ES-Projektseminar WS2016 Abschlussbericht

Team Mephobia
Projektseminar-Bericht eingereicht von
Jonas Gehrunger, Manishkumar Jain, Jan-Christoph Klie, Matthias Klimmek, Mario Wetzel
am 15. April 2017

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

Erklärung zum Projektseminar-Bericht

Hiermit versichern wir, das vorliegende Projektseminar-Bericht selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die wir aus fremden Quellen direkt oder indirekt übernommen habe, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Wir erklären uns damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 15. April 2017

(J. Gehrunger, M. Jain, J. Klie, M. Klimmek, M. Wetzel)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Das Projektseminars Echtzeitsysteme wird jedes Wintersemester vom Fachgebiet Echtzeitsysteme angeboten. Es richtet sich an Studenten der Fachrichtungen Informationssystemtechnik, Informatik und Elektro- und Informationstechnik, sowie Automatisierungstechnik und Mechatronik. Ziel ist die Umsetzung mehrerer Aufgaben auf einem autonom fahrenden Fahrzeug. Die Durchführung erfolgte in Gruppen von je 5 Studenten, die während des Semesters alle Aspekte der Projektarbeit kennen gelernt und umgesetzt haben.

Die zur Auswahl stehenden Themen waren:

1. Rundstrecke ohne Hindernisse
2. Rundstrecke mit Hindernissen
3. Parklücke finden und parallel einparken
4. Fahrbahnmarkierung erkennen und Spur halten
5. Autonomes Erkunden eines unbekannten Terrains

Die erste Aufgabe war für jedes Team als Pflichtaufgabe umzusetzen; von den restlichen vier Themen mussten noch zwei weitere ausgewählt werden. Es wurde relativ früh entschieden, auch die Rundstrecke mit Hindernissen zu bearbeiten, da erhofft und erwartet wurde, dass so der Ansatz aus der ersten Aufgabe benutzt werden kann. Als dritte Aufgabe wurde Fahrbahnerkennung/Spur halten gewählt.

2 Das Fahrzeug

In diesem Kapitel wird die Hardware und Software des Fahrzeuges beschrieben, auf Grundlage derer die verschiedenen Aufgaben gelöst wurden.

2.1 Hardware

Grundlage der Implementierung ist ein vierrädriges Auto-ähnliches Fahrzeug, das mit diversen Sensoren und einem on-board Computer ausgestattet ist. Zu den Sensoren gehören drei Ultraschallsensoren (vorne, links und rechts), eine Kinect-Kamera, ein Hall-Sensor am Hinterrad sowie eine IMU[1] bestehend aus einem 3D-Beschleunigungssensor und einem 3D-Magnetometer. Die Sensoren sind an einem Mikrocontroller angeschlossen, der eine Vorverarbeitung durchführt und die Daten an das eigentlich Gehirn des Autos, einen kompletten Mini-PC, weitergibt.

Im Weiteren werden die verschiedenen Sensoren detailliert vorgestellt. Die Weiterverarbeitung der Sensoren wird in Abschnitt 4 beschrieben.

2.1.1 Ultraschall

Jeder Ultraschallsensor besteht aus einem Paar von Sender und Empfänger, wobei der Sender in regelmäßigen Abständen Schallimpulse aussendet, die von der Umgebung des Autos reflektiert werden. Mithilfe der Zeitverzögerung bis zur Messung des ersten Echos lässt sich der Abstand zum nächstgelegenen Hindernis bestimmen. Der maximale gemessene Abstand ist dabei durch Beschränkung der Messdauer im Sensor gegeben. Trifft kein Echo innerhalb der festgelegten Zeit ein (beim verwendeten Sensor maximal auf 65ms oder 11m einstellbar), wird die Messung mit einem Abstandsergebnis von 0 Metern beendet.

In der Praxis sind die Messwerte des Sensors relativ stark schwankend, vor allem bedingt durch Objekte in der Umgebung mit unebenen Oberflächen (auf dem Rundkurs etwa Heizkörper) oder mit steilem Winkel zum Sensor selbst. Da die Stärke der Störungen mit steigendem Abstand zunehmimmt, beschränken wir die Messwerte in der Verarbeitung durch eine obere Grenze von 2 Metern.

Zusätzlich liefert der Sensor prinzipbedingt selbst keine Rückmeldung über die relative Position des Hindernisses, so dass etwa ein schiefer Winkel zur Wand zu auf den ersten Blick irreführenden Messwerten führen kann.

Benutzt werden die Ultraschallsensoren deswegen vor allem für die Bestimmung des tatsächlichen Wandabstandes zu Beginn der Wandschätzung mithilfe des Laserscans der Kinect.

[1] Internal Measurement Unit - Messeinheit zur Erfassung aller sechs kinematischen Freiheitsgrade

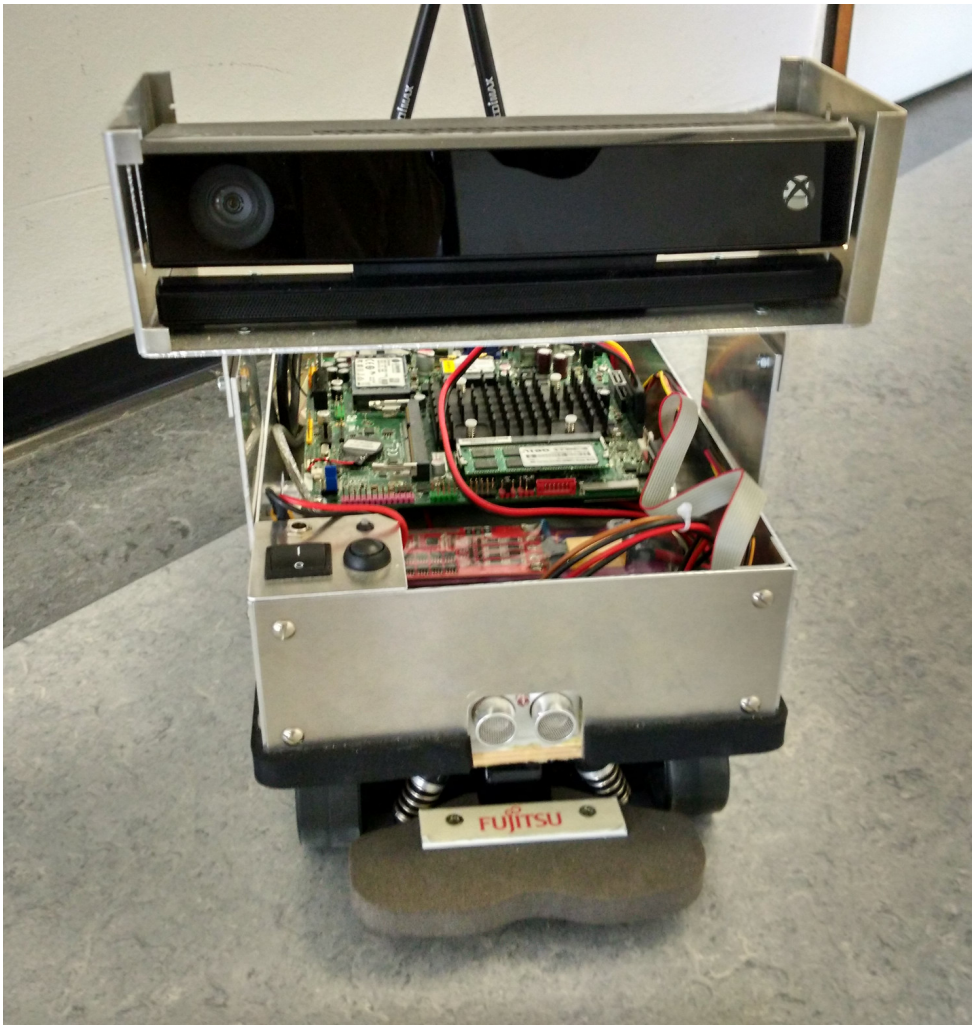

Abbildung 2.1: Das verwendete Auto im Projektseminar. Vorne sind Ultraschall und Kinect zu sehen, im Hintergrund die WLAN-Antennen.

2.1.2 Gyroskop/Accelerometer

Das Fahrzeug verfügt über einen MPU-9250A Beschleunigungssensor, der die aktuellen Drehraten- und Beschleunigungswerte in den drei Raumachsen misst. Die gemessenen Werte werden innerhalb der IMU bereits durch digitale Filter bearbeitet. Eine genauere Beschreibung dieses Sensors kann in [Pro, Abschnitt 1.3.3] gefunden werden.

Innerhalb der Implementierung wird das Gyroskop nicht direkt, sondern nach der Verarbeitung der Messwerte innerhalb von `pses_basis` als Teil der Fahrzeugodometrie verwendet.

