Ausarbeitung zum Projektseminar Echtzeitsysteme

Ausarbeitung eingereicht von
Simon Kadel, Daniel Kadletz, Lars Kliegel, Tim Schneider
am 13. April 2016

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Echtzeitsysteme

Elektrotechnik und Informationstechnik (FB18)

Zweitmitglied Informatik (FB20)

Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt

www.es.tu-darmstadt.de

Gutachter: Prof. Dr. Andy Schürr
Betreuer: Geza Kulcsar

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Aufgabenstellung . . . 1
- 1.2 Zielsetzung . . . 1

2 Übersicht **3**

3 PSES_Car **5**
- 3.1 car_handler . . . 5
- 3.2 controller_manager . . . 6
- 3.3 odometry . . . 7
- 3.4 securitycontroller . . . 9

4 Marker_Module **10**
- 4.1 marker_publish . . . 10
- 4.2 marker_interpreter . . . 11

5 Navigation **13**
- 5.1 Dubins Pfade . . . 13
- 5.2 simple_move_base_controller . . . 14
- 5.3 local_planner . . . 16
- 5.4 move_base_controller . . . 17

6 Joystick_Control **19**

7 US_Controller **20**

8 Einsatz der Packages **22**

9 Zusammenfassung und Fazit **23**

Abbildungsverzeichnis

2.1 Gesamtübersicht der genutzten ROS-Packages 3
3.1 Package-Übersicht von PSES_Car 5
4.1 Package-Übersicht des Marker_Module 10
5.1 Package Übersicht des Navigation-Package 13
5.2 Dubins Pfade 14
5.3 Mögliche Tangenten zwischen zwei Kreisen 15
6.1 Package-Übersicht des Joystick_Controller 19
7.1 Package-Übersicht des US_Controller 20

1 Einleitung

In dieser Ausarbeitung zum Projektseminar Echtzeitsysteme im Wintersemester 2015 / 16 ist die Lösung des Team Randoms dokumentiert.

Im diesem Kapitel wird auf die allgemeine Aufgabenstellung sowie die Team-interne Zielsetzung des Team Randoms im Speziellen eingegangen. Dann folgt eine Beschreibung und Dokumentation unserer Lösung (Kapitel 2-7). In Kapitel 8 wird gezeigt, wie man mit unserer Lösung die Ziele erreichen kann und abschließend wird in Kapitel 9 das Seminar und die Plattform von unserer Sicht aus noch einmal reflektiert und zusammengefasst.

1.1 Aufgabenstellung

Die im Wintersemester 2015 / 16 zum ersten Mal eingesetzte neue Plattform soll durch die Teams mit ersten Funktionen ausgestattet werden. Ziel soll es sein, dass Fahrzeug mittels der vorhandenen Sensorik (Ultraschall- und Hallsensoren, sowie einer Kamera) autonom an einer Wand entlang fahren zu lassen.

Des Weiteren soll die Kamera in Betrieb genommen werden. Hierzu soll eine Möglichkeit zur Erkennung von ArUco-Marker implementiert werden. Als Anwendung zur sinnvollen Nutzung der Kamera müssen Tore aus ArUco-Marker durchfahren werden.

Neben diesem Pflichtprogramm für alle Teams gibt es optionale Vertiefungsmöglichkeiten. Denkbar ist hierbei die Regelung des Fahrzeugs mittels Inertialsensorik zu erweitern. Alternativ ist die Implementierung einer Fernsteuerung sowie einer Car-to-Car Kommunikation möglich.

Ein weiteres Vertiefungsfeld stellt die ROS basierte Simulation dar. Fahrszenarien, die im diesjährigen Projektseminar Echtzeitsysteme umgesetzt werden könnten, sind diverse Arten des Einparken (vorwärts, rückwärts, seitwärts), das Verfolgen von beweglichen ArUco-Markern, sowie Kreuzungsszenarien welche mittels Car-to-Car gelöst werden.

Als Fernziel für die Fahrzeuge des Projektseminars Echtzeitsysteme ist die Teilnahme am Hochschulwettbewerb "Carolo-Cup"formuliert worden.

1.2 Zielsetzung

Die Zielsetzung des Team Randoms besteht neben dem Pflichtprogramm darin, einen möglichst hohen Grad an Wiederverwendbarkeit zu erreichen. Hierzu wollen wir soweit möglich auf bereits vorhandene ROS Funktionen bauen.

Des Weiteren soll sich unsere Implementierung möglichst nah an der ROS üblichen Struktur orientieren, sodass unsere Packages wie im Internet verfügbare ROS-Packages genutzt werden können.

Neben der hohen Wiederverwendbarkeit ist unsere Zielsetzung, unsere Ziele mit der aktuell verfügbaren Plattform zu erreichen, beziehungsweise das Bestmögliche mit der vorhandenen Hardware zu erreichen. Größere Umbauten möchten wir nicht vornehmen, da unsere Lösung dann nicht oder nur mit deutlichem Mehraufwand weiter verwendet werden kann.

Fahrfunktionen, die wir im Laufe des Semesters erreichen möchten, sind das Planen auf einer Karte und das möglichst exakte Abfahren dieses zuvor geplanten Pfads. Hierzu soll sich das Fahrzeug selbstständig in seiner Umgebung orientieren können.

Für die Entwicklung und das Testen einzelner Funktionen ist es außerdem hilfreich, wenn das Fahrzeug manuell gesteuert werden kann. Deshalb möchten wir eine Möglichkeit entwickeln, das Fahrzeug mittels eines Joysticks fernsteuern zu können.

2 Übersicht

Insgesamt wurden durch das Team Randoms fünf ROS-Packages entwickelt. Diese stellen in den einzelnen ROS-Nodes die zur Erfüllung der Aufgabenstellung erforderlichen Funktionalitäten zur Verfügung und werden in den folgenden Kapiteln ausführlich erläutert. In Abb. 2.1 sind die verschiedenen Packages als Gesamtsystem gezeigt. Außerdem ist die Kommuniaktion dargestellt, beschränkt auf die zum Verständnis wichtigen Daten.

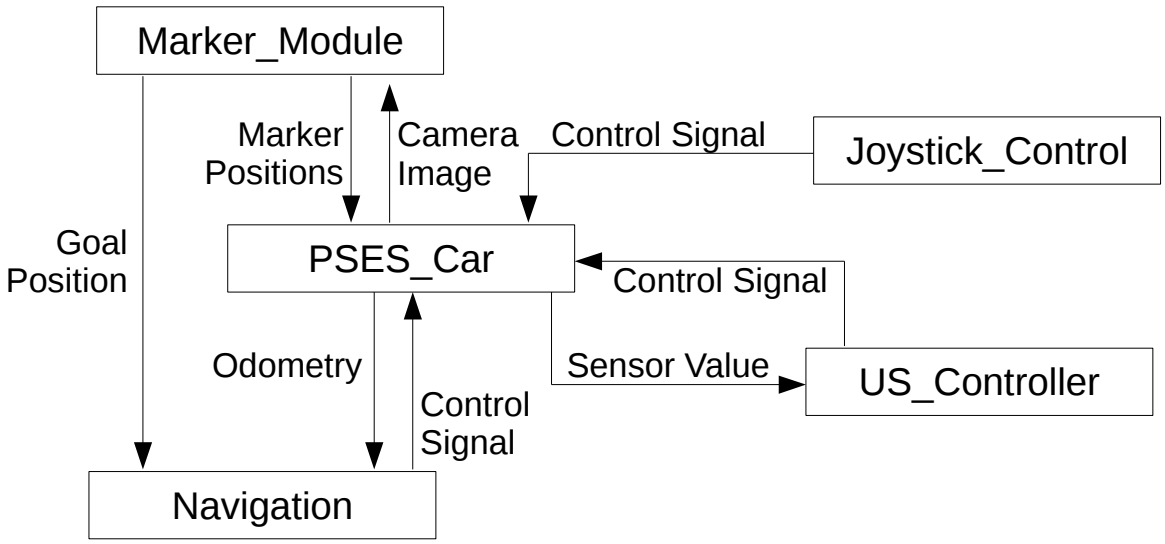

Abbildung 2.1: Übersicht über die entwickelten ROS-Packages und wie diese untereinander kommunizieren.

