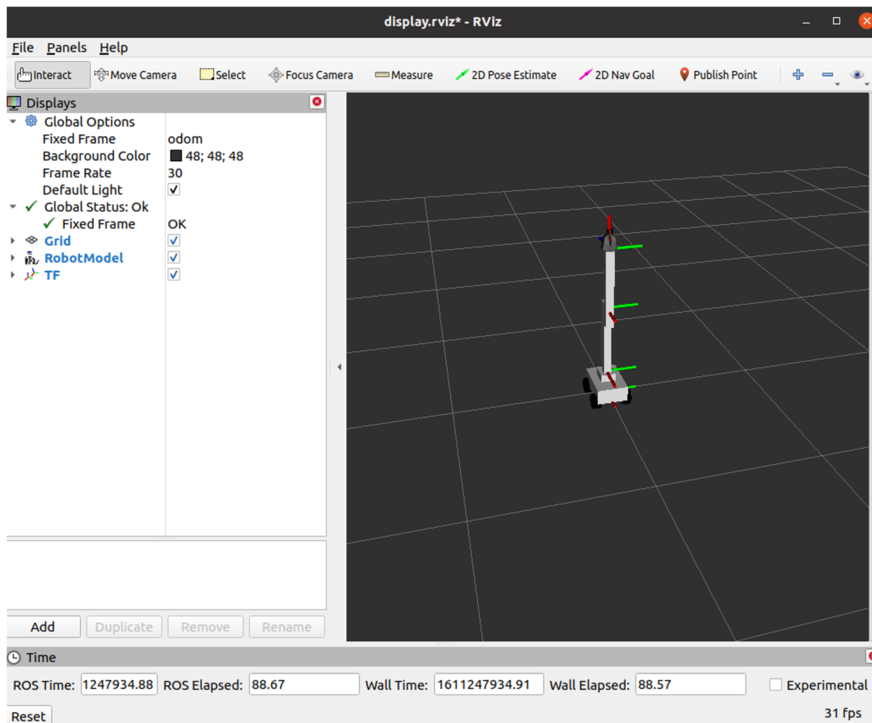


Abbildung 6-2 Modell im Rviz

Sobald unser Wagenmodell mit einem Roboterarm in Rviz angezeigt wird, muss es mit der bereits geschriebenen Python-Datei gesteuert werden. Dazu wird ein neues Terminal geöffnet und der folgende Befehl in das Terminal eingegeben. Die erste ist die Tastatursteuerung für den Wagen.

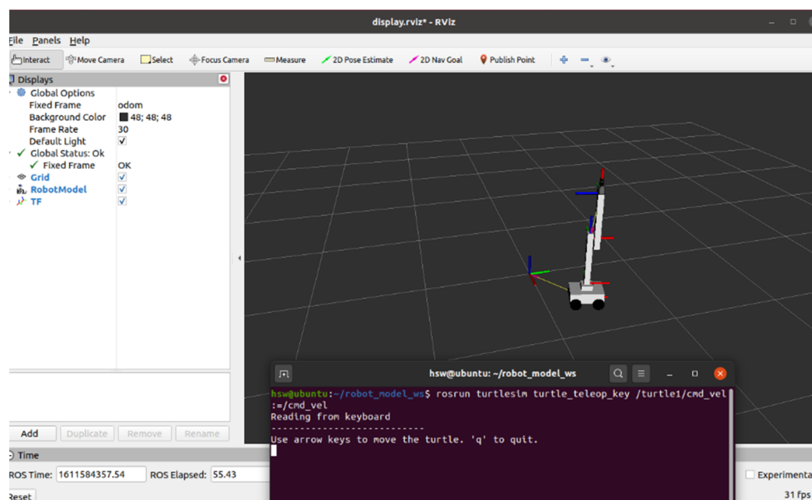


Abbildung 6-3 Tastatursteuerung im Rviz

1. rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/cmd_vel

Nach der Eingabe dieses Befehls kann der Wagen sowohl vorwärts und rückwärts als auch in verschiedene Drehrichtungen gesteuert werden, indem die Tasten nach oben, unten, links und rechts auf der Tastatur des Terminals gedrückt werden. Dies ist in **Abbildung 6-3** (s.o.) dargestellt.

Außerdem kann die Steuerung automatisiert werden, indem die Automatisierungsdatei, die in **Kapitel 5-3** bereits beschrieben wurde, direkt im geöffneten neuen Terminal ausgeführt wird, so dass der Roboter ohne Tastatur im Simulator bewegt werden kann.

Die automatische Steuerung umfasst das Open-Loop-System und das Closed-Loop-System. Die die Befehle werden im Folgenden gezeigt:

```
1.   rosrun robot_model move_robot_close.py
2.   rosrun robot_model move_robot_open.py
```

Da die Bewegung des Wagens und des Roboterarms relativ unabhängig voneinander ist, kann der Roboterarm eine automatische Steuerung durchführen, während der Betrieb des Wagens gesteuert wird. Deshalb soll der folgende Code gleichzeitig in das neu geöffnete Terminal eingegeben werden. Das Programm läuft wie in **Abbildung 6-4** dargestellt.