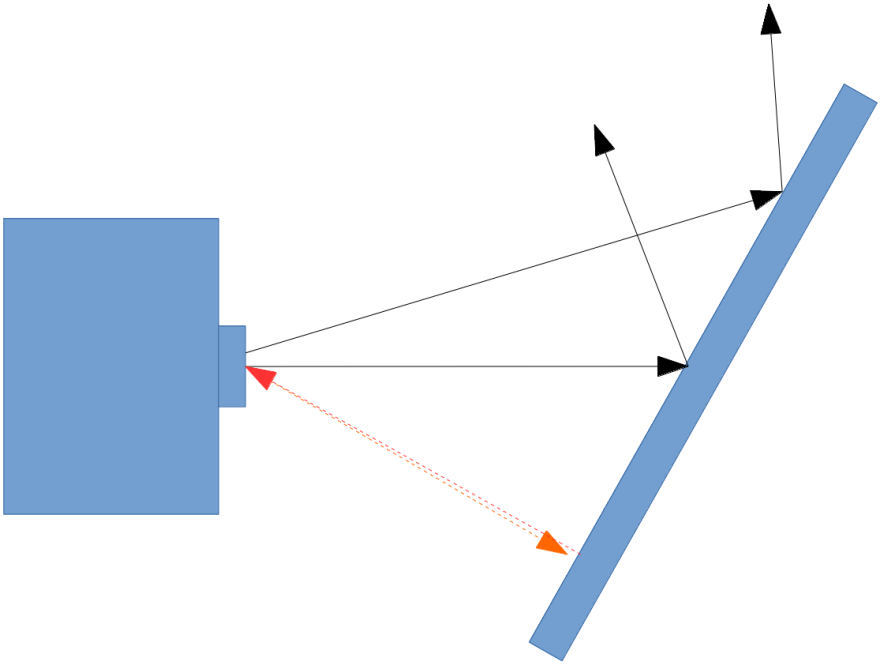

Abbildung 2.2: Beispiel für problematische Reflektion der Schallwellen, vor allem bei kleinen oder weit entfernten Strukturen führt dies zu hin- und herspringenden Messergebnissen

2.1.3 Odometrie

Für die Odometrie kommt der Hall-Sensor am Hinterrad des Fahrzeugs in Kombination mit der IMU zum Einsatz, woraus Position, Rotation und Geschwindigkeit des Fahrzeuges in Relation zum Startpunkt ermittelt werden.

2.1.4 Kinect

Die Kinect (Tiefenkamera) liefert Tiefeninformationen über den Bereich vor dem Fahrzeug durch Ableitung von Tiefeninformationen aus der Verformung der Punkte eines ausgestrahlten Punktemusters aus Infrarotlicht[2]. Die kinect-Bridge liefert die ermittelten Tiefeninformationen dabei als Graustufenbild und als 3D-Punktewolke.

Für die Auswertung des Tiefenbildes kommt das ROS-Package `depthimage_to_laserscan` zum Einsatz, das eine Pixelzeile des Tiefenbildes betrachtet und diese zu einem 180-Grad Laserscan projiziert. Besondere Aufmerksamkeit erfordert dabei die Neigung der Kinect, die durch abrupte Bewegung des Fahrzeuges verstellt werden und die, wie während der Testfahrten zu beobachten war, gerade durch den Blickwinkel zu Lampen an der Decke deutliche Auswirkungen auf die Intensität des auftretenden Rauschens haben kann.

[2] https://users.dickinson.edu/ jmac/selected-talks/kinect.pdf

2.2 Software

2.2.1 Robot Operating System

Das verwendete Meta-Betriebssystem ROS basiert auf der Verwendung eines Netzwerks aus Knoten (nodes), die untereinander auf bestimmten Kanälen (topics) Nachrichten (messages) versenden und empfangen. Dabei handelt es sich um eine asynchrone many-to-many-Kommunikation. Die verschiedenen Funktionalitäten sind in Paketen gruppiert, sodass die zur Ausführung notwendigen Funktionen einfach installiert werden können.

Die für ein Programm nötigen Knoten werden über sogenannte launch-Dateien in XML-Syntax mit den entsprechenden Parametern spezifiziert und können so gemeinsam gestartet werden. Neben den ausführbaren Knoten gibt es eine Vielzahl an Konfigurationsdateien, die zum Beispiel die Nachrichten definieren oder weitere Parameter enthalten.

2.2.2 Vorhandene Software

Auf dem PC des Autos läuft ein Lubuntu-Betriebssystem[3], auf dem ROS installiert ist. Dieses Semester wurde auch erstmals eine Simulation des Fahrzeugs bereitgestellt, sodass einfache Tests und erste Lösungsansätze auch ohne das Auto selbst durchgeführt werden konnten. In der Simulation wurden Messwerte für die vorhandenen Sensoren anhand einer Karte generiert.

Für die Grundfunktionalitäten des Fahrzeugs standen bereits ROS-Pakete zur Verfügung, sodass direkt mit den eigentlichen Aufgaben anfangen werden konnte.

Im Rahmen des Projektes wurden mehrere eigene Pakete erstellt, die jeweils in einem eigenen Git-Repository verwaltet werden. Somit entstand ein Paket für die Sensordatenverarbeitung (Filterung), sowie eines für jede der drei gewählten Aufgaben.

2.3 Gemessene Fahrzeugparameter

In der Theorie hat das Auto einen linearen Lenkbereich von -50 bis +50, sowie einen linearen Geschwindigkeitsbereich von -20 bis +20. Da es für spezielle Aufgaben wichtig ist, den genauen Einschlagwinkel oder die genaue Geschwindigkeit zu kennen, wurden die entsprechenden Werte gemessen. Dies ist insbesondere für die Konfiguration der Planer wichtig [4], da sie Minimal- und Maximalgeschwindigkeit benötigen und der ausgegebene Winkel in Steering Level umgerechnet werden muss.

[3] ein sparsames Linux System; in Version 14.04, siehe http://lubuntu.net/

[4] mehr dazu in Abschnitt 5

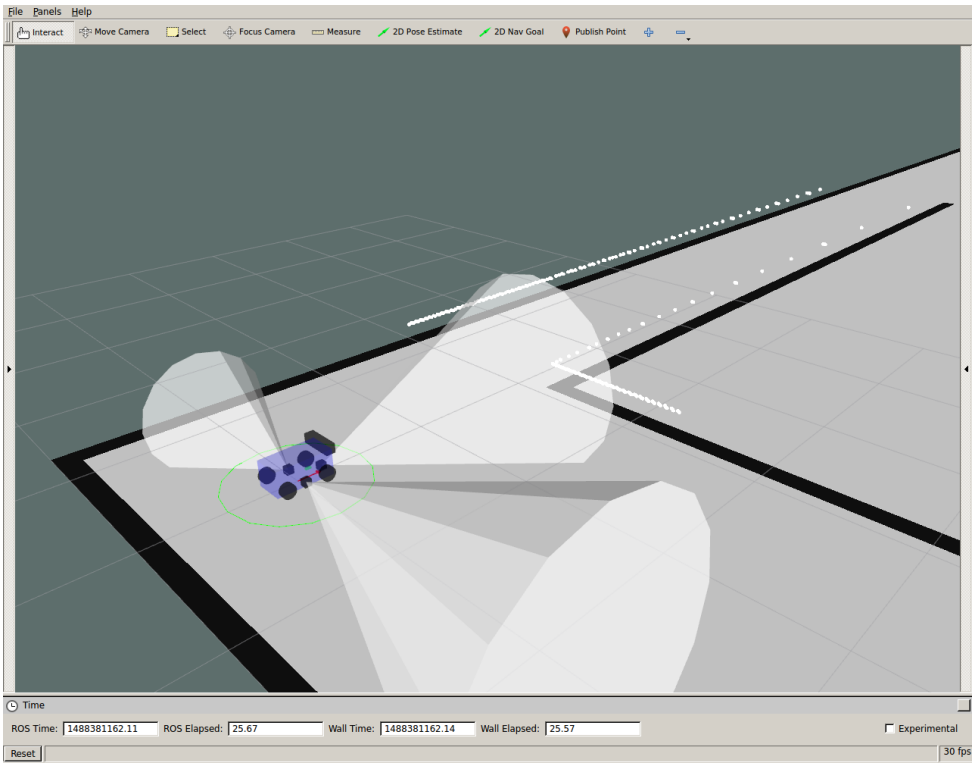

Abbildung 2.3: Man erkennt das Modell des Autos sowie die Sensorbereiche der drei Ultraschallsensoren. Die Kinect wird durch den hier weiß dargestellten Laserscan simuliert.

Tabelle 2.1: Lenkwinkel in Abhängigkeit von Steering Level

	5	**10**	**15**	**20**	**25**	**30**	**35**	**40**
+	2.5	5	9.5	12	14.5	18	23†	
-	4.5	8	11	14.5	17.5	21†	25	

†Hörbare Sättigung

Tabelle 2.2: Geschwindigkeit [m/s] in Abhängigkeit von Motor Level

L	5	6	7	8	9	10	11	12
v			1.0	1.1		1.4		1.6
	0.77	0.82			1.25		1.44	
	13	14	15	16	17	18	19	20
		0.6	1.7		2.0	2.1	2.2	2.3
	0.48			1.81				

3 Organisation und Herangehensweise

Im Weiteren wird beschrieben, wie die Arbeit um das Projektseminar organisiert wurde.

3.1 Teams

In diesem Durchgang des Projektseminars wurden erstmals Gruppen von 5 Leuten gebildet (vorher waren es 4). Da es schwierig ist, effektiv in dieser Größe gemeinsam an Themen zu arbeiten, wurde entschieden, dass zwei kleinere Subgruppen gebildet werden. Diese Gruppen trafen sich unabhängig voneinander. Jeden Freitag gab es dann ein gemeinsames Treffen mit allen Teammitgliedern, an dem der jeweilige Fortschritt besprochen wurde.