PSES_Car stellt hierbei die Grundfunktionalitäten zur Verfügung. Dies sind sowohl der car_handler als Schnittstelle zur Hardware, der controller_manager, welcher eingehende Nachrichten gemäß ihrer Priorität verarbeitet sowie die odometry zur Berechnung der aktuellen Position.

Das Navigation-Package stellt den local_planner zur Verfügung, welcher mittels des Konzepts der Dubins Pfade Trajektorien zum Erreichen von Zwischenzielen berechnet und diese an den move_base_controller zur Ausführung weitergibt. Als weiteres Package tritt außerdem das marker_module in Erscheinung. Dieses ist als Ganzes für die Erkennung der ArUco-Marker und deren Interpretation verantwortlich. Durch das marker_module werden einzelne Ziele an den local_planner weitergegeben, sodass diese angefahren werden können.

Neben den bereits aufgeführten Packages werden außerdem der US_Controller zur seitlichen Abstandsregelung mittels Ultraschall-Sensoren, sowie das Joystick_Control-Package zur direkten Steuerung des Fahrzeugs mittels eines Joysticks genutzt.

Da in den verschiedenen Packages zum Teil in unterschiedlichen Koordinatensystemen, auch frames genannt, gerechnet wird, benutzen wir die tf-Bibliothek. Diese er-

möglicht Umrechnungen zwischen den koordinatensystemen. Dazu wird im PSES_Car-Package ein robot_state_publisher-Node gestartet, der die zur Verfügung gestellte Roboterbeschreibungsdatei car.urdf benutzt, um die zur Umrechnung benötigten Informationen bereit zu stellen.

3 PSES_Car

In diesem Kapitel werden die Aufgaben, die Funktionsweise sowie die verwendeten Topics von PSES_Car erläutert. Hierzu ist in Abb. 3.1 ein Überblick über die Zusammenhänge der Nodes und den wichtigsten Signalen gegeben. Zusätzlich zu den gezeigten ROS-Nodes wird in dem Launchfile des Packages noch ein Kameratreiber und ein image_proc-Node gestartet, der das entzerrte Kamerabild zur Verfügung stellt, wenn eine entsprechende Kalibrierungsdatei zur Verfügung steht. Der verwendete Kameratreiber-Node libuvc_camera liefert von den für ROS verfügbaren Treibern die beste Auflösung (960x544) für die verwendete Kamera.

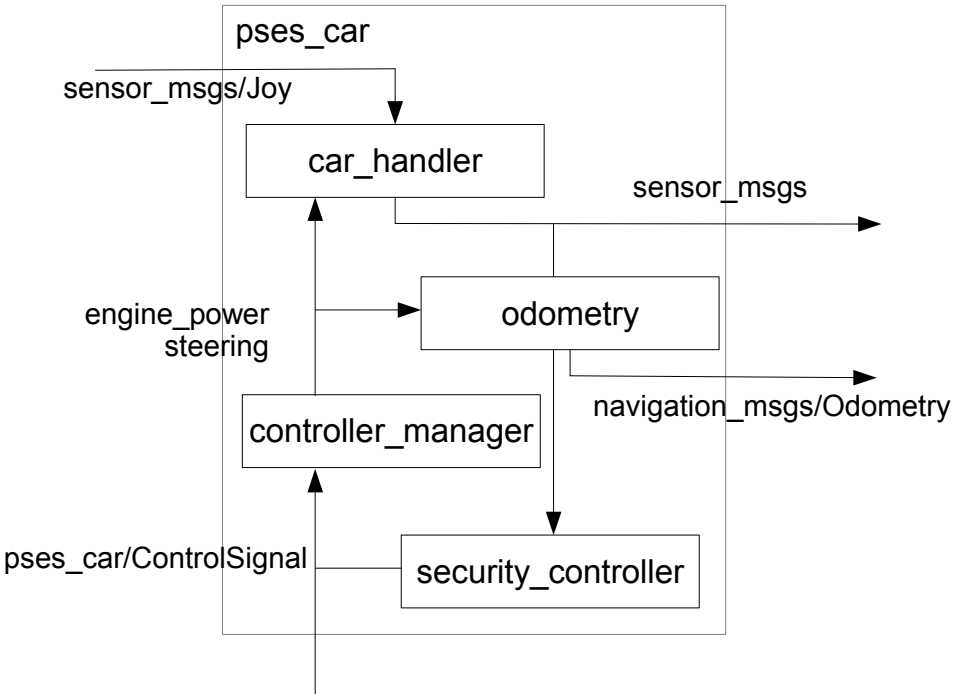

Abbildung 3.1: Package-Übersicht von PSES_Car mit den wichtigsten eingehenden und ausgehenden Signalen sowie den enthaltenen Nodes

3.1 car_handler

Aufgabe

Der car_handler dient als Interface zur Hardware des Fahrzeugs. Es werden alle verfügbaren Sensoren ausgewertet und die, teilweise gefilterten, Werte werden über entsprechende Topics veröffentlicht. Der USB-Port für die Kamera wird geöffnet. Außerdem sendet der car_handler Steuerbefehle, die mittels ROS-Topics empfangen wurden, weiter an die Elektromotoren des Fahrzeugs.

Funktionsweise

Die Sensorwerte des Linien-, des Hall- sowie des Spannungssensors für die Batteriespannung werden ungefiltert veröffentlicht. Die Sensorwerte der drei Ultraschallsensoren werden auf Werte zwischen 2,2 und 0,05 Meter begrenzt, um Rauschen und unplausible Werte sofort zu verwerfen. Anschließend werden diese Werte ebenfalls veröffentlicht.

Beim ersten Aufrufen des car_handler wird der USB-Port für die USB-Kamera geöffnet, sodass eine Kommunikation der zuständigen Module mit der Kamera direkt möglich wird.

Die empfangenen Steuersignale für die Elektromotoren werden weitergegeben. Hierbei wird der Wert des Lenksignals um zwei inkrementiert um den vorhandenen Lenkfehler bei gerader Fahrt zu korrigieren.

Der car_handler ermöglicht es außerdem den RealTime-Kernel des Micro-Controllers zu nutzen.

ROS-Interface

Subscribed Topics:

`/engine_power` (std_msgs/Int32)
: Signal für die gewünschte Geschwindigkeit des Fahrzeugs.

`/steering` (std_msgs/Int32)
: Stellsignal für die Lenkung des Fahrzeugs.

Published Topics:

`/front_us` (sensor_msgs/Range)
: Sensorwerte des frontalen Ultraschallsensors.

`/right_us` (sensor_msgs/Range)
: Sensorwerte des rechten Ultraschallsensors.

`/left_us` (sensor_msgs/Range)
: Sensorwerte des linken Ultraschallsensors.

`/line_sensor` (std_msgs/Char)
: Sensorwerte des Liniensensors.

`/battery` (std_msgs/Int32)
: Sensorwerte des Spannungssensors für die Batteriespannung.