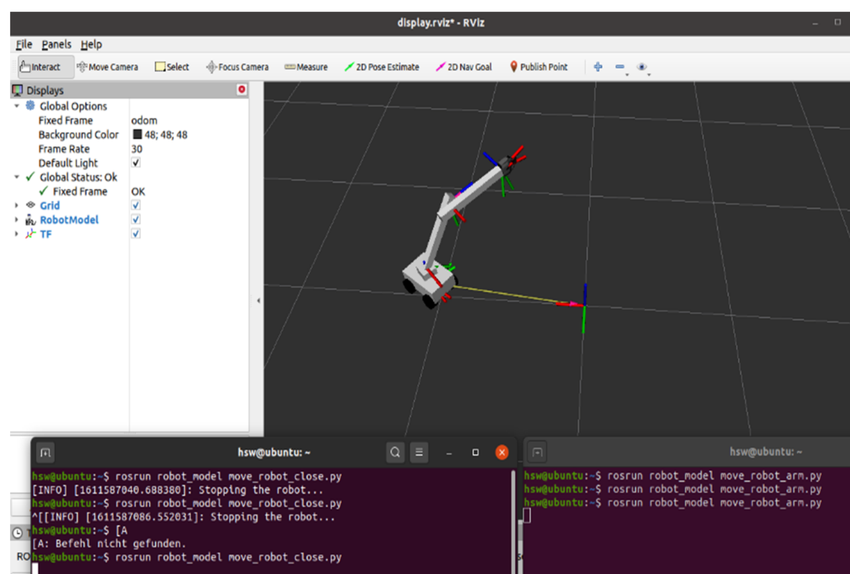


Abbildung 6-4 Das laufenden Programm

```
1.   rosrun robot_model move_robot_arm.py
```

Wenn die Steuerdatei sowohl für den Roboterwagen als auch für den Roboterarm ausgeführt wird, sieht das Knotendiagramm des Programms wie in **Abbildung 6-5** dargestellt aus.

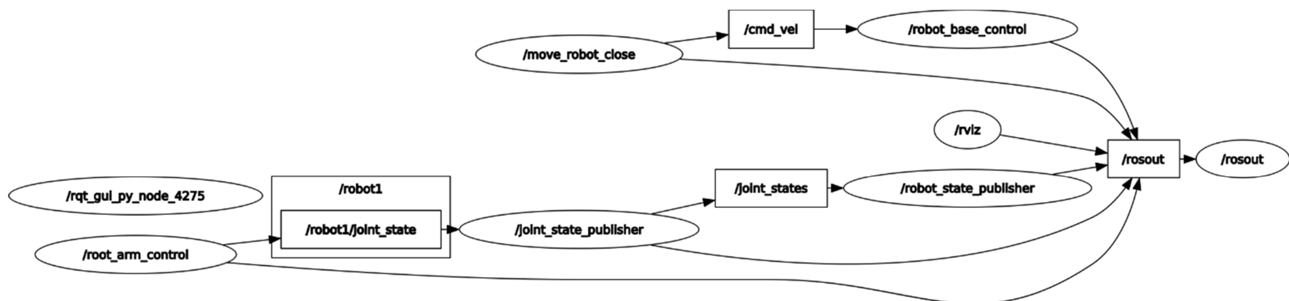


Abbildung 6-5 Knotendiagramm

7 Analyse der Ergebnisse

7.1 Trajektorien des Roboterwagens

In diesem Kapitel werden die zuvor in den Kapiteln 4, 5 und 6 entwickelten Systeme getestet, um die Funktionalität der Entwurfsmethoden zu überprüfen, damit sie im nachfolgenden Prozess korrigiert werden können. Die Routenplanung des Trolleys wird in der nächsten Ausstellung ausführlich besprochen. Die Planung von Trajektorien ist ein grundlegendes Topic für Roboteranwendungen und Automatisierung im Allgemeinen. Die Fähigkeit, Trajektorien mit vorgegebenen Eigenschaften zu erzeugen, ist ein entscheidender Punkt, um signifikante Ergebnisse in Bezug auf Qualität und Einfachheit bei der Ausführung der erforderlichen Bewegung zu gewährleisten, insbesondere bei den in vielen Anwendungen erforderlichen hohen Betriebsgeschwindigkeiten. [24] Im Moment kann der Bewegungsablauf der Roboterwagen für die beiden unterschiedlichen Steuerungsbedingungen und im Rviz-Simulator angezeigt werden.

Da die Trajektorie des Roboterwagens in dieser Arbeit so konzipiert ist, dass sie unter einer bestimmten Route fährt, kann die durch die Anführung des Nodes von Path angezeigt werden, um die Trajektorie im Simulator von Rviz anzeigen zu lassen. Da unsere Roboterbahn ein Rechteck mit geschlossener Schleife ist und die Route aus Punkten und Linien besteht, wird zunächst gelernt, wie Punkte und Linien in Rviz eingeführt werden können. Daher wird auf das Tutorial in Ros Wiki über die Erstellung