Die Aufteilung wurde so gezogen, dass eine Gruppe sich um die Hardware, vor allem Sensoren gekümmert hat. Ziel der Sensorgruppe war es, die Infrastruktur (z.B. möglichst gute Messwerte bereitstellen, Laserscans erweitern, Sensorpipeline anpassen) für die andere Gruppe zu liefern. Diese andere Gruppe hat sich dann konkret um die gewählten Aufgaben kümmern. Im Sensorenteam waren zwei Leute, im Aufgabenteam drei.

3.2 Planung

Es wurde früh entschieden, dass keine konkrete Zeitplanung (etwa Gantt) gemacht wird, da in anderen Projekten schlechte Erfahrungen damit gemacht wurde. Es sah schwierig aus, am Anfang des Projektes über den Zeitraum eines Semesters halbwegs genau die Bearbeitungszeiten einzelner Pakete abzuschätzen, daher wurde ein anderes Vorgehen benutzt.

Statt einer globalen Planung zu Beginn wurden die einzelnen größen Aufgaben (Epics) gesammelt, etwa Sensorverarbeitung und die drei Disziplinen. Diese wurden dann in kleinere Pakete aufgeteilt (Stories), die sich in der Zeit zwischen zwei Gruppentreffen erledigen lassen. An den Gruppentreffen selber wurde von jeden Mitglied vorgestellt, was seit dem letzten Treffen erreicht wurde, wo es Probleme gibt, wodurch sie durch andere, noch nicht erledigte Aufgaben blockiert sind und was in der nächsten Woche geplant ist. Die Planung wurde dann von allen besprochen, damit alle Abhängigkeiten erfüllt sind und schließlich umgesetzt. Zu diesem Zeitpunkt wurde auch abgeschätzt, wie lange einzelne Aufgaben dauern, wenn sie länger als eine Woche dauern.

3.3 Trello

Um die Aufgaben zu verwalten, wurde die Projektmanagementsoftware Trello eingesetzt. Große Aufgaben werden in Boards (Pinnwand) verwaltet, Unteraufgaben als Eintrag auf dem jeweiligen Board. Es wurden Boards für die drei Disziplinen, Projektma-

nagement und Sensorgruppe angelegt. Nach dem Runterbrechen von größerer Funktionalität in Teilaufgaben wurden die einzelnen Arbeitspakete in das jeweilige Board eingetragen, mit Enddatum versehen und den bearbeitenden Personen zugeordnet. Je nach Fortschritt der Aufgabe wird es in eine Spalte des Boards eingeordnet, etwa *Planned*, *In progress*, *Testing*, *Completed*. Ein Beispielboard ist in Fig. 3.1 dargestellt.

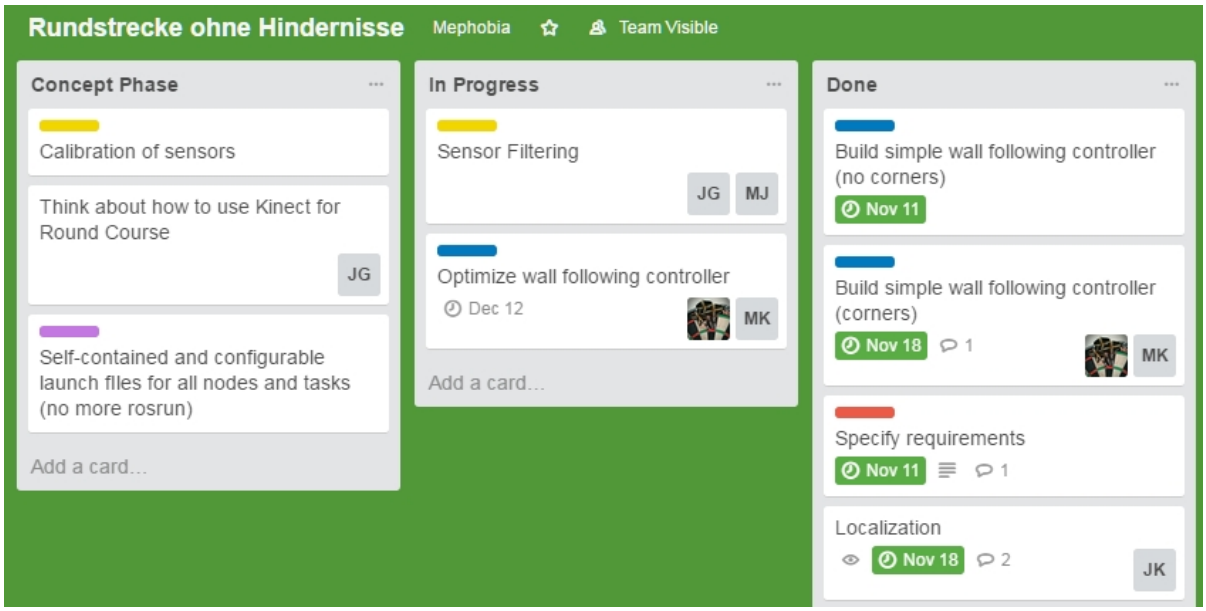

Abbildung 3.1: Board der Aufgabe 'Rundstrecke ohne Hindernisse'. Teilaufgaben werden links eingetragen und wandern dann nach rechts durch das Board. Die Markierungen zeichnen Aufgaben aus, die zusammengehören, etwa blau für Regelung.

3.4 Codeverwaltung

Das Programmieren diverser Artefakte mit mehreren Leuten, die gleichzeitig an dem gleichen Code arbeiten, ist ohne Quellcodeverwaltung aufwendig und fehleranfällig. Daher wurde sehr früh entschieden, dass Git und insbesondere Github benutzt wird. Es wurde eine Github-Gruppe gegründet; jedes ROS-Paket wurde in einem eigenen Repository verwaltet. Wenn mehrere Teammitglieder gleichzeitig an einem Paket arbeiten mussten, wurden Git-Branches benutzt und am Ende in den Masterbranch gemergt. Die Gruppe kann unter https://github.com/orgs/pses-mephobia/ gefunden werden.

4 Sensornode

Die Sensoren des Fahrzeuges werden im `mephobia_sensors_node` aufbereitet und an die anderen ROS-Nodes weitergeleitet. Dabei wird neben der reinen Filterung der Messwerte auch die Rate der weitergeleiteten Messwerte gegenüber der Rate der neu eintreffenden Daten reduziert. Dies führt dazu, dass kurze Störungen weniger ins Gewicht fallen und das Gesamtsystem am Ende stabiler läuft. Um die aufbereiteten Messwerte zu nutzen ist es dabei erforderlich, dass andere ROS-Nodes `mephobia_sensors_node` und nicht das PSES-Basis-Modul abonnieren, das direkt mit dem uc-Board des Fahrzeuges kommuniziert.

Diese Weiterleitung ermöglicht zusätzlich auch die Implementation einer Notbremse, die bei sehr kurzem Abstand zu einem Hindernis nach vorne nur noch stehenbleiben oder rückwärts fahren zulässt, um einen Zusammenstoß des Fahrzeuges zu verhindern.

Die Messwerte von IMU und Hall-Sensor werden bereits im `pses_basis`-Modul und deswegen innerhalb der Implementierung nicht aktiv gefiltert, sondern durch einen Reset ergänzt. Dabei wird der aktuell zurückgemeldete Winkel des Fahrzeuges auf Anforderung an das Sensornode zum neuen Referenzwinkel und entsprechend alle weiteren Messwerte verschoben.

Die Werte der Ultraschallsensoren werden jeweils durch einen eindimensionalen Kalman-Filter mit konstanter Kovarianzen-Matrix [GA02] gefiltert, der glattere Ergebnisse mit kürzerer Ansprechzeit liefert als der zunächst getesteter Filter mit gleitendem Durchschnitt. Weiterhin werden alle Messwerte unter dem Schwellenwert 1cm und über der Grenze 2m jeweils als 2m angenommen, da der Sensor wie im Hardware-Überblick beschrieben einen unendlichen Abstand durch 0m statt durch den maximal möglichen Abstand repräsentiert sowie Messwerte oberhalb von 2 Metern in der Praxis sehr stark verrauscht sind.

4.1 Kinect

Für die gleichzeitige Erkennung von Wänden und Hindernissen in der Aufgabe Rundkurs mit Hindernissen setzen wir zwei Laserscans ein, die jeweils um einen Offset verschoben sind. Wie im Überblick über die Hardware bereits beschrieben verwenden wir das `depthimage_to_laserscan`-Node, das jedoch von einem festen Streifen in der Mitte des Bildes ausgeht. Um unsere Anforderungen zu erfüllen wurde dieses Modul um weitere Parameter erweitert die eine einstellbare Verschiebung des betrachteten Streifens für jede Instanz von `depthimage_to_laserscan` ermöglicht.

Die Bestimmung des Offsets von der Mitte erfolgt dabei, indem eine Testfahrt durchgeführt und im Anschluss in der Aufzeichnung des Tiefenbildes bestimmt wird, welche Höhe die gesetzten Hindernisse maximal im Tiefenbild erreichen und auf welcher Höhe der Boden im Bild endet. Wichtig ist dabei ein Sicherheitsabstand, da bei verstellter Nei-

gung der Kinect sonst etwa der Boden als scheinbare parallel zur Front des Fahrzeuges liegende Wand erfasst werden kann.