`/hall_sensor` (std_msgs/UInt32)
: Sensorwerte des Hall-Sensors.

3.2 controller_manager

Aufgabe

Der controller_manager verwaltet alle eingehenden Steuerungsbefehle und veröffentlicht anschließend das am höchsten Priorisierte.

Funktionsweise

Der controller_manager verarbeitet alle eingehenden priorisierten Steuersignale und wählt dabei das mit der höchsten Priorität aus. Zu beachten ist hierbei, dass die niedrigste Priorität bei 100 liegt und höhere Prioritäten einem kleineren Zahlenwert als 100 entsprechen. Steuerbefehle mit einer gleichen oder höheren Priorität als vorhergehende werden sofort veröffentlicht. Befehle mit einer niedrigeren Priorität werden frühestens 10 Sekunden nach Veröffentlichung höher priorisierter Werte verarbeitet. Es werden zu jedem Zeitpunkt nur aktuell vorliegende Steuerbefehle verarbeitet. Veraltete werden verworfen.

Des Weiteren ist zu beachten, dass sich der securitycontroller durch das Senden von steering und / oder engine_power = -1000 beim controller_manager abmeldet und die Priorität hierdurch auf 1000 gesetzt wird. Jeder folgende Befehl ist somit höher priorisiert. Diese Abmeldefunktion kann auch von anderen Nodes verwendet werden. Die Priorität wird allerdings nur auf 1000 gesetzt, wenn kein höher priorisiertes Signal anliegt als das abzumeldende.

Falls kein Steuerbefehl sendender Node vorhanden ist, bleibt das Fahrzeug automatisch stehen.

ROS-Interface

Subscribed Topics:

`/control_signal` (pses_car/controlSignal)
: Priorisiertes Kontrollsignal, welches die Steuerungsbefehle für die Lenkung und die Motorgeschwindigkeit beinhaltet.

Published Topics:

`/engine_power` (std_msgs/Int32)
: Signal für die gewünschte Geschwindigkeit des Fahrzeugs.

`/steering` (std_msgs/Int32)
: Stellsignal für die Lenkung des Fahrzeugs.

3.3 odometry

Aufgabe

Die Odometrie ist für die Lokalisierung des Fahrzeugs zuständig. Dazu werden neben der klassischen Odometrie auch die Positionen gesichteter, bekannter ArUco-Marker hinzugezogen.

Funktionsweise

Das Package odometry ist eine Kombination zweier Techniken zur Selbstlokalisierung: klassische Odometrie und die Positionskorrektur anhand bekannter ArUco-Marker. Zum Startzeitpunkt wird davon ausgegangen, dass sich das Fahrzeug am Koordinatenursprung befindet. Um nun eine Lokalisierung durch klassische Odometrie durchführen zu können, werden die Steuerbefehle des Fahrzeugs (engine_power und steering) emp-

fangen und mithilfe der letzten geschätzen Position des Fahrzeugs eine neue Position geschätzt.

Da die klassische Odometrie in jedem Zyklus einen kumulativen Fehler erzeugt, ist es notwendig, regelmäßige Positionskorrekturen durchzuführen. Dazu werden auf dem Testgelände ArUco-Marker verteilt und deren Positionen, Orientierungen und Ids über den Parameter `markers` an die Odometrie übermittelt. Werden nun einer oder mehrere der Marker vom marker_publisher erkannt, so wird zunächst für jeden dieser Marker berechnet, wo das Fahrzeug sein müsste, um den Marker an der entsprechenden Stelle, mit der entsprechenden Ausrichtung sehen zu können.

Da die veröffentlichten Markerpositionen und -orientierungen gerade bei besonders flachen oder besonders Steilen Winkeln zwischen orthogonalen Achse des Markers und der Mittelachse des Fahrzeugs sehr ungenau werden können, stimmen die errechneten Positionen normalerweise nicht überein. Um dennoch hohe Genauigkeit zu erreichen, werden die errechneten Positionen und Orientierungen gemittelt und stellen nun eine neue Theorie über den Aufenthaltsort des Fahrzeugs dar. Damit das Umherspringen der veröffentlichten Markerpositionen und -orientierungen nicht in einem Umherspringen der geschätzten Fahrzeugposition und -orientierung resultiert, werden die Werte nicht direkt übernommen, sondern fließen zu 5 % in diese ein.

ROS-Interface

Subscribed Topics:

`/engine_power` (std_msgs/Int32)
Signal für die gewünschte Geschwindigkeit des Fahrzeugs.

`/steering` (std_msgs/Int32)
Stellsignal für die Lenkung des Fahrzeugs.

`/marker_publisher_node/markers` (marker_module/MarkerArray)
Positionen, Orienterungen und Ids der aktuell gesichteten ArUco-Marker.

`/initialpose` (geometry_msgs/PoseWithCovarianceStamped)
Wird hierüber eine Pose veröffentlicht, so wird diese umgehend als aktuelle Pose der Odometrie übernommen.

Published Topics:

`/odometry` (nav_msgs/Odometry)
Enthält die geschätzte Position und Orientierung des Fahrzeugs zum aktuellen Zeitpunkt.

`/costmap/footprint` (geometry_msgs/Polygon)
Footprint des Fahrzeugs.

`/known_global_markers` (geometry_msgs/PoseArray)
Die Positionen und Orientierungen aller, der Odometrie bekannten Marker.

Parameter:

`radius_right` (float)
Lenkradius bei einem Einschlag von 45. (default 1.0)

`radius_left` (float)
Lenkradius bei einem Einschlag von -45. (default 1.0)
`velocities` (float array)
Die Positionen und Orientierungen aller, der Odometrie bekannten Marker.
`markers` (complex)
Die Positionen und Orientierungen der bekannten Marker.

3.4 securitycontroller

Aufgabe

Der securitiycontroller soll dafür sorgen, dass es zu keinen Kollisionen des Fahrzeugs mit anderen Objekten kommt.

Funktionsweise

Der securitycontroller überwacht die Abstände zu anderen Objekten zu beiden Seiten und in frontaler Richtung mittels der Ultraschallsensoren. Hierzu werden die Sensorwerte mit einem Threshold (default: 0,2 m) verglichen. Bei einer drohenden seitlichen Kollision, d.h. auf einer Seite Abstand kleiner Threshold, wird versucht durch Gegenlenken mit maximalem Einschlag eine Kollision zu vermeiden. Sollte der Abstand auf beiden Seite zu gering sein, wird geradeaus gefahren. Bei einem zu geringen frontalen Abstand hält das Fahrzeug automatisch an.

Durch Senden von -1000 für einen der beiden Werte in control_signal meldet sich der securitycontroller beim controller_manager für die Steuerung dieses Wertes ab. Durch Senden von -1000 für beide Werte wird die Steuerung des Fahrzeugs durch den securitycontroller, bis zum erneuten Unterschreiten des Thresholds, komplett aufgehoben.

ROS-Interface

Subscribed Topics:

`/front_us` (sensor_msgs/Range)
Sensorwerte des frontalen Ultraschallsensors.
`/right_us` (sensor_msgs/Range)
Sensorwerte des rechten Ultraschallsensors.
`/left_us` (sensor_msgs/Range)
Sensorwerte des linken Ultraschallsensors.

Published Topics:

`/control_signal` (std_msgs/Int32)
Signal für die gewünschte Geschwindigkeit des Fahrzeugs.

4 Marker_Module

In diesem Kapitel werden die Aufgaben, die Funktionsweise sowie die verwendeten Topics vom Marker_Module näher erläutert. Um einen groben Überblick über die Zusammenhänge zwischen den wichtigsten Signalen und den Nodes des Packages zu geben, kann Abb. 4.1 herangezogen werden.