von Punkten und Linien in Rviz verwiesen. Die Marker POINTS, LINE_STRIP und LINE_LIST verwenden alle das points-Mitglied der Nachricht visualization_msgs/Marker. Der Typ POINTS platziert einen Punkt an jedem hinzugefügten Punkt. Der Typ LINE_STRIP verwendet jeden Punkt als Scheitelpunkt in einer verbundenen Menge von Linien, wobei Punkt 0 mit Punkt 1, 1 mit 2, 2 mit 3 usw. verbunden ist. Der Typ LINE_LIST erzeugt aus jedem Punktpaar unverbundene Linien, d. h. Punkt 0 zu 1, 2 zu 3, usw. [25] Nach dem Prinzip der Punktverbindung werden die Positionen, an denen sich jeder Einheitszeitpunkt des Roboterschlittens befindet, miteinander verbunden, so dass der Verlauf der Bewegung vom Roboterwagen dargestellt werden kann. Anschließend wird die Begründung für die Trajektorien erhaltet. Es ist notwendig, die Nachricht per Marker-Display an Rviz zu senden. Das Marker-Display ermöglicht im Gegensatz zu anderen Displays die Anzeige von Visualisierungsdaten in Rviz, ohne dass Rviz etwas über die Interpretation dieser Daten wissen muss. Stattdessen werden rohe Objekte über visualization_msgs/Marker-Nachrichten an die Anzeige gesendet, mit denen Sie Dinge wie Pfeile, Kästen, Kugeln und Linien anzeigen können. [26]

Der Bewegungszustand des Wagens wird ermittelt, indem die Werte der *X*- und *Y*-Variablen *dx* und *dy* in den aktuellen Koordinaten aufgerufen werden. Wenn die Bewegungsvariablen *dx, dy* des Wagens größer als 0,05 Einheiten sind, wird ein Punkt für seinen Zustand markiert und alle markierbaren Punkte werden verbunden, so dass der Verlauf der Bewegung des Wagens markiert werden kann. Ein Teil des Codes ist unten dargestellt:

```
1.  bool is_new_pose(const geometry_msgs::Pose& pose)
2.  {
3.      float dx = pose.position.x - last_pose.position.x;
4.      float dy = pose.position.y - last_pose.position.y;
5.  
6.      double dis = sqrt(dx*dx+dy*dy);
7.      std::cout << dis << std::endl;
8.      if(dis > 0.01)
9.      {
10.         last_pose = pose;
```

```
11.             return true;
12.         }
13.
14.         return false;
15.     }
16.     void odom_callback(const nav_msgs::Odometry::ConstPtr& msg)
17.     {
18.         double time = ros::Time::now().toSec();
19.         if(time - last_pub_path_time < 0.05)
20.         {
21.             //std::cout << "time near!\n";
22.             return;
23.         }
24.
25.         last_pub_path_time = time;
26.
27.         if(!is_new_pose(msg->pose.pose))
28.         {
29.             std::cout << "pose near!\n";
30.             return;
31.         }
32.
33.             path.header.stamp=ros::Time::now();
```

Da der Code im offiziellen Ros-Tutorial in C++ ausgeführt wird, sind die Befehle am Ende von CMakeLists.txt hinzuzufügen, um das C++-Programm zum Laufen zu bringen. Die Befehle sind unten dargestellt:

```
1.  add_executable(show_path_node src/showpath.cpp)
2.  target_link_libraries(show_path_node ${catkin_LIBRARIES})
```

Nachdem die neue Datei mit Catkin_make kompiliert wurde, kann die Datei Showpath ausgeführt werden, damit der Wagen die Trajektorien während der Bewegung anzeigt.

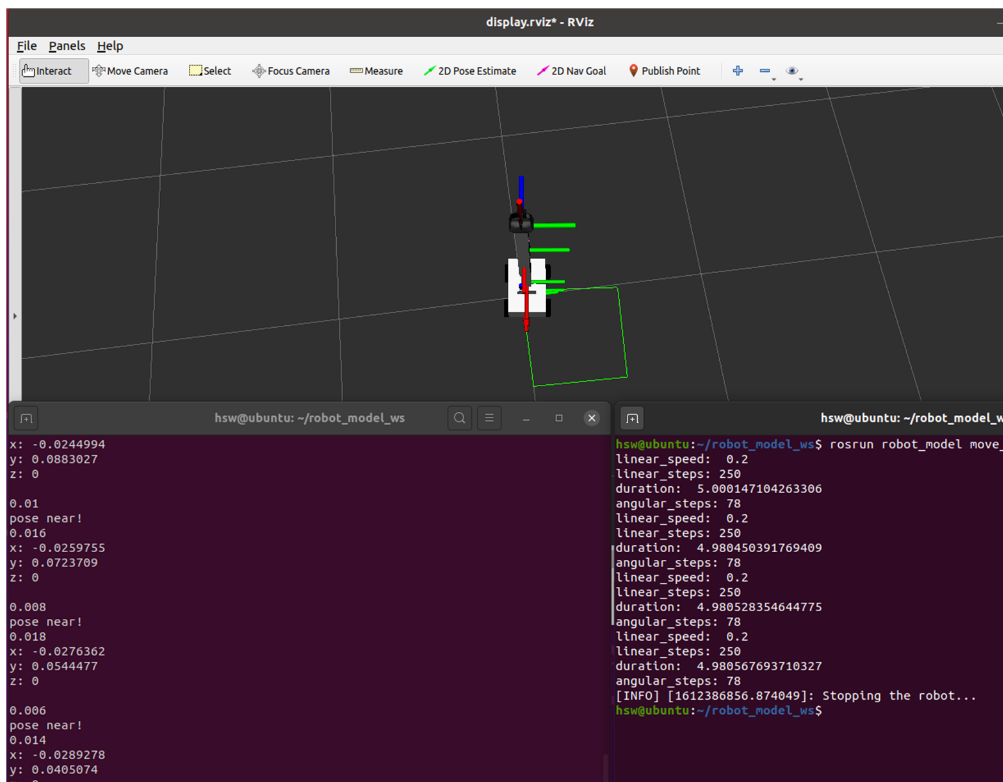


Abbildung 7-1 Showpath von Open-loop Control

Die Ergebnisse des Laufs sind in der **Abbildung 7-1** dargestellt.