Da es sich bei der Kinect um eine Infrarot-Tiefenkamera handelt, wird diese stark von der äußeren Lichtsituation beeinflusst. Diese äußern sich vor allem in kleinen Einzelpixelstörungen, deren Häufigkeit mit künstlichem Umgebungslicht zunimmt. Für die Filterung dieser Störungen kommt der durch OpenCV bereitgestellte Median-Filter zum Einsatz, der bei normalen Lichtverhältnissen gute Ergebnisse liefert, indem er jeden Pixel durch den Median seiner unmittelbar benachbarten Pixel ersetzt. Bei starkem Kunstlicht (im Rundkurs etwa Nachmittags, wenn die Sonne untergeht) reicht dieser jedoch nicht immer aus, um gegen die erhöhte Zahl an Störungen anzukommen, was zwar die Auswertung als Laserscan nicht verhindert, aber vor allem beim Aufbau der Costmap innerhalb der *Rundkurs mit Hindernissen*-Aufgabe[5] durch kleine erkannte, aber nicht tatsächlich vorhandene Hindernisse zu Problemen führt.

[5] siehe Abschnitt 7

5 ROS Navigation Stack

Um aus erfassten Sensordaten eine Trajektorie für das Fahrzeug zu erstellen, gibt es in ROS ein Framework, das die erforderlichen Schritte durchführt: den Navigation Stack. Dieser besteht aus einem Hauptknoten (`move_base`), der Informationen aus mehreren Quellen bezieht und je nach Konfiguration eine Nachricht mit Steuerbefehlen verbreitet. Da der Navigation Stack sehr gut im ROS-Wiki[6] dokumentiert ist, soll hier nur auf Besonderheiten bei dem verwendeten Fahrzeug eingegangen werden.

Um eine Trajektorie, also einen zeitabhängigen Pfad, zu erstellen, werden nacheinander ein globaler und ein lokaler Pfad bestimmt. Global bedeutet hier, dass der erstellte Plan weit in mehr oder weniger unbekanntes Gebiet führen kann. Er dient also dazu, vorneweg einen Ideallinie zu erstellen, der gefolgt werden soll. Dafür wird ein Planungsalgorithmus verwendet, der zu einem gegebenen Zielpunkt einen gültigen Pfad findet.

Der lokale Plan dagegen ist eher kurz und versucht das Fahrzeug anhand erhaltener Sensordaten zurück auf den globalen Pfad zu bringen. Damit aus den Pfaden eine Trajektorie wird, berechnet der Navigation Stack für jeden Punkt des lokalen Plans die Geschwindigkeit, die das Fahrzeug dort annehmen soll. Im Idealfall wird dadurch der global erstellte Pfad genau abgefahren.

Als Sensordaten wird der von der Kinect erzeugte Laserscan an den Navigation Stack gesendet. Dieser wird verwendet um einen globalen Plan zu erstellen, der die abzufahrende Strecke über eine größere Distanz darstellt. Außerdem wird der Laserscan verwendet um Hindernisse zu markieren.

Die Befehle, die der lokale Planer ausgibt, sind vom Typ `Twist`. Das bedeutet, dass sie für Roboter mit Differentialantrieb sind, die auch auf der Stelle rotieren können. Da dies für das Auto nicht zutrifft, muss eine Umrechnung der Befehle erfolgen:

$$\text{geometry\_msgs/Twist} : v, \omega \rightarrow \text{pses\_basis/command: steeringLevel, speedLevel}$$

$$r = \frac{v}{\omega}$$

$$\phi = \arctan\left(\frac{\text{wheelbase}}{r}\right)$$

$$\text{steeringLevel} = \frac{180}{\pi} * \frac{\phi}{\text{maxSteeringAngle}} * \text{maxSteeringCmd}$$

$$\text{speedLevel} = \frac{\text{maxSpeedCmd} - \text{minSpeedCmd}}{\text{maxVel} - \text{minVel}} * (v - \text{minVel})$$

Zum Angleich an die Ungenauigkeiten des realen Antriebes werden jeweils noch Offsets verwendet, die die minmal mögliche Geschwindigkeit und den Lenkeinschlag zum Geradeausfahren vorgeben.

[6] http://wiki.ros.org/navigation/Tutorials/RobotSetup

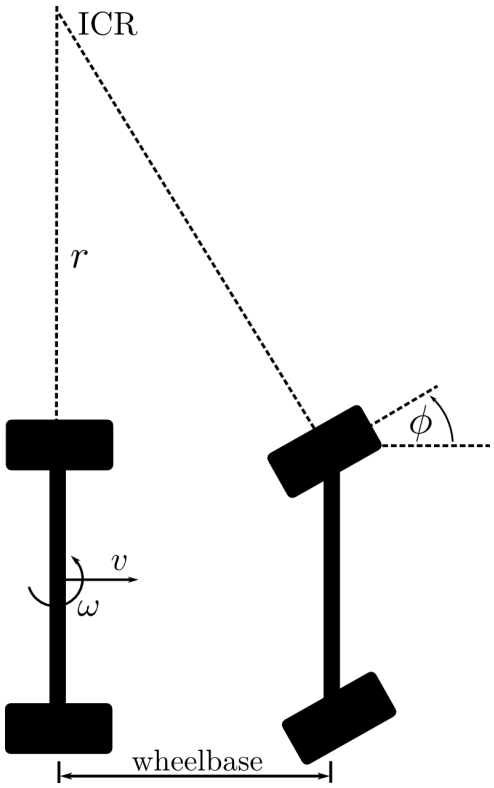

Abbildung 5.1: Modell der Fahrzeuggeometrie des verwendeten Autos, Quelle: ROS teb_local_planner

Mithilfe des Laserscans kann in der näheren Umgebung eine Karte erstellt werden, aus der wiederum eine Costmap erzeugt wird. Die Costmap vergibt für jede Zelle der Karte Kosten, wie teuer es ist, dort entlang zu fahren. Durch Einstellung eines Inflationsradius kann damit festgelegt werden, welchen Abstand das Fahrzeug von Hindernissen einhalten soll.

Weitere Erläuterungen zur Verwendung des Navigation-Stacks sind bei den Lösungsansätzen in den folgenden Kapiteln zu finden.

6 Rundstrecke ohne Hindernisse

Das Ziel bei der Aufgabe ‘Rundstrecke ohne Hindernisse’ ist es, einen Rundkurs innerhalb des Fachgebiets Echtzeitsysteme möglichst schnell abzufahren. Die Schwierigkeiten sind dabei die Kurven, von denen manche sehr eng zu fahren sind. Biegt man erst ab, sobald das Auto an der Kante steht, so ist es durch den begrenzten Lenkradius fast nicht möglich, diese zu nehmen, ohne mit der Wand zu kollidieren.

6.1 PID-Regelung

Diese Aufgabe wurde in mehreren Schritten angegangen. Um sich möglichst schnell mit dem Auto vertraut zu machen, wurde zunächst eine einfache Lösung gesucht. Dies wurde mit dem PID-Regler erreicht. Ein PID-Regler ist ein Regler, der die Regelabweichung $e(t)$ selbst (P), dessen Integration (I) und dessen Ableitung (D) betrachtet und aus dieser Summe die Stellgröße $u(t)$ berechnet:

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\, d\tau + K_\mathrm{d} \frac{de(t)}{dt}$$

Als zu regelnde Größe wurde der Abstand zur rechten Wand gewählt. Da die rechte Wand die innere Wand des Rundkurses ist, wenn mit dem Uhrzeigersinn gefahren wird, fährt das Auto so den Rundkurs.

Die Auslegung der Regelung besteht darin, passende $K_\mathrm{p}, K_\mathrm{i}, K_\mathrm{d}$ zu finden, damit das System stabil ist und schnell in den gewünschten Zustand überführt werden kann. Im ersten Schritt wurden die Parameter empirisch in der Simulation bestimmt und auf einer Teststrecke validiert.

Da es einen Sprung in den Werten des Ultraschalls gibt, wenn die Kante einer Kurve passiert wird, versucht der PID-Regler diesen Fehler auszuregeln. Das resultiert darin, dass er sehr stark zur Wand lenkt und somit abbiegt.

Dieser erste Ansatz konnte sehr zeitnah implementiert werden. Nachteile waren, dass die Regelparameter bestimmt werden mussten, was nicht zufriedenstellend gelöst werden konnte. Diese sind auch abhängig von der Geschwindigkeit, somit wäre eine empirische Bestimmung sehr zeitaufwendig und ungenau. Außerdem fehlte noch eine Geschwindigkeitsregelung . Schließlich ist das Abbiegen nicht für enge Kurven geeignet und nicht einfach kontrollierbar. Wenn man nämlich versucht den Regler schneller auszulegen, damit die Kurven enger gefahren werden, wird das Auto auf den Geraden instabiler. Daher wurde nach Möglichkeiten gesucht, diesen Ansatz zu verbessern.

6.2 Abbiegen mit Gierrate

Um das Verhalten beim Abbiegen zu verbessern, wurde dafür ein separater Regler entworfen, der nur aktiviert wird, wenn das Ende der zu folgenden Wand erkannt wurde. Dann wird der PID-Regler deaktiviert und stattdessen ein P-Regler aktiviert, der die Rotation um die Gierachse regelt. Das Ziel dieser Regelung ist es das Auto um 90° nach rechts um die Gierachse zu drehen und hierdurch abzubiegen. Nachdem sich das Auto um 90° nach rechts gedreht hat wird wieder auf den PID-Regler zum Folgen der Wand umgeschaltet.