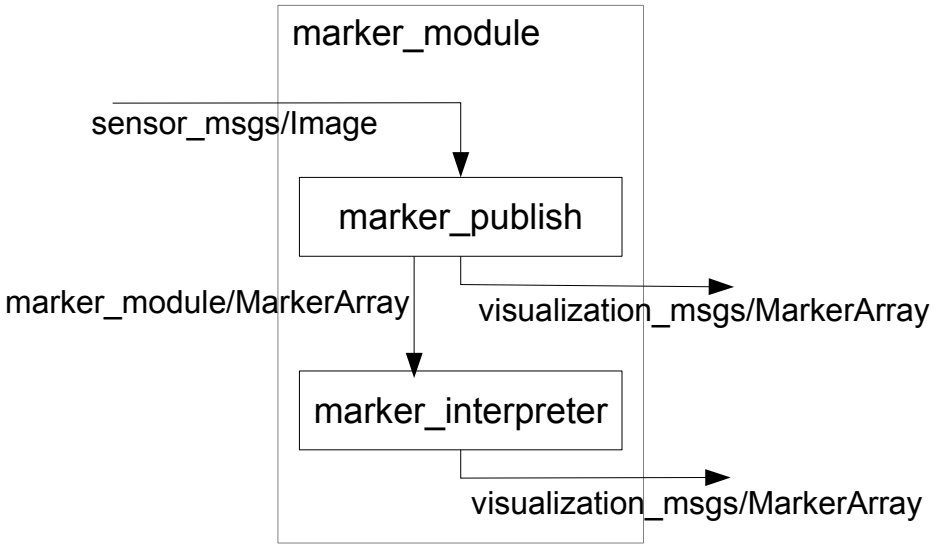

Abbildung 4.1: Package-Übersicht des Marker_Module mit den wichtigsten eingehenden und ausgehenden Signalen sowie den enthaltenen Nodes

4.1 marker_publish

Der marker_publish-Node ist eine angepasste Version des gleichnamigen Nodes aus dem aruco_ros-Package von pal-robotics. Er wurde für unseren Anwendungszweck angepasst, sodass die erkannten Marker direkt in RViz visualisiert werden können.

Aufgabe

Der marker_publish-Node wird benötigt, um aus dem entzerrten Kamerabild und den Kalibrierungsdaten der Kamera 3D-Postionen von Aruco-Markern zu berechnen. Die Position und die ID der Marker können für verschiedene Aufgaben verwendet werden, zum Beispiel können sie als Tore interpretiert werden oder aber als Landmarken zur Orientierung dienen.

Funktionsweise

Die Marker-Erkennung geschieht in zwei Schritten. Im ersten Schritt werden die Pixelpositionen der Marker im entzerrten Kamerabild mit Hilfe der Aruco-Bibliothek [GJMSMCMJ14] erkannt. Im zweiten Schritt wird aus der Pixelposition und den Kalibrierungsdaten die 3D-Position berechnet. Dafür wird die Größe des Markers und die Orientierung seiner Kanten zueinander verwendet.

ROS-Interface

Subscribed Topics:

`/camera/camera_info` (sensor_msgs/CamroseraInfo)
Information über das Kamerabild, wie zum Beispiel die Auflösung.

`/camera/image_rect` (sensor_msgs/Image)
Entzerrtes Bild der Kamera

Published Topics:

`/marker_publisher_node/markers` (marker_module/MarkerArray)
Positionen der erkannten Marker im Sichtfeld der Kamera.

Parameter:

`marker_size` (float)
Größe der Marker in Metern.

`image_is_rectified` (bool)
Gibt an, ob das Bild entzerrt ist oder nicht.

`reference_frame` (String)
Name des Koordinatensystems, in das die Marker umgerechnet werden.

`camera_frame` (String)
Name des Koordinatensystems der Kamera.

4.2 marker_interpreter

Aufgabe

Der marker_interpreter ist dafür zuständig, aus den veröffentlichten Positionen der ArUco-Marker der Tore Ziele für die Navigation zu generieren. Dies ist erforderlich, um die Aufgabe des Tore-Durchfahrens zu lösen.

Funktionsweise

Sobald neue Markerpositionen vom marker_publisher empfangen werden, werden daraus Zielposen berechnet. Hierbei wird davon ausgegangen, dass alle Marker mit Ids kleiner als 100 zu Toren gehören. Ein Tor setzt sich immer aus einem rechten und einem linken Marker zusammen, deren Zwischenraum das Tor darstellt. Die Id des linken Markers ist stets gerade und um eins geringer als die Id des rechten Markers. Dadurch lässt sich die Ausrichtung eines Tors eindeutig festlegen.

Eine Zielpose besteht immer aus einer Position und einer Orientierung. Die Position der Zielpose errechnet sich als exakter Mittelpunkt der, die beiden Markerpositionen verbindenden, Geraden. Die Orientierung der Zielpose orthogonal zu dieser Graden und der obigen Beschreibung entsprechend, so ausgerichtet, dass der Marker mit der kleineren Id links liegt.

Der marker_interpreter hat zwei Zustände: zum einen das Suchen und Setzen eines Ziels und zum anderen das Warten auf das Erreichen eines bereits gesetzten Ziels. Im ersten Zustand wird der marker_interpreter ein neues Ziel setzen, sobald mindestens ein Tor-Marker erkannt wurde. Wird von einem Tor nur ein Marker erkannt, so wird

die Position des zweiten Markers, unter der Annahme einer fixen Torbreite (Parameter default_goal_width) und der Ausrichtung des erkannten Markers in exakt entgegengesetzter Richtung zur Zielorientierung, geschätzt. Konnten mehrere Zielposen errechnet werden, so wird die Pose des Tores mit der geringsten Distanz zum Fahrzeug als neues Ziel ausgewählt. Wurde ein Ziel gefunden, so wird es über das MoveBase-Interface gesendet. Der marker_interpreter geht nun in den zweiten Zustand über.

Im zweiten Zustand wartet der marker_interpreter auf die Nachricht des Erreichens des Ziels über das MoveBase-Interface. In diesem Zustand werden nur die veröffentlichten Positionen der Marker des aktuellen Tors betrachtet. Sollte festgestellt werden, dass die aktuell errechnete Position des Ziels von der ursprünglich gesetzten Zielposition um mehr als 5 cm abweicht, so wird das Ziel neu gesetzt.

ROS-Interface

Subscribed Topics:

`/marker_publisher_node/markers` (marker_module/MarkerArray)
Positionen der erkannten Marker im Sichtfeld der Kamera.

Published Topics:

`/visualization_marker_array` (visualization_msgs/MarkerArray)
Dient zur Visualisierung der geschätzten Markerpositionen.

Parameter:

`default_goal_width` (float)
Standardbreite der Tore in Metern. Wird gebraucht, um die Positionen nicht erkannter Marker abschätzen zu können. (default 0.5 m)

5 Navigation

Das Navigation-Package dient zur Fahrzeugnavigation. Dabei wird unterschieden zwischen der Navigation mittels einer Karte (move_base_controller) und lediglich das Erreichen eines Punktes ohne Berücksichtigung der Umgebung (simple_move_base_controller). Es werden in beiden Navigationsarten Dubins Pfade zur lokalen Routenberechnung verwendet, welche im folgenden Kapitel näher erläutert werden. Die Package-Struktur mit den wichtigsten eingehenden und ausgehenden Signalen ist in Abb. 5.1 visualisiert.