Die offene Bewegung des Roboterwagens kann in 4 Schritte unterteilt werden, die 4 Vorschüben sowie der Lenkung entsprechen. In der folgenden Tabelle sind einige der Daten des Roboterwagens während des Prozesses aufgeführt.

Tabelle 7-1 Open-loop Control

linear_speed	0,2	0,2	0,2	0,2
linear_steps	250	250	250	250
duration	78	78	78	78
angular_steps	5.000147104	4.980450391	4.980528354	4.980567693

In ähnlicher Weise werden die Daten für die 4 Läufe sowie die Umdrehungen für den Closed-Loop-Regelkreis ermittelt. Der Prozess der Bedienung ist in **Abbildung 7-2** dargestellt

Tabelle 7-2 Closed-Loop Control

linear_speed	0,2	0,2	0,2	0,2
linear_steps	500	500	500	500
duration	156	156	156	156
angular_steps	5.0034534104	4.990345391	4.990356354	4.990734693

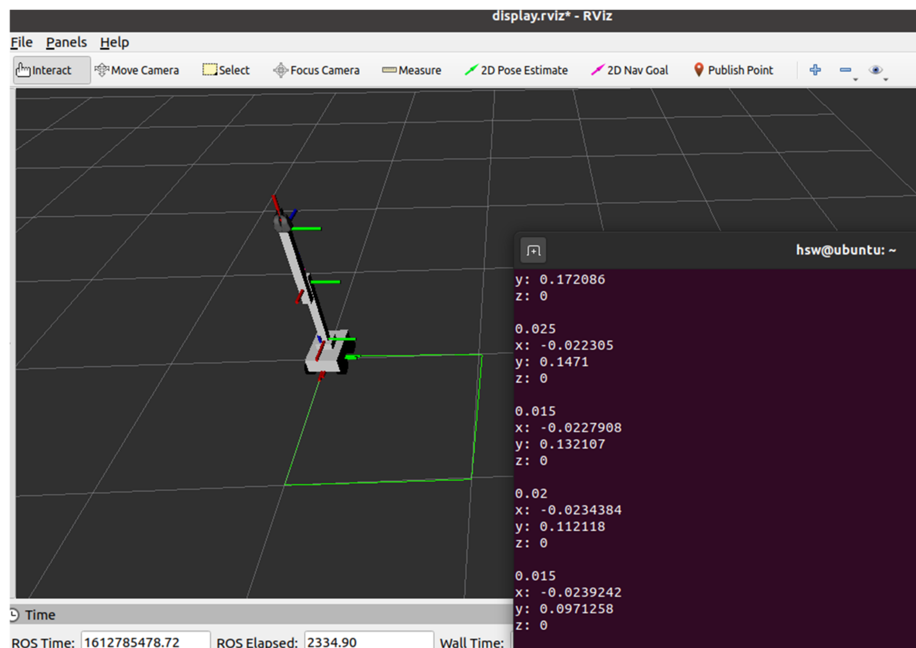


Abbildung 7-2 Showpath von Close-loop Control

7.2 Übliche Slam-Algorithmen in ROS

Obwohl es in dieser Wagenprogrammierung keine Anforderungen bezüglich der Vision-Aspekte des Roboters und der *SLAM*-Algorithmen gibt, sind diese für den folgenden Herstellungsprozess des Roboters unerlässlich.

SLAM *(Simultaneous Localization and Mapping)* beschreibt das Problem von einem mobilen Roboter, bei dem der mobile Roboter oder das autonome Vehikel die Aufgabe hat, gleichzeitig eine Karte zu erstellen und sich zu lokalisieren. Der Roboter muss seine eigene Position auf der Grundlage seiner Kenntnis der zu einem bestimmten Zeitpunkt

verfügbaren Karte ableiten. Die Umgebung sowie die Ausgangsposition des Roboters sind a priori nicht bekannt. [27]

Die fünf gängigen 2D-Laserscanning-basierten SLAM-Algorithmen, die üblicherweise in der ROS-Plattform zur Verfügung stehen, sind *Gmapping, HectorSLAM, KartoSLAM, CoreSLAM und LagoSLAM*. [28] [29]

Gmapping ist ein Open-Source-Algorithmus, der auf dem gefilterten SLAM-Framework basiert und derzeit die am häufigsten verwendete *2D-SLAM-Methode* ist, die Laserpunktwolkendaten in Kombination mit den Roboterposen zur Lokalisierung und Erstellung von Karten verwendet. Es wurde schon in das ROS integriert. Der *RBPF*-Algorithmus (*Rao-Blackwellised Particle Filter*) wird bei der Erstellung von genauen Rasterkarten verwendet. [30]