Das Problem bei diesem Ansatz ist, dass man dafür einen Mechanismus benötigt, der erkennt, wenn die Wand endet und abgebogen werden muss. Anfangs wurde das Ende der Wand erkannt, sobald der Ultraschallsensor einen Abstand größer einem vorgegebenen Schwellenwert liefert. Wenn man das Ende der Wand aber schon etwas vorher erkennen würde, könnte man schon im Voraus einlenken und somit die Kurven enger fahren.

6.3 Wanderkennung und Navigation Stack

Der folgende Ansatz basiert darauf, dass mithilfe der Kinect die Wand erkannt wird und mit diesen Daten erstellt der globale Planer des ROS Navigation Stack einen globalen Plan. Diesem Plan folgt das Auto durch den `time-elastic-band local planner`.

6.3.1 Wanderkennung mithilfe der Kinect

Wie zuvor geschrieben ist es schwierig bis unmöglich, einige Kurven zu nehmen, wenn man erst an der Kante selbst einlenkt. Daher ist der logische nächste Schritt, die Kante vorher zu erkennen und dann präventiv einzulenken. Zur frühzeitigen Erkennung der Kante lässt sich der Laserscan, der mithilfe des ROS-Paketes `depthimage_to_laserscan` [7] aus dem Tiefenbild der Kinect gewonnen wird, gut verwenden, da dieser ein Abbild der unmittelbaren Umgebung vor dem Auto liefert.

Der Laserscan enthält den Abstand zu Hindernissen für verschiedene Winkel. Um aus diesen Daten die Position und Richtung der Wand erkennen zu können, ist es sinnvoll den Laserscan zuerst in Koordinaten eines absoluten TF-Frames[8] zu konvertieren (2D-Punktewolke).

Der Algorithmus für das Erkennen der Wand basiert auf einem iterativen Ansatz. Mit jeder neuen Messung des Laserscans werden auf Basis der bisherigen Schätzung der Wand alle Punkte der Punktewolke entfernt, dessen Abstand zur bisherigen Schätzung der Wand größer ist als ein gewisser Wert. Dadurch werden im Voraus alle Punkte entfernt, die die bisherige Schätzung der Wand nicht verbessern. Die verbleibenden Punkt

[7] mehr dazu in Abschnitt 4.1

[8] TF ist ein ROS-Paket, dass die Relation mehrere Koordinatensysteme zueinander für unterschiedliche Zeitpunkte verwaltet.

werden danach der Reihen nach durchgegangen und das Ende der Wand wird erkannt, wenn der Abstand zweier Punkten größer als der dafür vorgegebene Wert ist. Auf diese Weise kann sichergestellt werden, dass offene Türen nicht als Ecke erkannt werden. Aus den verbleibenden Punkten wird eine Ausgleichsgerade berechnet, die zur neuen Schätzung der Wand wird. Da es vorkommen kann, dass die Erkennung des Endpunktes der Wand durch Rauschen gestört wird, werden die Endpunkte entsprechend ihrer Häufigkeit gewichtet und nur der Endpunkt mit der größten Gewichtung wird als aktueller Zielpunkt gewählt.

Da am Anfang noch keine Schätzung der Wand vorhanden ist, geht man davon aus, dass das Auto parallel zur Wand steht und der Abstand zur Wand wird mittels der Ultraschallsensoren bestimmt. Nach einer Kurve wird die Wandschätzung ebenfalls um 90° gedreht.

Der Nachteil dieses Ansatzes ist aber, dass wenn sich an der Startposition viel Raschen im Tiefenbild befindet die Schätzung der Wand schlecht ist und sich diese erst mit mehreren Messungen verbessert, weil der Richtungsvektor der Wand nach jeder Iteration aufaddiert wird.

6.3.2 Erstellen des globalen Plans

Da man durch die Wanderkennung mithilfe der Kinect die genaue Position und Richtung der Wand kennt, kann man diese Daten nutzen um daraus eine globalen Plan zu erstellen. Der globale Plan stellt den Pfad zwischen der aktuellen Position des Autos und einem vorgegebenen Zielpunkt dar. Da der Laserscan eine begrenzte Distanz hat, muss der globale Plan regelmäßig aktualisiert werden. Dies geschieht, wenn das Auto einen Meter vom aktuellen Zielpunkt des Plans entfernt ist.

Solange das Ende der Wand noch nicht erkannt wurde, plant der globale Planer eine Gerade. Wenn aber das Ende der Wand erkannt wurde, wird eine Kurve geplant, wobei die Kurve schon vor der Kante angefangen wird.

6.3.3 Lösung mit time-elastic-band local planner

Um dem erstellten globalen Plan zu folgen, wird der time-elastic-band (teb) Planer verwendet[9], der die lokale Trajektorie erstellt. Der Planer sendet dann Nachrichten mit den einzustellenden Geschwindigkeiten und Lenkwinkeln, die über einen Knoten in Fahrzeugbefehle umgewandelt werden.

Da der globale Plan größtenteils gerade verläuft und sich nicht oft ändert, ist es möglich dem lokalen Planer die Vorgabe zu geben, sehr genau dem globalen Plan zu folgen. Dadurch können hohe Geschwindigkeiten erreicht werden.

Der lokale Planer wird durch diverse Parameter konfiguriert, über die die Fahrzeuggeometrie und die möglichen Bewegungen mitgeteilt werden. Einzelne Aspekte der Trajek-

[9] http://wiki.ros.org/teb_local_planner/

torie könenn unterschiedlich gewichtet werden, um die Beachtung von Hindernissen, Einhaltung der maximalen Lenkwinkel und Geschwindigkeiten, etc. zu steuern. Es kann etwa Rückwärtsfahren bestraft werden, da dies ohne rückwärtige Sensoren gefährlich ist.

Da der lokale Planer eine Trajektorie plant, ist automatisch eine Geschwindigkeitsregelung inkludiert. Somit muss diese nicht zusätzlich geregelt werden.

6.3.4 Verwendete Knoten

Die folgenden Knoten werden für diese Aufgabe gestartet:

mephobia_controller_node enthält die oben beschriebene Implementierung der Aufgabe

mephobia_sensors_node filtert die Sensordaten

cmd_vel_to_pses_basis_command übersetzt die Kommandos des Planers in Befehle, die das Auto steuern

depthimage_to_laserscan erzeugt einen Laserscan aus dem Tiefenbild der Kinect

move_base wird mit dem globalen und lokalen Planer konfiguriert und führt die Trajektorienplanung aus

Zusätzlich wird das bereitgestellte PSES_Basis launch-File gestartet, in dem der Knoten zur Verteilung der Sensordaten und Verarbeitung der Motorbefehle gestartet wird.

6.3.5 Konfiguration

Das ROS-Pakte `mephobia_controller` enthält die Implementierung dieser Aufgabe und besteht aus dem eigentlichen Controller, dem `circuit_planner`, der als Plugin für den globalen Planer des Navigation Stacks fungiert, sowie aus Konfigurationsdateien und launch-Dateien, die die notwendigen Knoten mit entsprechenden Parametern laden.

Der Controller wird durch Parameter in `mephobia_controller` konfiguriert. Dabei kann beispielsweise angegeben werden, ob das Auto der rechten oder der linken Wand folgen soll und der dabei einzuhaltende Abstand zur Wand. Des Weiteren lässt sich damit auch der Kurvenradius beim Abbiegen einstellen.

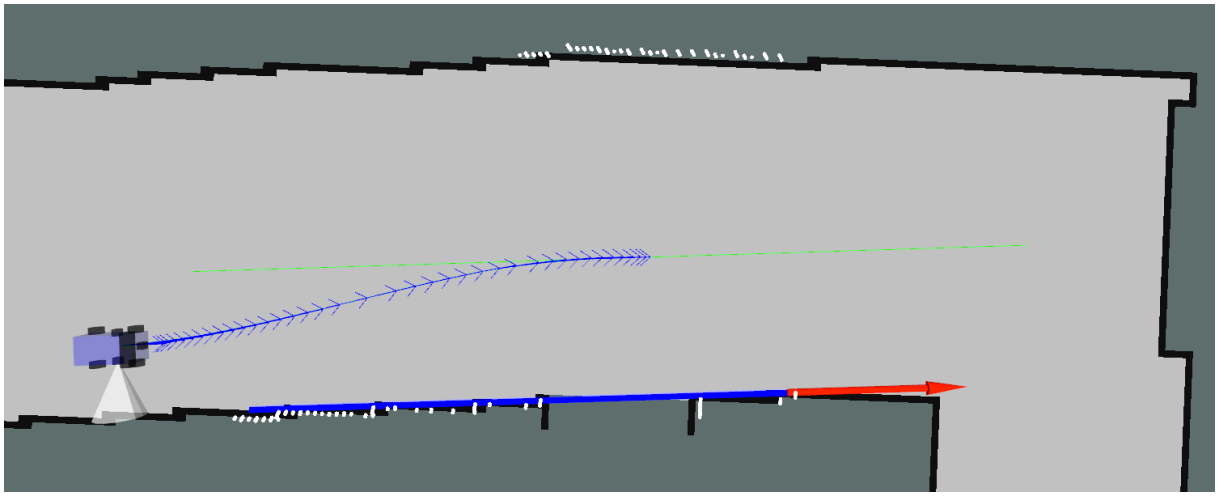

Abbildung 6.1: Simulation des Controllers während einer Geraden. Man erkennt den Laserscan (weiße Punkte), die Schätzung der Wand (blaue Linie), den globalen Plan (grüne Linie) mit dem aktuellen Zielpunkt (roter Pfeil) sowie die lokale Trajektorie (blaue Pfeile).