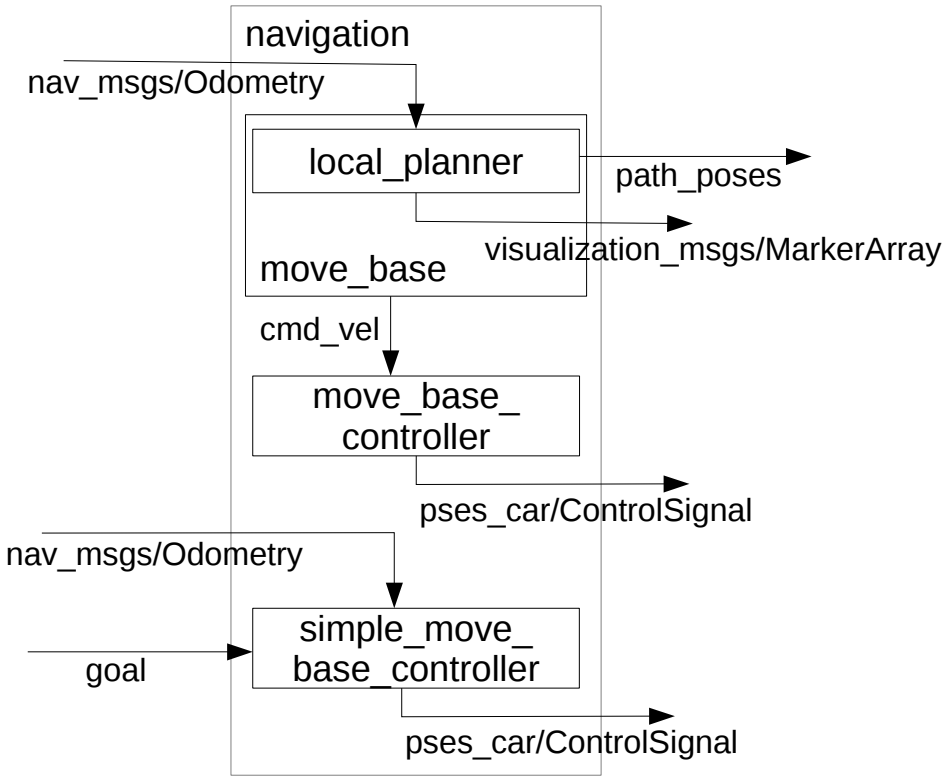

Abbildung 5.1: Package Übersicht des Navigation-Package mit den wichtigsten eingehenden und ausgehenden Signalen sowie den enthaltenen Nodes

5.1 Dubins Pfade

Um eine Routenplanung für Fahrzeuge, wie in diesem Seminar verwendet, zu realisieren, können Dubins Pfade herangezogen werden. Bei diesen Pfaden wird davon ausgegangen, dass es einen festen Lenkradius für Links- und Rechtskurven gibt, sowie das Fahrzeug nur in eine Richtung fahren kann.

Hier müssen zum Erreichen eines beliebigen Ziels ohne Berücksichtigung von Hindernissen lediglich drei feste Fahrmanöver aneinander gereiht werden. Diese sind das geradeaus Fahren, das Rechtskurven und das Linkskurven Fahren. Dabei wird immer mit einer Kurvenfahrt begonnen und geendet.

Zur Bestimmung dieser Pfade werden jeweils ein Kreis rechts und links am Ziel- bzw. Startpunkt eingezeichnet. Diese Kreise stellen die Kurvenfahrten dar. Nun werden zwischen den Kreisen Tangenten gezogen, welche die jeweiligen Strecken zum geradeaus Fahren darstellen. Bei der Auswahl der gültigen Tangenten ist zu achten, dass es zu keinen Richtungsänderungen des Fahrzeugs kommt. Eine Veranschaulichung dieser Konstruktion ist in Abbildung 5.1 gegeben. Hier ist die Planung für den Weg von Punkt M zu Punkt P dargestellt. [Kat13]

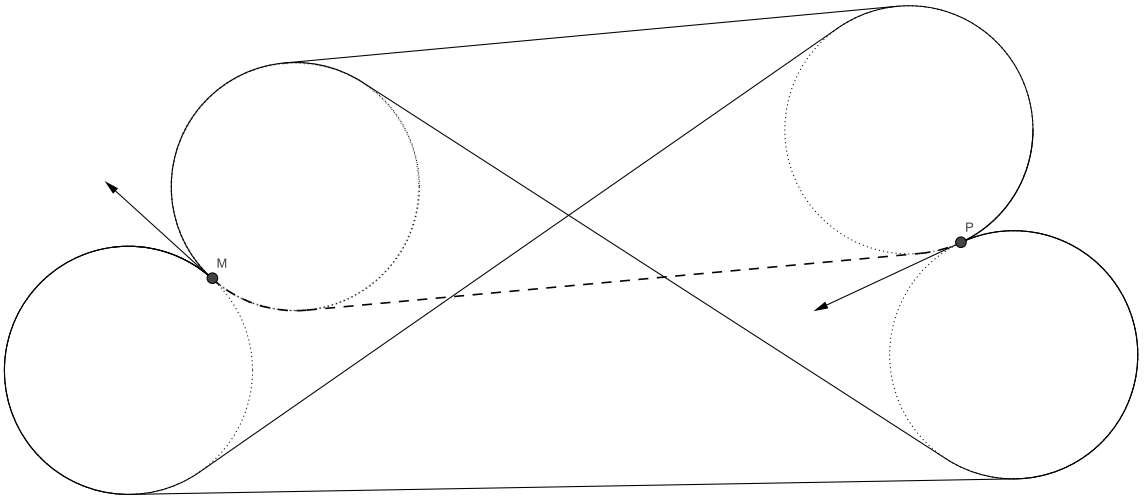

Abbildung 5.2: Dubins Pfade von Punkt M zu Punkt P [Kat13]

Auf weitere Sonderfälle, die bei Dubins Pfaden auftreten können, wird an dieser Stelle nicht weiter drauf eingegangen, da wir diese nicht betrachtet haben. Diese und das Einbinden von Richtungswechseln können aber als mögliche Erweiterungen unserer Implementierung dienen. Wie unsere beiden Implementierungen genauer aussehen wird im Folgenden betrachtet.

5.2 simple_move_base_controller

Aufgabe

Der simple_move_base_controller berechnet die nötigen Steuerbefehle und führt diese aus, um ein über ein Topic eingelesenes Ziel zu erreichen. Dabei wird allerdings nicht berücksichtigt, ob Hindernisse auf dem Pfad vorhanden sind. Die Steuerbefehle werden so berechnet, dass sie dem kürzesten Dubins Pfad folgen.

Funktionsweise

Wenn ein neues Ziel eingelesen wird, wird dieses intern gesetzt und die nötigen Steuerbefehle berechnet, um dieses zu erreichen. Da der Pfad anhand von Dubins Pfaden berechnet wird, werden zu Beginn die Mittelpunkte der vier Kreise bestimmt. Die Kreise werden jeweils durch ihre Orientierung (im oder gegen den Uhrzeigersinn) und ihre Position (Start oder Ziel) unterschieden.

Im Anschluss werden für jedes Kreispaar von Start- und Zielkreisen die Tangenten berechnet und anhand dieser mögliche Steuerbefehle erzeugt. Insgesamt gibt es zwischen zwei Kreisen vier mögliche Tangenten. Zwei parallele Tangenten (blau), die parallel zur Geraden zwischen den beiden Mittelpunkten sind, und zwei (rot), die sich schneiden (Abb. 5.2).