Das *HectorSLAM-System* ist in 3 Hauptschritten aufgebaut. Der erste Schritt ist die Vorverarbeitung, d. h. ein Filterungsprozess der durch das Lidar gewonnenen Daten, dann werden die Umgebungsmerkmale durch die Kamera extrahiert. Der zweite Schritt ist der Abgleich. Während der kontinuierlichen Erfassung der Umgebung durch das Lidar wird jeder Ort auf der Karte durch das Lidar-Signal verarbeitet, was den einzelnen Punktwolkendaten entspricht. Der dritte Schritt ist die Kartenfusion. Nachdem die tatsächliche Kartenumgebung und die gesammelten Punktwolkendaten nach und nach abgeglichen wurden, können wir die neuen von Lidar gesammelten Daten schrittweise in die Karte einfügen, um eine Kartenaktualisierung zu erreichen. [31]

KartoSLAM ist ein von SRI International's Karto Robotics entwickelter graphbasierter SLAM-Ansatz, der für ROS erweitert wurde, indem eine hochoptimierte und nichtiterative Cholesky-Matrixzerlegung für spärliche lineare Systeme als Löser verwendet wurde. [32]

CoreSLAM ist ein ROS-Wrapper für den originalen 200-Zeilen-Code *tinySLAM*-Algorithmus, der mit dem Ziel erstellt wurde, einfach und leicht verständlich zu sein mit minimalem Leistungsverlust. Der Algorithmus ist in zwei verschiedene Schritte unterteilt: Abstandsberechnung und Aktualisierung der Karte. Im ersten Schritt wird für jeden eingehenden Scan der Abstand auf der Grundlage eines sehr einfachen PF-Algorithmus berechnet. Der PF gleicht jeden Scan aus dem LRF mit der Karte ab und jedes Partikel des

Filters repräsentiert eine mögliche Pose des Roboters und hat ein zugehöriges Gewicht, das von vorherigen Iterationen abhängt. [29]

LagoSLAM ist eine lineare approximative Graph-Optimierung, genauer gesagt wird bei jeder Iteration eine lokale konvexe Approximation des Ausgangsproblems gelöst, um die Graph-Konfiguration zu aktualisieren. Dieser Vorgang wird wiederholt, bis ein lokales Minimum der Kostenfunktion erreicht ist. Dieser Optimierungsprozess ist jedoch stark von den anfänglichen Vermutungen abhängig, um zu konvergieren. [33]

Nach dem ersten Kennenlernen und Verstehen verschiedener Slam-Algorithmen werden deren grundlegende Methoden im nächsten Schritt des Roboterbaus zum Vergleich angewendet. Jede Methode wird sowohl in simulierten Experimenten als auch in realen Experimenten angewandt und die Ergebnisse werden verglichen, um die geeignete Methode zur Diskussion zu stellen.

7.3 Kinematische Modellierung und Simulationsanalyse eines Roboterarms

Für das *Multi-Grad-of-Freiheit-Roboterarm-Trajektorienplanungsproblem* in komplexer Umgebung kann die aktuelle Methode verwendet werden, um die virtuelle Steuerung des Roboterarmmodells durch die ROS-Plattform zu realisieren. Anschließend kann das ROS-Tool *Moveit!* verwendet werden, um die Informationen der einzelnen Gelenkpositionen während der Bewegung des Roboterarms zu exportieren und die Planung weiter zu vervollständigen.

Das in dieser Arbeit behandelte Roboterarmmodell wird unter Verwendung des Unified Robot Description Format (URDF) aufgebaut, um das Roboterarmmodell zu erstellen, die anfängliche Konfiguration des Roboterarms zu vervollständigen und schließlich das Tool Rviz zu verwenden, um die Bewegung der linearen und kreisförmigen Trajektorie des Roboterarms zu simulieren und die Bewegungssteuerung des massiven Roboterarms bereitzustellen. Schließlich wird das Tool Rviz verwendet, um die lineare und kreisförmige Trajektorie-Bewegung des Roboterarms zu simulieren und die Grundvoraussetzungen für die Bewegungssteuerung des massiven Arms zu schaffen.

.

7.3.1 positive kinematische Lösung und Analyse

Ein Roboterarm ist eine Gelenkstruktur, die aus einer Reihe von in Reihe geschalteten Gelenken mit jeweils einem Freiheitsgrad besteht, die eine Translation oder Rotation ausführen können. [34]Die positive kinematische Lösung besteht darin, die Position des Endeffektors mit den einzelnen Gelenkvariablen des Roboterarms und den Strukturparametern des Gestänges zu lösen und dabei die parametrische Methode D-H (Denavit-Hartenberg) zur Beschreibung der Translationsstrecken, Rotationswinkel usw. zwischen zwei benachbarten Koordinatensystemen zu verwenden. [35]

Der Roboterarm in dieser Arbeit besteht aus einer Grundplatte und zwei Hebeln mit einem Manipulator am Ende der Hebel. Die Hauptabmessungen der Konstruktion und die zugehörigen Parameter des für dieses System ausgewählten Roboterarms sind in **Abbildung 7-3** markiert.