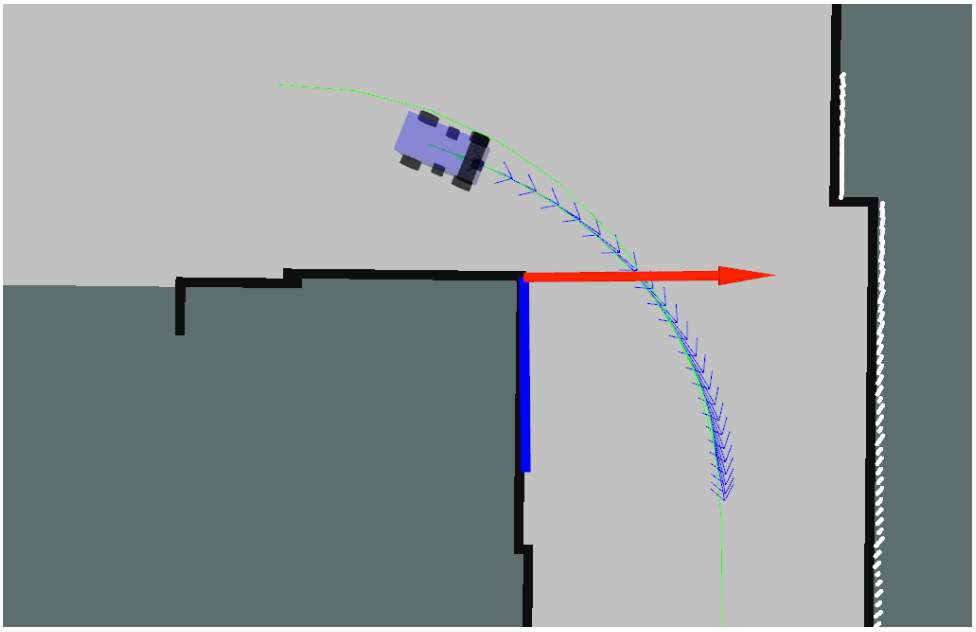

Abbildung 6.2: Simulation des Controllers während einer Kurve. Man erkennt den Laserscan (weiße Punkte), die Schätzung der Wand (blaue Linie), den globalen Plan (grüne Linie) mit dem aktuellen Zielpunkt (roter Pfeil) sowie die lokale Trajektorie (blaue Pfeile).

7 Rundstrecke mit Hindernissen

Um den im vorherigen Abschnitt beschriebenen Rundkurs auch dann erfolgreich zu durchfahren, wenn unbekannte Hindernisse den Weg erschweren, ist ein anderes Vorgehen als bisher erforderlich. Im Folgenden werden verschiedene Ideen vorgestellt, von denen schließlich eine implementiert wurde.

Ein erster naiver Ansatz ist, auf der bekannten Karte einen globalen Pfad zu planen. Dazu bestimmt der Roboter seine Position auf der Karte durch Odometrie und Landmarken autonom. Wenn durch einen Laserscan die nicht kartierten Hindernisse erkannt werden, wird eine Trajektorie geplant, um diesen auszuweichen (Local Plan). Voraussetzungen dafür sind dass die Karte ausreichend genau ist und die Lokalisierung zuverlässig funktioniert.

Da die Karte erst spät zur Verfügung stand, nicht von unserem Team erstellt wurde und auch der Rundkurs ohne Hindernisse ohne Karte gelöst wurde, wurde entschieden, auch bei dieser Aufgabe auf eine Planung anhand einer Karte zu verzichten. Stattdessen wurde ein Ansatz gewählt, bei dem die Lösung der ersten Aufgabe wiederverwendet werden konnte. Trotzdem wurde die Karte zum Testen in der Simulation verwendet.

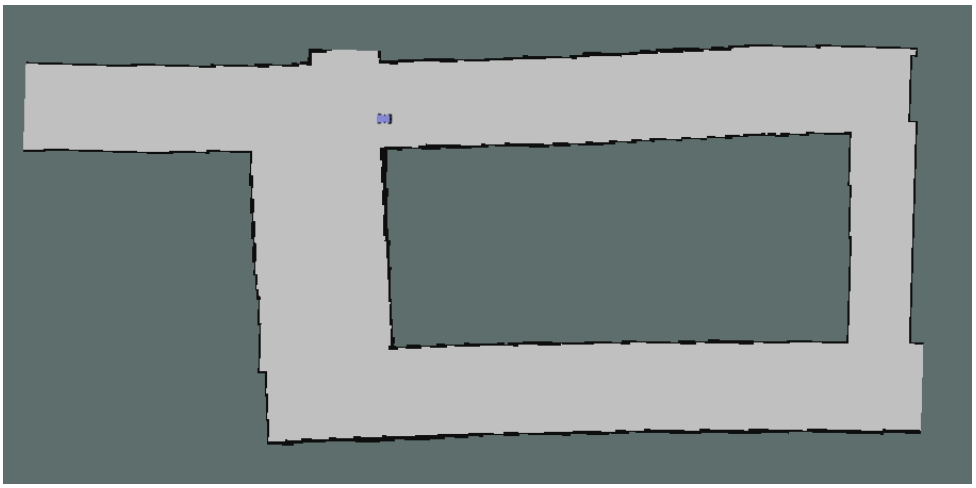

Abbildung 7.1: Karte des Fachgebiets Rechnersysteme. Die rechte Schleife ist die Strecke des Rundkurses.

7.1 Umsetzung

Durch eine kleine Modifikation im ROS-Paket `depthimage_to_laserscan` konnten, wie in Abschnitt 4.1 beschrieben, zwei Laserscans mit verschiedenen Höhen erzeugt werden. Dabei wird mit dem oberen Scan nur die Wand erkannt und dieser gefolgt. Der untere Scan dagegen detektiert auch im Weg stehende Hindernisse.

Somit wird ein globaler Plan wie in einer Umgebung ohne Hindernisse erzeugt, diesem aber so gefolgt, dass gleichzeitig Hindernisse umfahren werden. Um dies zu erreichen

muss der lokale Planer so konfiguriert werden, dass es genug Freiheiten gibt, den globalen Plan zu verlassen und trotzdem das Ziel angefahren wird.

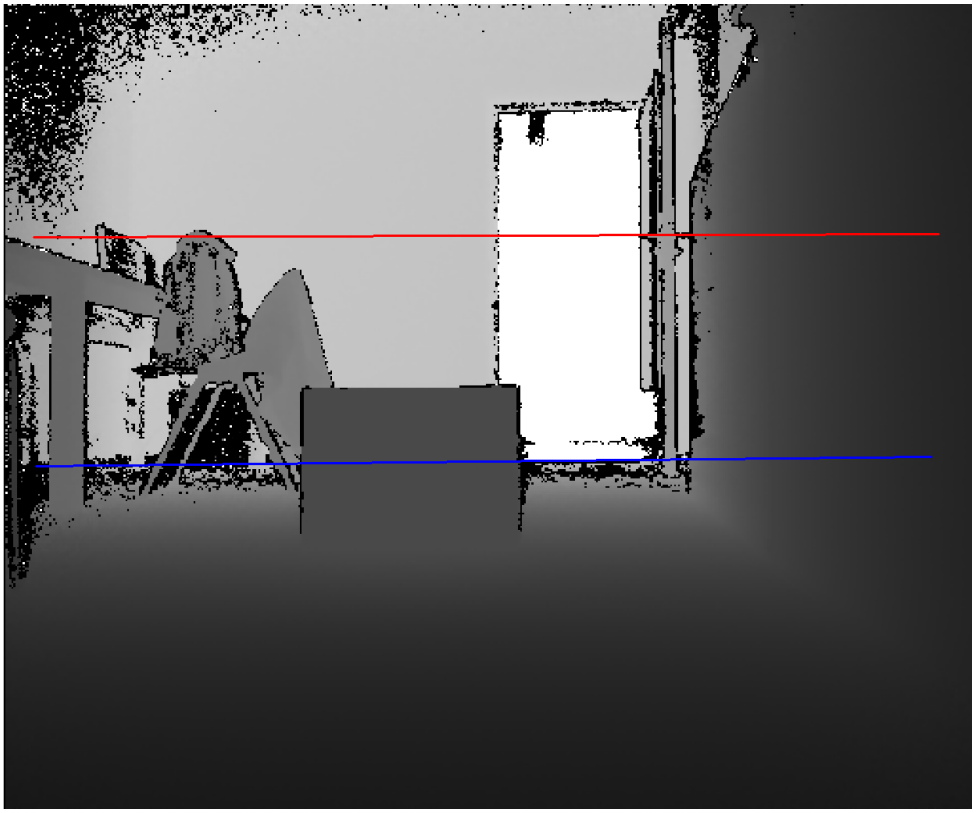

Abbildung 7.2: Tiefenbild der Kinect. Je dunkler ein Pixel ist, desto näher. Es sind zwei Laserscans auf unterschiedlichen Höhen eigezeichnet. Der obere Laserscan kann über Hindernisse hinwegsehen und trotzdem die Wand erkennen. Der untere erkennt auch den im Weg stehenden Karton.