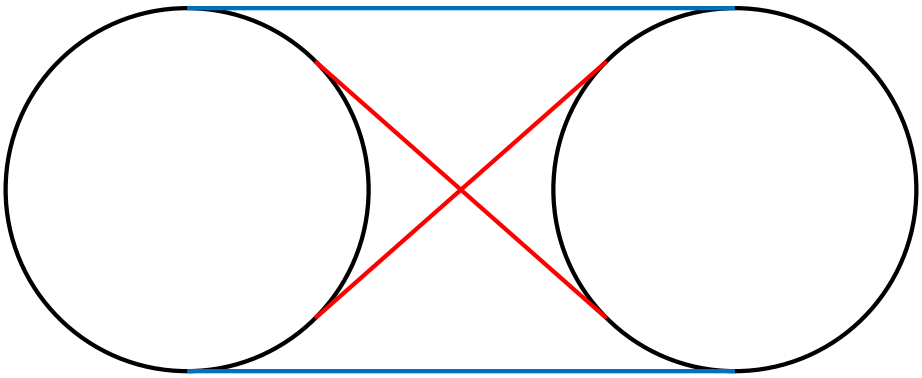

Abbildung 5.3: Mögliche Tangenten zwischen zwei Kreisen

Für die Berechnung müssen bei gleicher Orientierung der Start- und Zielkreise lediglich die parallelen Tangenten berechnet werden. Bei unterschiedlicher Orientierung sind nur die sich schneidenden Tangenten möglich. Dies liegt daran, dass in den anderen Fällen die Fahrzeugorientierung am Zielpunkt nicht eingehalten werden kann.

Anhand der Kreise und Tangenten werden für jedes Kreispaar Steuerbefehle erstellt. Diese bestehen aus einer Kurvenfahrt, gefolgt von einer geradeaus Fahrt und abschließend wieder mit einer Kurvenfahrt. Für jeden Steuerbefehl, wird die Zeit berechnet, die er anliegen muss.

Nach jeder Berechnung von neuen Steuerbefehlen wird die Gesamtzeit bestimmt und mit der bisher kürzesten Dauer verglichen. Sollte die neue Gesamtzeit kürzer sein, so werden die alten Befehle verworfen und die neuen angenommen. Dadurch wird am Ende, wenn alle vier Kreispaar Kombinationen abgearbeitet wurden, die Route mit der kürzesten Dauer angenommen.

Damit sind die Steuerbefehle theoretisch erstellt. Da der Servomotor in der Realität nicht genau genug ist, müssen noch zusätzliche Befehle eingefügt werden. Diese führen zu einem Übersteuern, wodurch die Ungenauigkeiten teilweise kompensiert werden können. Damit sind nun die Steuerbefehle erstellt.

Um die Steuerbefehle abzuarbeiten wird ein Timer verwendet. Dieser führt die zuvor berechneten Steuerbefehle aus, in dem er diese an den controller_manager sendet. Ist die Zeit eines Befehls abgelaufen wird automatisch der nächste Befehl ausgeführt, sofern noch weitere vorhanden sind. Wenn kein weiterer Befehl mehr vorhanden ist, wird das Fahrzeug gestoppt, weil das Ziel erreicht ist.

Zusätzlich wird regelmäßig überprüft, ob die Fahrzeugposition mit der Zielposition und einer Toleranz von 5 cm übereinstimmt. Sollte dies der Fall sein, so wird das Fahrzeug gestoppt.

ROS-Interface

Subscribed Topics:

`/odometry` (nav_msgs/Odometry)
Sensorwerte des rechten Ultraschallsensors.

`/move_base_simple/goal` (geometry_msgs/PoseStamped)
Neue Zielposition.

Published Topics:

`/control_signal` (pses_car/controlSignal)
Priorisiertes Kontrollsignal, welches die Steuerungsbefehle für die Lenkung und die Motorgeschwindigkeit beinhaltet.

`/visualization_marker_array` (visualization_msgs/MarkerArray)
Dient zur Visualisierung der Routenplanung.

Parameter:

`radius_left` (float)
Wert für den linken Kurvenradius.

`radius_right` (float)
Wert für den rechten Kurvenradius.

`velocities` (std/vector<double>)
Vector mit Geschwindigkeitswerten.

5.3 local_planner

Aufgabe

Der local_planner implementiert das BaseLocalPlanner Interface [bas] aus dem nav_core. Dabei dient dieser wie der simple_move_base_controller zur Berechnung einer lokalen Route. Diese wird allerdings nicht von ihm in Form von Steuersignalen ausgeführt, sondern an move_base weitergeleitet.

Die Verwendung erfolgt im Zusammenhang mit dem move_base-Node [mov] des nav_core-Package [nav]. Der move_base-Node verwendet zusätzlich eine lokale und globale cost_map [cos], einen map_server [map] und einen global_planner [glo] aus dem nav_core-Package. Dadurch ist eine Navigation mittels Karte möglich.

Funktionsweise

Der local_planner bekommt einen Plan als Vector übergeben. Dieser muss intern vom Typ geometry_msgs/PoseStamped auf tf/Transform konvertiert werden. Dabei wird die Orientierung der einzelnen Positionen so gesetzt, dass sie jeweils auf die folge Position zeigt. Lediglich die Orientierung der letzten Position wird beibehalten. Dies ist erforderlich, da in move_base von einem holonomen Roboter ausgegangen wird, dessen Orientierung dementsprechend nicht dauerhaft in Fahrtrichtung zeigen muss.

Das Abarbeiten des Planes folgt dem Prinzip, dass kleinere Abschnitte geplant werden und nach Abarbeitung dieser der nächste Abschnitt geplant wird. So wird der Gesamtplan nach und nach abgearbeitet.

Bei der Berechnung der Befehle wird zu Beginn die zum Fahrzeug am nächstliegendste Position im Plan gesucht. Sollte diese das Ziel sein, so werden alle Steuerbefehle auf null gesetzt. Ansonsten werden neue Befehle berechnet, falls keine Befehle mehr anstehen oder die vermutliche Position nach Abarbeitung der restlichen Befehle von der geplanten Position zu weit abweicht.

Sollte dies der Fall sein, wird die Position im Plan ermittelt, die eine vorgegebene Distanz entfernt ist. Dabei werden jeweils die Entfernungen zwischen zwei folge Positionen im Plan aufsummiert, bis die vorgegebene Distanz erreicht ist. Sollte von der ermittelten Position innerhalb der vorgegebenen Distanz das Ziel erreicht werden, so wird die ermittelte Position verworfen und das Ziel direkt angefahren.

Nachdem die nächste Zielposition feststeht, werden wie beim simple_move_base_controller die neuen Befehle berechnet und die Zeit vom setzen des ersten Befehls gespeichert. Werden keine neuen Befehle berechnet, sondern alte abgearbeitet, so werden sie nach Ablauf ihrer Zeit gelöscht, sodass der nächste Befehl abgearbeitet werden kann. Anschließend wird der Zeitstempel des zuletzt gesetzten Befehls aktualisiert.

Da die Befehle im move_base Node in Linear- und Winkelgeschwindigkeit angegeben werden, werden am Ende jeder Befehlsberechnung die vom simple_move_base_controller bekannten Befehle in diese Form transformiert.

ROS-Interface

Subscribed Topics:

`/odometry` (nav_msgs/Odometry)
Sensorwerte des rechten Ultraschallsensors.

Published Topics:

`/path_poses` (geometry_msgs/PoseArray)
Dient zur Visualisierung des übergebenen Plans.

`/visualization_marker_array` (visualization_msgs/MarkerArray)
Dient zur Visualisierung der Routenplanung.

Parameter:

`radius_left` (float)
Wert für den linken Kurvenradius.