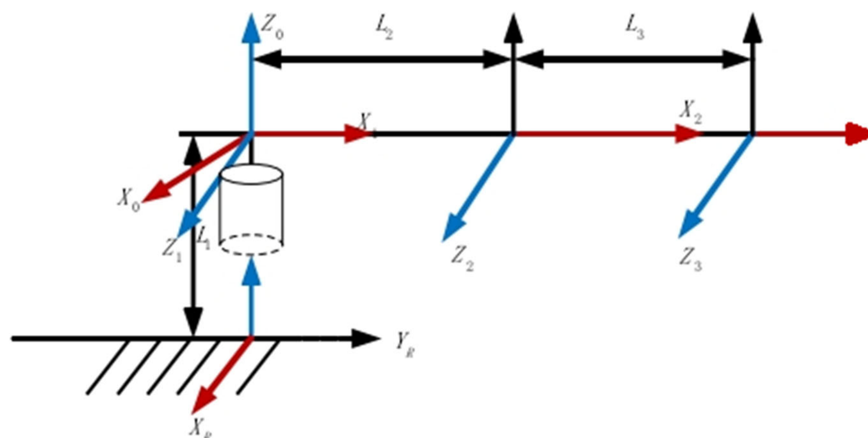


Abbildung 7-3 Koordinatensystem der Roboterarm-Verbindung

Basierend auf der Positionstransformation des Gestänge Koordinatensystems kann dann jeder Gestänge Parameter des Roboterarms wie in **Tabelle 7.3** gezeigt gemessen werden.

Tabelle 7-3 Roboterarm D-H Parameterblatt

Joint	a_i	α_i	d_i	θ_i
L_1	0	90	0.1	0
L_2	0	0	0.5	0
L_3	0	0	0.5	0

7.3.2 Inverse Kinematik lösen und analysieren

Wenn die Position und die Lage des Endeffektors des Roboterarms bekannt sind, ist es notwendig, die Koordinatenbeziehung zwischen dem Endeffektor des Roboterarms und jedem Gelenk zu erhalten, um jedes Gelenk des Roboterarms in die Position zu bewegen, die der Endeffektor-Position entspricht, was auch als Lösungsproblem der inversen Kinematik des Roboterarms bekannt ist.

Für die kinematische Lösung wird im Moveit!-Plugin standardmäßig die KDL-Bibliothek (Kinematics Dynamics Library) verwendet, um die Lösung der Vorwärts- und der inversen Kinematik durchzuführen und es wird eine API-Schnittstelle für den Benutzer bereitgestellt, um die Lösung der inversen Kinematik nach einem eigenen Algorithmus durchzuführen. Daher wird unter Berücksichtigung der Vielfalt der Roboterarmstrukturen die Methode der inversen Matrixmultiplikation gewählt, um die inverse Kinematik des Roboterarms zu lösen [36].

8 Schlussbetrachtung

8.1 Zusammenfassung

Die Projektarbeit handelt von der Steuerung von Industrierobotern unterschiedlicher Hersteller mit dem Betriebssystem ROS. Hier wird eine Roboterplattform mit vier Rädern und einem Roboterarm betrachtet, die innerhalb des Rviz-Simulators der ROS-Plattform betrieben werden kann. Zu diesem Zweck wird untersucht und dokumentiert, wie ein Industrieroboter mit ROS gesteuert werden kann. Der zweite Schritt ist das Erlernen und Einrichten einer ROS-Arbeitsumgebung auf einer virtuellen Maschine. Der dritte Schritt ist die Steuerung des Roboterwagens für das Vorwärts-Rückwärtsfahren und die Lenkung über eine Tastatur. Dazu gehören dann die Steuerung und Regelung des Roboterwagens durch Running Code und schließlich die automatische Steuerung des Roboterarms. In Kapitel 2 und 3 des Artikels wurde der aktuelle Stand der Technik beschrieben und die Einführung in die Verwendung der Grundlagen, Befehle und Funktionen der Ros-Plattform durchgeführt. In Kapitel 4 wurden der Entwurf des Hauptteils des Wagens und einige der Codeläufe durchgeführt. Kapitel 5 beschreibt den Prozess der Herstellung des

Roboterarms und die Implementierung der Verbindung zum Roboterwagen durch Befehle. In Kapitel 6 wird gezeigt, wie das geschriebenen Programmpaket über *Roslaunch* auf der ROS-Plattform ausgeführt wird und der Roboterwagen mit dem Roboterarm durch Befehle bewegt werden kann. In Kapitel 7 wird die Bewegungstrajektorie des Wagens dokumentiert und es werden Informationen zu *Slam-Algorithmen*, die in Zukunft verwendet werden könnten, geprüft und untersucht.

8.2 Ergebnisse der Arbeit

In dieser Projektarbeit wurde ein kleines intelligentes Fahrzeug mit einem Roboterarm auf Basis des Betriebssystems ROS entwickelt, das per Tastatur und automatisch gesteuert werden kann. Der intelligente Wagen kann im Simulator Rviz unter der ROS-Plattform automatisch gesteuert werden, und die Bewegung des Wagens kann per Befehl angezeigt werden. Auf dem Roboterwagen wurde ein einfacher Roboterarm aufgebaut und betrieben, der über verschiedene Codes einfache Greifaktionen ausführen kann. Der Schwerpunkt dieser Arbeit liegt auf der Implementierung des gesamten Systems, einschließlich der Konstruktion des Roboterarms des Wagens. Die Vorstellung verschiedener Algorithmen zu *Slam* wird im Abschnitt Ergebnisse diskutiert. In der Tat sind die intelligente Positionierung und Navigation des Wagens ein sehr großes Thema, das Algorithmen in verschiedenen Kategorien umfasst, und jedes Stück verdient eine detaillierte und eingehende Untersuchung. Das Ziel dieser Projektarbeit ist es, die einzelnen Teile verstehen zu lernen, sie organisch zu kombinieren und schließlich einen Roboterwagen zu realisieren, der in einer Simulator-Umgebung betrieben werden kann.