7.1.1 Verwendete Knoten

Die folgenden Knoten werden für diese Aufgabe gestartet:

mephobia_controller_node erzeugt den globalen Plan

mephobia_sensors_node filtert die Sensordaten

cmd_vel_to_pses_basis_command übersetzt die Kommandos des Planers in Befehle, die das Auto steuern

depthimage_to_laserscan erzeugt einen Laserscan aus dem Tiefenbild der Kinect

move_base wird mit dem globalen und lokalen Planer konfiguriert und führt die Trajektorienplanung aus

Zusätzlich wird das bereitgestellte PSES_Basis launch-File gestartet, in dem der Knoten zur Verteilung der Sensordaten und Verarbeitung der Motorbefehle gestartet wird.

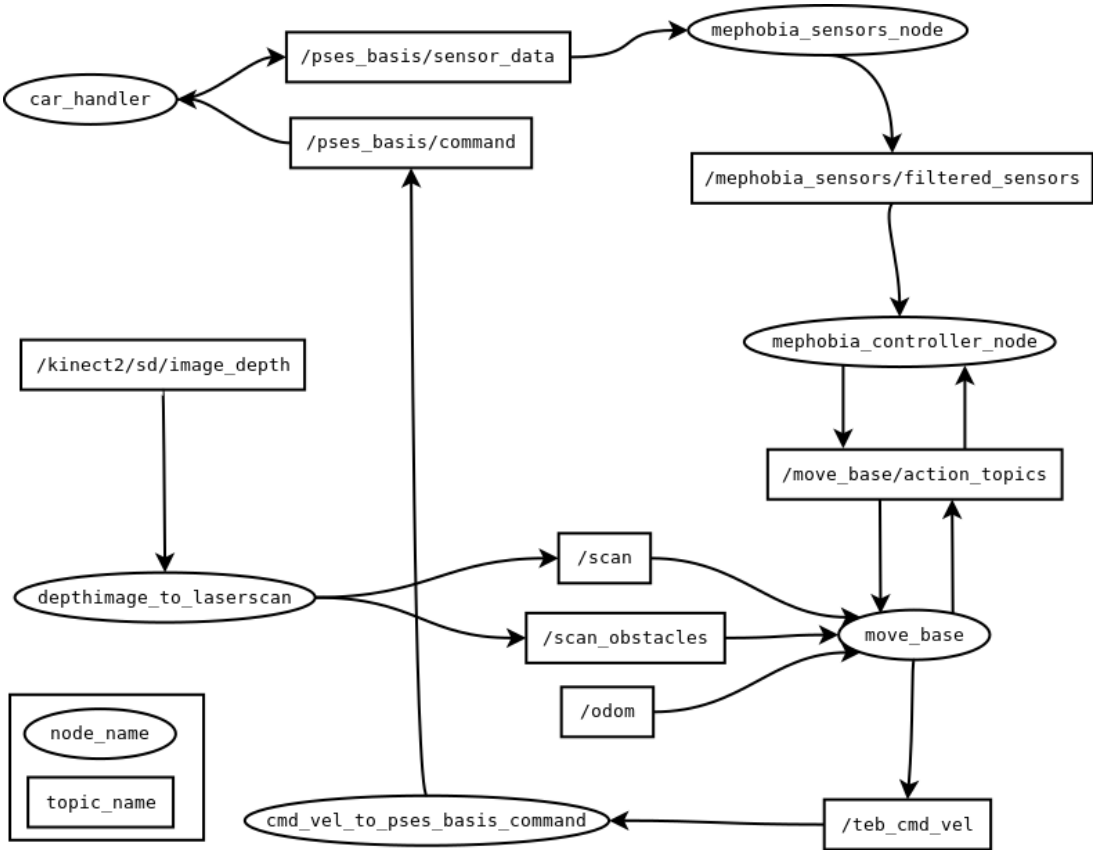

Abbildung 7.3: Vereinfachte Darstellung des Zusammenspiels der einzelnen Knoten

7.1.2 Konfiguration

Das ROS-Pakte `mephobia_teb_planner`, das die Implementierung dieser Aufgabe beinhaltet, besteht nur aus Konfigurationsdateien und launch-Dateien, die die notwendigen Knoten mit entsprechenden Parametern laden. Dabei wird der Controller der vorherigen Aufgabe gestartet und der dafür vorgesehene Laserscan mit einem Offset nach oben versehen, sodass über Hindernisse hinweg geschaut wird. Dann wird der ROS-Navigation-Stack wie oben mit einem Global- und einem Lokalplaner gestartet.

Wie zuvor wird auch hier der teb-Planer eingesetzt, der den unteren Laserscan erhält und daraus mithilfe eines Costmap Converters eine Costmap erstellt. Dabei werden einzelne Punkte des Scans herausgeworfen und größere Cluster zu Linien umgesetzt. Der Planer erhält hier etwas andere Gewichte, um eine größere Abweichung vom globalen Plan zu erlauben, falls Hindernisse den Weg blockieren.

7.2 Probleme

Bei der Live-Demo konnten wir unsere Ergebnisse leider nicht richtig präsentieren, da die Startposition kurz vor einer Kurve war und die Kurvenerkennung bei dem Hinderniskurs auch während der Testphase nicht immer robust funktioniert hat. Das Pro-

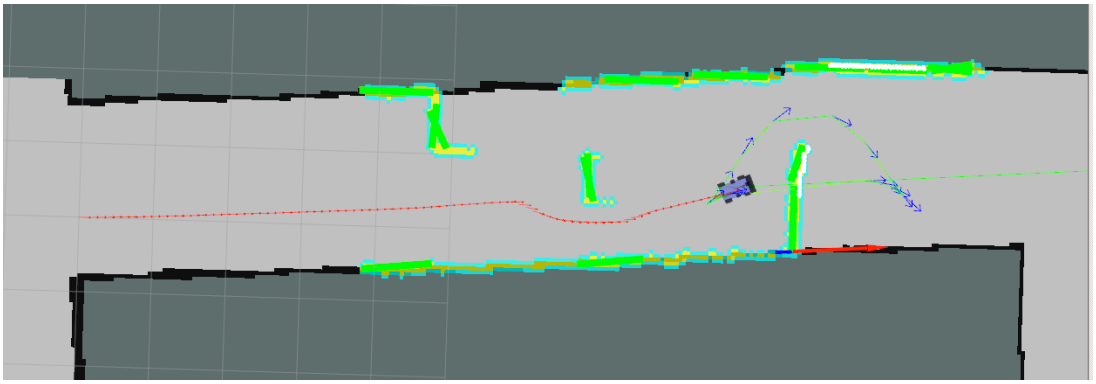

Abbildung 7.4: Über der türkisen Costmap sind die durch den Converter erzeugten grünen Linien zu sehen, die erkannte Hindernisse darstellen, um die ein Pfad geplant wird.

blem dabei war, dass das Ende der Wand nicht im Laserscan erkannt wurde. Da bei dem Hinderniskurs das Auto auch mal schräg zur Wand stehen kann, ist die Erkennung schwieriger als bei dem Kurs ohne Hindernisse. Bei der Implementierung und den Testläufen hatten wir uns auf die Umfahrung von Hindernissen auf einer geraden Strecke fokussiert.

8 Fahrbahnmarkierung erkennen und Spur halten

Eine der Aufgaben des Carolo-Cups ist es, auf einer zweispurigen Straße, anpgepasst auf die Größe des Roboterautos, zu folgen [Car16, S. 18-21]. Dabei soll das Auto stets auf seiner Fahrbahnseite bleiben, dem Streckenverlauf folgen und die Markierungen nicht übertreten. Diese Aufgabe wurde auch im Rahmen des Projektseminars angeboten.

Die Aufgabe lässt sich grob in zwei Teile zerlegen: zunächst müssen die Fahrbahnmarkierungen erkannt werden. Dann muss mit diesem Wissen das Auto so bewegt werden, dass es in der Spur bleibt und nicht übertritt.

Die Fahrbahnerkennung konnte aus Zeitmangel nicht auf dem richtigen Fahrzeug zum Laufen gebracht werden. Daher wurde der Fokus darauf gelegt, eine Simulation für diese Aufgabe zu erstellen, damit Ansätze für die Regelung einfach ausprobiert werden können.

Abbildung 8.1: Ansicht einer Fahrbahn, wie sie auch beim Carolo-Cup Verwendung finden könnte. Die deutliche Abgrenzung zwischen Fahrbahn und Markierungen erleichtert die Erkennung enorm.

8.1 Fahrbahnerkennung

Die Fahrbahnerkennung wurde bereits im Rahmen einer vorherigen Bachelorarbeit implementiert [Sep17]. Die Idee dahinter ist, das Bild der Kinect in eine Ansicht von oben zu transformieren. Somit wird die perspektivische Projektion in eine orthographische Projektion umgewandelt, parallele Geraden in der Welt sind danach auch wieder parallel in der Abbildung.

Anschließend wird ein Kantenfilter auf das transformierte Bild angewandt, um die Fahrbahnmarkierungen herauszustellen. Schließlich werden die einzelnen Pixeln zu Linien

zusammengefasst, um eine geometrische Beschreibung der Markierungen zu erhalten. Dies können Geraden oder etwa Splines sein.