`radius_right` (float)
Wert für den rechten Kurvenradius.

`velocities` (std/vector<double>)
Vector mit Geschwindigkeitswerten.

5.4 move_base_controller

Aufgabe

Um die Wiederverwendbarkeit zu gewährleisten wurde die Form der Steuerbefehle des move_base Nodes beibehalten. Um diese jedoch zu verwenden, müssen sie entsprechend transformiert werden.

Funktionsweise

Die Steuerbefehle des move_base Nodes bestehen aus einer Linear- und Winkelgeschwindigkeit. Diese werden zum einen in die Fahrzeuggeschwindigkeit und zum anderen in den Lenkeinschlag umgerechnet. Abschließend wird der daraus bestehende Steuerbefehl gesendet.

ROS-Interface

Subscribed Topics:

`/cmd_vel` (geometry_msgs/Twist)
Steuerbefehl in Linear- und Winkelgeschwindigkeit.

Published Topics:

`/control_signal` (pses_car/controlSignal)
Priorisiertes Kontrollsignal, welches die Steuerungsbefehle für die Lenkung und die Motorgeschwindigkeit beinhaltet.

Parameter:

`radius_left` (float)
Wert für den linken Kurvenradius.
`radius_right` (float)
Wert für den rechten Kurvenradius.
`velocities` (std/vector<double>)
Vector mit Geschwindigkeitswerten.

6 Joystick_Control

Das Package Joystick_Control bietet die Möglichkeit, das Auto manuell zu steuern. Es verwendet dazu den joystick_controller-Node.

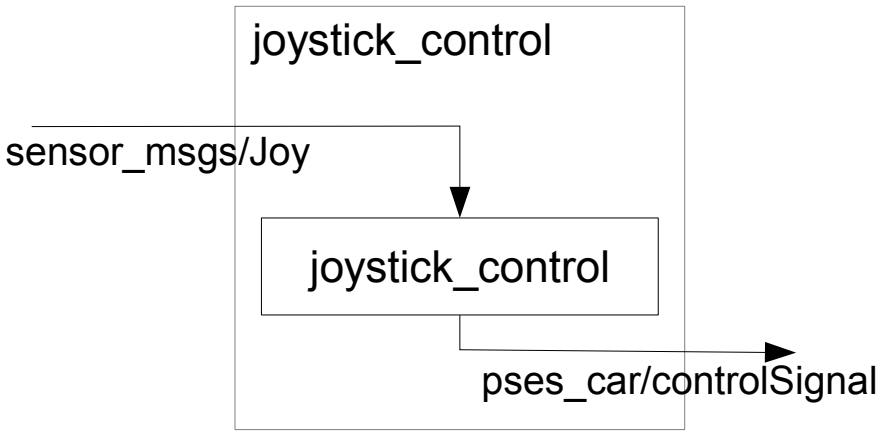

Abbildung 6.1: Package-Übersicht des Joystick_Controller mit eingehenden und ausgehenden Signalen sowie dem enthaltenen Node joystick_control

Aufgabe

Der joystick_controller-Node übersetzt die Benutzereingaben in Steuerbefehle für das PSES_Car Package. Für unsere Anforderungen genügte es, die drei Bewegungsachsen in Motorgeschwindigkeit und Lenkung zu übersetzen. An den meisten Joysticks oder Controllern steht auch noch eine Vielzahl an Knöpfen zur Verfügung. Um durch diese Aktionen auszulösen, muss dieser Node entsprechend erweitert werden.

Funktionsweise

Der joystick_controller-Node reagiert direkt auf jede über das joy-Topic gesendete Nachricht und übersetzt diese in einen Steuerbefehl. Die X-Achse wird dabei in die Lenkung übersetzt. Die Y- und Z-Achse werden kombiniert, um die Motorgeschwindigkeit zu berechnen. Die Z-Achse regelt hierbei den maximalen Wert, der zwischen (-)2 und (-)10 liegen kann, während die Y-Achse den tatsächlichen Wert bestimmt.

ROS-Interface

Subscribed Topics:

`/joy` (sensor_msgs/Joy)

Eingangssignale des angeschlossenen Joysticks oder Controllers.

Published Topics:

`/control_signal` (pses_car/controlSignal)

Priorisiertes Kontrollsignal, welches die Steuerungsbefehle für die Lenkung und die Motorgeschwindigkeit beinhaltet.

7 US_Controller

In diesem Kapitel werden die Aufgaben, die Funktionsweise und die verwendeten Topics des US_Controllers erläutert. Eine Übersicht des Packages ist in Abb. 7.1 gegeben. Hier sind die Zusammenhänge sowohl der eingehenden als auch ausgehenden Signale und dem im Package enthaltenen Node gezeigt.

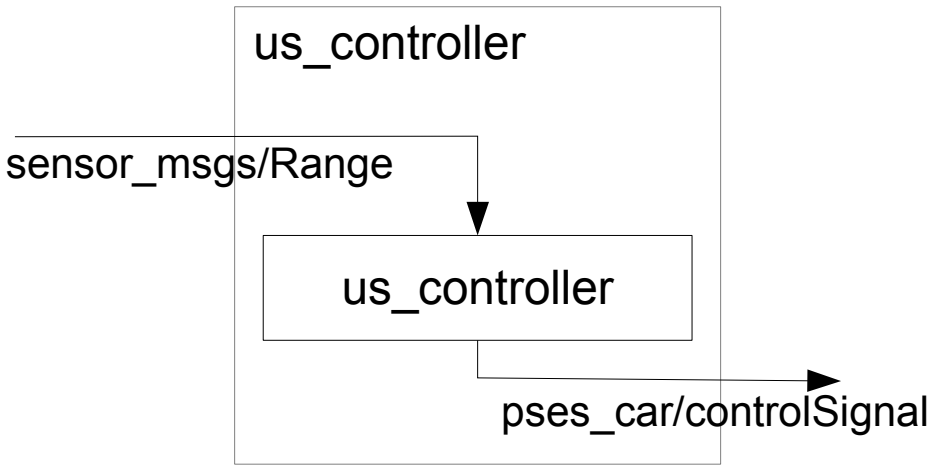

Abbildung 7.1: Package-Übersicht des US_Controller mit eingehenden und ausgehenden Signalen sowie dem enthaltenen Node us_controller

Aufgabe

Der US_Controller dient dazu einen vorgegebenen Abstand zur Seite hin einzuhalten. Dabei kann entweder der Abstand zur rechten oder zur linken Seite als Referenzwert angegeben werden. Anschließend werden entsprechende Steuerbefehle erstellt, um den Abstand zur Seite möglichst konstant zu halten. Der Abstand wird mittels der seitlichen Ultraschallsensoren ermittelt.

Funktionsweise

Der Referenzwert für den einzuhaltenden Abstand kann innerhalb des default-Konstruktors eingestellt werden. Dabei werden Werte größer null als rechts und Werte kleiner null als links vom Auto interpretiert. Hierbei ist zu berücksichtigen, dass der Abstand jeweils von den Außenkanten des Autos in Zentimeter angegeben werden muss.

Zur Regelung des Abstands wird ein PD-Regler verwendet. Auf einen I-Anteil konnte verzichtet werden, da dieser bereits auf der Strecke vorhanden ist. Die Regelparameter sind intern einmal `kp` für die Gleichanteilsverstärkung und zum anderen `kd` für die Differentialverstärkung. Die Variable `dt` muss immer gleich mit der Aufrufperiode des Reglers sein, da diese für die Berechnung des Differentialanteils benötigt wird. Sie gibt die Zeitdifferenz zwischen den Messwerten an und wird in ps gemessen.