8.3 Ausblick

Das Ergebnis der Arbeit ist die Implementation eines Roboterwagens mit einem Roboterarm, der in einer Simulationsumgebung arbeiten kann. Hierbei ging es um Softwarebedienung und -design, wobei das System sich als sehr komplex erwiesen hat. So wurde das System zunächst mit einigen Unvollkommenheiten implementiert, bis jetzt hat der Schlitten nach dem Close-Loop-Control einen etwas kleineren Lenkwinkel als π, dieser Fehler ist natürlich bei der Durchführung der Regelung vorhanden. Eine andere

Unzulänglichkeit ist, dass die Bewegung das Roboterarms eher einfach ist, und es noch nicht möglich ist, die Erkennung des Objekts durch den Slam-Algorithmus durchzuführen, sondern nur die Bewegung des Roboterarms auszuführen. In dieser Arbeit wurden nur verschiedene *Slam-Algorithmen* kurz vorgestellt. In einer weiterem Arbeit müssen jedoch die Details und Anwendungen der *Slam-Algorithmen* ausführlich besprochen werden, auf deren Grundlage die visuellen Eingabemodelle mit dem Ziel aufgebaut werden, die Synergien zwischen der Vision und der Bewegung des Roboterwagens zu erreichen. Ein weiterer Punkt ist die kinematische Analyse des mobilen Roboterarms, die die positiven und negativen Lösungen der Betriebswissenschaft beinhaltet. Die Berechnungen ermöglichen es dem Roboterarm, die Zielbewegung im Simulator auszuführen und die Bewegungstrajektorie aufzeichnen zu können. Diese konnten in diesem Artikel sowohl aus zeitlichen als auch aus technischen Gründen nicht ergänzt werden.

Ich hoffe, diese im Rahmen der nächsten Diplomarbeit vervollständigen zu können.

9 Literaturverzeichnis

[1] Deutscher Caritasverband e.V., „Sozialpolitische Positionen Kampagne 2018," *„Jeder Mensch braucht ein Zuhause",* p. 4, 2018.

[2] Christian Richter, Robert Zickler von tu-dresden, „Projektbewilligung eines Mauerrobotors (Wallbot)," 1 11 2019. [Online]. Available: https://tu-dresden.de/ing/maschinenwesen/imd/bm/die-professur/news/projektbewilligung-mauerrobotor-wallrob.

[3] Bjormar Jensen Deloitte AG, „Mensch und Maschine: Roboter auf dem Vorrnarsch?," 2015.

[4] L. Qiang, "Home mobile service robot path planning research," pp. 2-3, 2012.

[5] W. Yongyuan, „System analysis and simulation research of industrial robots," 2014.

[6] X. Jiang, Introduction to Robotics, Liaoning Science and Technology Publishing House, 1994.

[7] S. H. JI Pengcheng, „The current situation of service robots and their development trend," *Natural Science Edition,* pp. 73-78, 2010.

[8] W.-P. L. Tsung-Hsien Yang, „A Service-Oriented Framework for the Development of Home Robots," *International Journal of Advanced Robotic Systems,* pp. 122-132, 2012.

[9] O. Khatib, „Mobile manipulation: The robotic assistant," *Robotics and Autonomous System,* pp. 175-183, 1999.

[10] S. K. ·. J. Shi, „An autonomous mobile manipulator for assembly tasks," *Auton Robot,* 22 1 2009.

[11] C. C. J. F. J. K. A. B. W. T. J. C. P. M. H. A. V. Ulrich Reiser, „Creating a product vision for service robot applications," *The 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems,* 11 10 2009.

[12] L. G. Nan, *Forschung zur Verfolgungssteuerung und Bewegungsplanung eines mobilen Roboterarms,* Shanghai Jiao Tong University .

[13] S. S. S. ·. D. F. ·. C. J. H. ·. D. B. C. ·. R. D. ·. G. G. ·. G. H. ·. K. ·. M. VandeWeghe, „HERB: a home exploring robotic butler," *Auton Robot,* pp. 5-20, 29 1 2009.

[14] D. L. Lirong, Practical tutorials for Ubuntu Linux operating system, Post & Telecom Press Co.,LTD.

[15] Ubuntu, „Growing Ubuntu for cloud and IoT, rather than phone and convergence," 5 4 2017.

[16] baidu, „VMware," baidu, 2019. [Online]. Available: https://baike.baidu.com/item/VMware/5461553?fr=aladdin. [Zugriff am 4 1 2021].

[17] J. A. Mishra R, „based service robot platform[J].," *International Conference on Control,Automation and Robotics,* pp. 55-59, 2018.

[18] T. R. D. F. Z. J. C. Y. L. X. Liu Yachiu, „Simulationsstudie eines Vision-Servo-7DOF-Roboterarms basierend auf ROS und EtherCAT [J].," *Journal of Natural Sciences, Heilongjiang University,* pp. 35(01)94-101, 2018.

[19] R. P. GOEBEL, „ROS By Example," in *A Do-It-Yourself Guide to the ROS*, p. 11.

[20] R. P. GOEBEL, „ROS By Example," in *A Do-It-Yourself Guide to the ROS*, 2012, pp. 8-10.

[21] China University MOOC, „Einführung in Roboterbetriebssysteme Kurs-Handout," China University MOOC, [Online]. Available: https://sychaichangkun.gitbooks.io/ros-tutorial-icourse163/content/chapter3/3.1.html. [Zugriff am 4 1 2021].

[22] wiki.ros, „SolidWorks to URDF Exporter,“ [Online]. Available: http://wiki.ros.org/sw_urdf_exporter. [Zugriff am 8 1 2021].

[23] N. S. Nise, CONTROL SYSTEMS ENGINEERING, Seventh Edition Hrsg., Pomona: California State Polytechnic University, 2014, p. 2.

[24] A. G. ·. P. Boscariol, „Trajectory Planning in Robotics,“ *Mathematics in Computer Science,* p. 1, 2012.

[25] Ros.org, „rviz/Tutorials/Markers: Points and Lines,“ [Online]. Available: http://wiki.ros.org/rviz/Tutorials/Markers%3A%20Points%20and%20Lines. [Zugriff am 1 2 2021].

[26] Ros.org, „Wiki: rviz/Tutorials/Markers: Basic Shapes,“ Ros, [Online]. Available: http://wiki.ros.org/rviz/Tutorials/Markers%3A%20Basic%20Shapes.

[27] F. I. a. T. B. Hugh Durrant-Whyte, „Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms,“ *IEEE Robotics and Automation Magazine,* pp. 99-110, 2006.

[28] L. Yang, „Research and implementation of SLAM algorithm for mobile robot based on ROS system,“ xi'an polytechnic university.

[29] J. M. Santos, D. Portugal und R. P. Rocha, „An evaluation of 2D SLAM techniques available in Robot Operating System,“ *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR),* pp. 2374-3247 , 10 2013.

[30] W. Huanqin, „New photoelectric distance measurement and three-dimensional imaging technology research,“ *Journal of the University of Science and Technology of China,* 2009.

[31] K. K. Shing, „Full-scene self-driving nurse bed design based on Hector SLAM algorithm,“ *Public Communication of Science & Technology,* 4 2020.

[32] B. L. M. E. Régis Vincent, "Comparison of indoor robot localization techniques in the absence of GPS," *The International Society for Optical Engineering,* 4 2010.

[33] R. A. J. A. C. a. B. B. Luca Carlone, „A Linear Approximation for Graph-based Simultaneous Localization and Mapping," *Robotics: Science and Systems,* 2011.

[34] Z. Botao, *Studies of motion planning algorithms and applications for mobile manipulator,* East China University of Science and Technology, 2011.

[35] M. H. ·. A. K. H. S. ·. W. Steinhilper, Kinematik und Robotik, Springer-Verlag Berlin Heidelberg GmbH.

[36] Q. wei, *ROS-BASED PLANNING AND MOTION SIMULATION OF MOBILE MANIPULATOR,* Harbin Institute of Technology, 2014.

[37] C. C. W. Y. C. Xianming, „"Introduction to Robot Operating Systems" course handout," China University MOOC, [Online]. Available: https://sychaichangkun.gitbooks.io/ros-tutorial-icourse163/content/chapter4/4.1.html.

[38] „http://wiki.ros.org," Ros.org, [Online]. Available: http://wiki.ros.org/care-o-bot.

[39] D. Pavlovic, „Analyse und Vergleich von SLAM-Algorithmen," Technische Universität Darmstadt, 2016.

10 Abbildungsverzeichnis

Abbildung 2-1 Drei traditionelle mobile Serviceroboter 4

Abbildung 2-3 Care-O-Bot bringt Bier 5

Abbildung 2-3 HERB und Care-O-bot Roboter [38] 5

Abbildung 2-4 Ubuntu im VMware 6

Abbildung 2-5 Rviz-Emulator 8

Abbildung 3-1 Arbeitsablauf bei der Kompilierung [21] 14

Abbildung 4-1 Wagenkörper 27

Abbildung 4-2 Durchführung in Rviz 34

Abbildung 4-3 Node-Relationship-Diagramm 37

Abbildung 5-1 Entwurf des Roboterarms 38

Abbildung 5-2 Strukturdiagramm 41

Abbildung 5-3 Verteilungsstruktur des Roboters 43

Abbildung 5-4 Modell mit einem Roboterarm 44

Abbildung 5-5 Open-Loop-System [23] 45

Abbildung 5-6 Closed-Loop (Feedback Control) Systems [13] 47

Abbildung 6-1 Eigenschaften der Python-Datei 52

Abbildung 6-2 Modell im Rviz 54

Abbildung 6-3 Tastatursteuerung im Rviz 54

Abbildung 6-4 Das laufenden Programm 55

Abbildung 6-5 Knotendiagramm 56

Abbildung 7-1 Showpath von Open-loop Control 59

Abbildung 7-2 Showpath von Close-loop Control 60

Abbildung 7-3 Koordinatensystem der Roboterarm-Verbindung 63

11 Tabellenverzeichnis

Tabelle 3-1 ROS-Shell-Befehle 10

Tabelle 3-2 Ausführungsbefehl 10

Tabelle 3-3 Meldungsbefehl 11

Tabelle 7-1 Open-loop Control 59

Tabelle 7-2 Closed-Loop Control 60

Tabelle 7-3 Roboterarm D-H Parameterblatt 63