Als Eingangsgröße des Reglers dient der Abstand zu der geregelten Seite. Dieser wird über die Ultraschallsensoren ermittelt. Die Ausgangsgröße der Regelung ist der Lenkeinschlag, welcher Werte zwischen -50 und 50 annehmen kann.

Abschließend von jedem Regelvorgang wird ein `controlSignal` erstellt und gesendet, welches die Regelausgangsgröße als `steering`-Wert und die vorab eingestellte Geschwindigkeit und Signalpriorität beinhaltet.

ROS-Interface

Subscribed Topics:

`/right_us` (sensor_msgs/Range)
Sensorwerte des rechten Ultraschallsensors.

`/left_us` (sensor_msgs/Range)
Sensorwerte des linken Ultraschallsensors.

Published Topics:

`/control_signal` (pses_car/controlSignal)
Priorisiertes Kontrollsignal, welches die Steuerungsbefehle für die Lenkung und die Motorgeschwindigkeit beinhaltet.

8 Einsatz der Packages

Um die in Abschnitt 1.1 vorgestellten Aufgaben zu erfüllen, werden die erstellten Packages entsprechend kombiniert. Grundsätzlich muss für jede Aufgabe das PSES_Car-Package gestartet werden. Welche Packages ansonsten nötig sind und wie diese zusammen arbeiten wird in den folgenden Abschnitten erklärt. Für jede der drei Aufgaben haben wir auch ein Skript geschrieben, mit dem alle Packages gestartet werden können. Der Name des Skripts ist jeweils in Klammern aufgeführt.

- **An der Wand entlang fahren**(start_wall_distance_regulation.sh)

  Um das Auto durch den Ultraschallsensor geregelt an einer geraden Wand entlang fahren zu lassen. Dazu muss zu dem PSES_Car-Package noch der US_Controller gestartet werden. Dieser bekommt als Eingabe die Werte der Ultraschallsensoren und liefert als Ausgabe ein Kontrollsignal.

- **Tore durchfahren**(start_goal_finder.sh)

  Damit Tore erkannt und durchfahren werden können, muss das Marker_Module-Package und aus dem Navigation-Package der simple_move_base_controller-Node gestartet werden. Das Marker_Module-Package analysiert das Kamerabild und gibt eine Zielposition aus. Der simple_move_base_controller-Node berechnet aus dieser Zielposition und der Odometrie ein Kontrollsignal, dass er an das PSES_Car-Package schickt.

- **Navigation auf der Karte**(start_map_planning.sh)

  Um auf einer Karte ein Ziel einzeichnen und dieses anfahren zu können, muss das Marker_Module-Package und das Navigation-Package gestartet werden. Außerdem wird RViz benötigt, um manuell ein Ziel auf der Karte zu setzen. Für dieses Ziel wird im Navigation-Package eine Route auf der Karte geplant. Das Marker_Module-Package findet in dem Kamerabild Markerpositionen, die vom PSES_Car-Package zur Korrektur der Odometry verwendet werden. Die Odometry wird vom Navigation-Package zusammen mit der geplanten Route verwendet, um ein Kontrollsignal an das PSES_Car-Package zu schicken.

9 Zusammenfassung und Fazit

Das Pflichtprogramm an der Wand entlang fahren und Tore aus ArUco-Markern durchfahren konnte erfüllt werden. Beim Wand entlang fahren ist aufgefallen, dass ein Ultraschallsensor je Seite für genaue Messungen zu wenig ist. Mit einem weiteren pro Seite ließe sich die relative Lage des Autos zur Wand bestimmen und somit die Abstandsmessung vermutlich verbessern.

Mit einer gut kalibrierten Kamera ist es auch gelungen die Tore durch ArUco-Marker zielsicher zu durchfahren. Hier war die Routenplanung und dessen präzise Abfahren eine Herausforderung. Erstere konnte mittels Routenplanung durch Dubins Pfade erreicht werden. Da es sich um eine allgemeine Planung handelt, konnte diese auch für weitere Aufgaben verwendet werden.

Beim Abfahren der berechneten Route stellte sich schnell heraus, dass der Hallsensor an unserem Auto keine plausiblen Werte sendet. Aus diesem Grund konnte nicht die zurückgelegte Strecke als Anhaltspunkt verwendet werden. Als Lösung wurden die Steuerbefehle anhand ihrer Dauer aktualisiert. Dies hat gegenüber dem Messen der zurückgelegten Strecke den Nachteil, dass eine schwankende Motorbatteriespannung zu Abweichungen führt, da die Geschwindigkeit nicht konstant bleibt.

Des Weiteren ist die Lenkung recht unpräzise, wodurch die Lenkung nach einer Kurvenfahrt nicht wieder in die Nullstellung gesetzt werden konnte. Diese Ungenauigkeit konnte durch gezieltes Übersteuern minimiert werden. Weitere Inertialsensorik, wodurch auch der aktuelle Lenkwinkel ausgelesen werden könnte, wären sehr vorteilhaft.

Weiter ist aufgefallen, dass die verwendete Kamera ein relativ kleines Sichtfeld besitzt und mit den ROS-Treibern nicht die maximal mögliche Auflösung erreicht. Das Problem mit dem geringen Sichtfeld wurde mit einer Weitwinkellinse gelöst. Eigentlich wollten wir keine Änderungen an unserer Versuchsplattform durchführen. Aber da dies kein großer Eingriff darstellt und eine enorme Verbesserung zur Folge hatte, sind wir hier als einzige Ausnahme von unserem eigentlichen Vorhaben abgewichen.

Unser selbst gesetztes Ziel, sich anhand einer Karte zu orientieren und einen geplanten Pfad ab zufahren, konnte ebenfalls realisiert werden. Hier konnte die Routenplanung von dem Tore Durchfahren wiederverwendet werden. Um die schon beschriebenen Ungenauigkeiten beim Abfahren zu korrigieren, wurden in die Karte feste Punkte von ArUco-Markern eingezeichnet, womit die Position des Autos während der Fahrt auf der Karte korrigiert werden konnte.

Insgesamt lässt sich sagen, dass wir alle Ziele erfüllen konnten, die wir uns Anfangs gesetzt haben. Wie oben genauer erwähnt, ließe sich das Ergebnis bereits durch hinzufügen weniger Sensoren deutlich verbessern.

Literatur

[bas] *nav_core::BaseLocalPlanner Class Reference*. – http://docs.ros.org/-api/nav_core/html/classnav__core_1_1BaseLocalPlanner.html

[cos] *costmap_2d*. – http://wiki.ros.org/costmap_2d?distro=indigo

[GJMSMCMJ14] Garrido-Jurado, S. ; Muñoz-Salinas, R. ; Madrid-Cuevas, F.J. ; Marín-Jiménez, M.J.: Automatic generation and detection of highly reliable fiducial markers under occlusion. In: *Pattern Recognition* 47 (2014), Nr. 6, S. 2280 – 2292. – ISSN 0031–3203

[glo] *global_planner*. – http://wiki.ros.org/global_planner?distro=indigo

[Kat13] Kathe, Florian: *Algorithmen der Bewegungsplanung für nicht-holonome Roboter*. 2013. – https://www.uni-koblenz-landau.de/de/koblenz/-fb4/ist/AGZoebel/Lehre/sommer2013/SeminarASidA/ausarbeitung-kathe

[map] *map_server*. – http://wiki.ros.org/map_server?distro=indigo

[mov] *move_base*. – http://wiki.ros.org/move_base?distro=indigo

[nav] *nav_core*. – http://wiki.ros.org/nav_core