8.2 Simulation

Die Simulation besteht aus zwei Teilen: zunächst muss eine Strecke vorgegeben werden, die in der Simulation abgefahren werden soll. Dies übernimmt der `PathGenerationNode`. Eine Strecke hier besteht aus einer Liste von `geometry_msgs/PoseStamped`-Nachrichten. Diese beinhalten die x-y-Koordinaten und die Blickrichtung des Punktes sowie den Namen des verwendeten Koordinatensystems. Um diese Pfade zu generieren, wurde ein einfaches Programm entwickelt. Eine Strecke wird dann durch Befehle wie

```
turtle = Turtle()
turtle.forward(3)
turtle.arc(270, 3)
turtle.forward(6)
turtle.arc(-270, 3)
turtle.forward(3)
```

beschrieben. Der so erstellte Pfad wird in eine Textdatei geschrieben und vom `PathGenerationNode` eingelesen. Der gesamte Pfad als Referenz und den Abschnitt, den das Auto gerade sehen kann, wird dann dem `LaneFollowingNode` übergeben, der für die Regelung des Autos zuständig ist.

Zur Bestimmung, was genau das Auto gerade sehen kann, wird zunächst der Punkt auf dem Pfad bestimmt, der am nächsten zum Auto liegt. Das Auto kann dann alle Punkte in einem Abstand zwischen zwei konfigurierbaren Werten (*min_sense_distance* und *max_sense_distance*) von diesem Punkt aus sehen. Punkte, deren Blickrichtung zu weit von der des Autos abweichen, werden ignoriert. Dies kann etwa bei Kreuzungen wichtig sein, damit das Auto geradeaus fährt und nicht abbiegt.

Diese beiden Pfade werden zusammen mit dem Auto und dem zurückgelegten Weg in der Simulation dargestellt. Eingezeichnet sind die gesamte Strecke (weiß) und der Teil der Strecke, den das Auto sehen kann (rot). Die Fahrbahn ist grau.

8.3 Regelung

Das Ziel der Regelung zum Folgen der Fahrbahn ist, möglichst genau dem Pfad zu folgen. Vor allem soll verhindert werden, dass das Auto Kurven schneidet oder anderweitig Markierungen übertritt. Das Koordinatensystem ist so gewählt, dass die Kamera der Ursprung ist und die x-Achse in Fahrtrichtung liegt.

Im Folgenden werden zwei Ansätze vorgestellt, wie eine solche Regelung aussehen kann.

8.3.1 Carrot Controller

Zunächst wird eine Lösung vorgestellt, die auch ähnlich in [Sep17] als erster Ansatz vorgestellt wurde. Das Auto folgt einem Punkt auf dem von ihm sichtbaren Pfad. Genauer gesagt, versucht das Auto die gleiche Ausrichtung (Pose) zu haben wie dieser Punkt. Es ist analog zum Esel und der Möhre, daher der Name 'Carrot Controller'.

Somit ist der Regelfehler die Differenz zwischen aktueller Ausrichtung und Ausrichtung des Punktes. Betrachtet man dies im lokalen Koordinatensystem des Autos, so ist es nur die Ausrichtung des Punktes.

Die Bestimmung des zu folgenden Punktes führt zu ein paar Problemen: ist er zu nah gewählt, können stärkere Krümmungen der Strecke nur schwierig gefolgt werden. Ist er zu weit gewählt, dann werden Kurven geschnitten und Markierungen übertreten. Außerdem gibt es meist eine konstante Regelabweichung: dem Verlauf wird gefolgt, jedoch fährt das Auto neben der Strecke. Dies ist etwa in Fig. 8.3 zu sehen. Allgemein lässt sich sagen, dass es kein optimalen Wert für den Lookahead gibt, da er abhängig von Geschwindigkeit und der (meist unbekannten) Strecke ist.

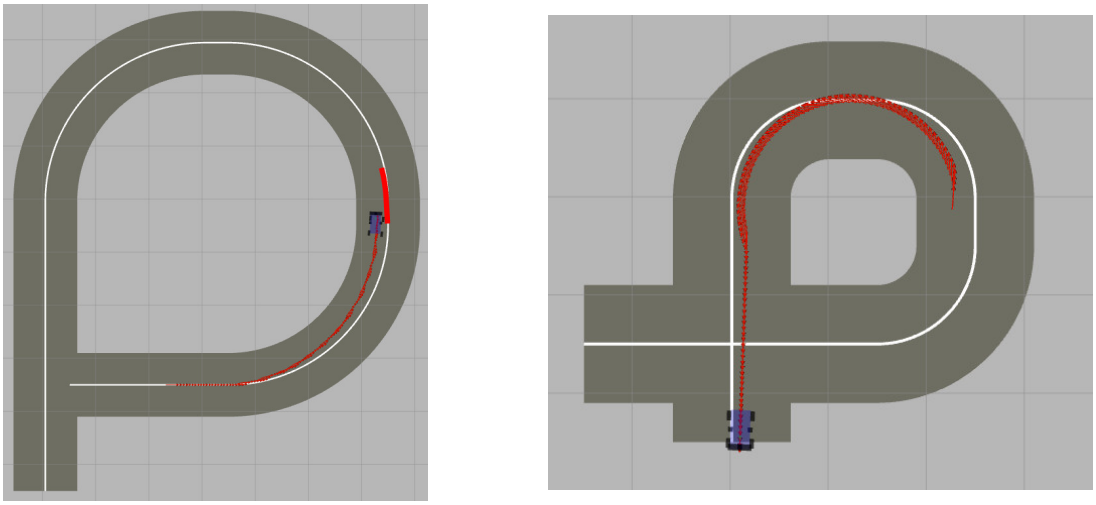

Abbildung 8.2: Fahrbahnfolgen durch Carrot Controller. Die Abweichung von der geplanten Strecke und das Übertreten der Markierungen ist deutlich zu sehen.

8.3.2 Stanley Method

Ein großes Problem, dass der zuvor beschriebene Carrot-Controller hat, ist die konstante Regelabweichung. Außerdem hat der PID-Regler für die Ausrichtung viele Parameter, die bestimmt werden müssen. Eine naheliegende Lösung ist, die Querabweichung des Autos zum Pfad mit in den Regelfehler einzubeziehen. Somit werden Abweichungen in Position *und* Ausrichtung bestraft, sodass ein bessere Folgegenauigkeit zu erwarten ist.

Es gibt mehrere konkrete Regler, die diese Idee implementieren, siehe etwa [Sni09] für einen Überblick. Es wurde die einfachste gewählt, die auch in der Darpa Grand Challenge vom Gewinnerauto Stanley eingesetzt wurde [TMD+06].

Durch die geschickte Wahl des Koordinatensystems ist der Fehler in der Ausrichtung wieder die Krümmung des Pfades, die Querabweichung ist die y-Koordinate. Diese Werte können jeweils von dem dem Auto nächsten Punkt oder von einem Punkt mit einem bestimmten Abstand zum Auto bestimmt werden.

Der gewünschte Lenkeinschlag $\delta(t)$ bestimmt sich dann wie folgt:

$$\delta(t) = \psi(t) + \tan^{-1} \frac{k\varepsilon(t)}{u(t)}$$

Dabei ist $\psi(t)$ die Ausrichtung des Zielpunktes, $\varepsilon(t)$ die Querabweichung und $u(t)$ die Geschwindigkeit. Der einzige Parameter ist k und bestimmt, die stark die Querabweichung bestraft wird. Er wurde empirisch in der Simulation bestimmt.

Der Arcustangens kann als Sigmoid verstanden werden, der die möglichen Werte auf $-90° < y < 90°$ beschränkt (oder in einen Winkel umwandelt), damit das Auto nicht rückwärts fährt, um den Pfad zu erreichen. Es wird durch die Geschwindigkeit geteilt, da Lenkeinschläge bei höherer Geschwindigkeiten geringer seien sollten.

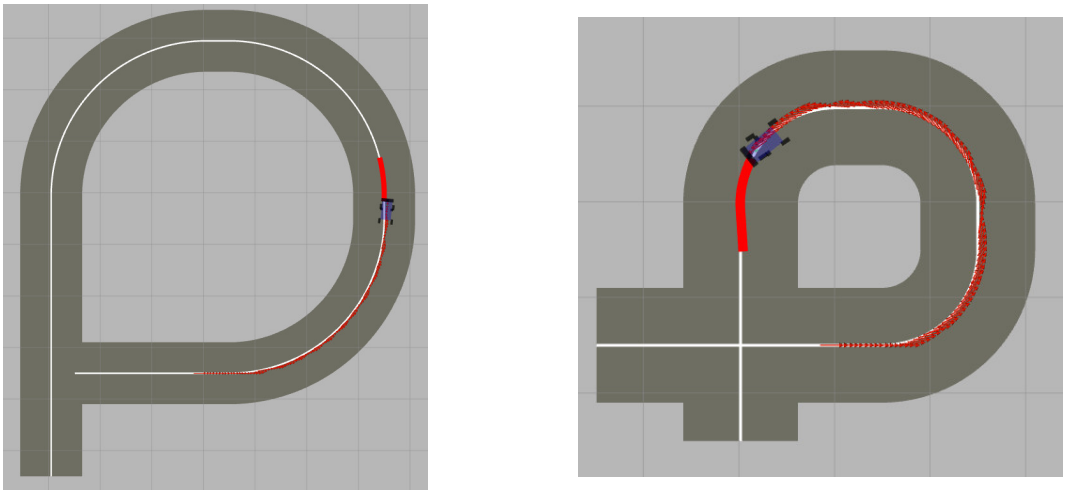

Abbildung 8.3: Fahrbahnfolgen durch Stanley Controller. Abweichungen von der geplanten Strecke sind gering bis gar nicht vorhanden. Es gibt keine konstante Regelabweichung in den Kurven.

8.4 Ergebnisse aus der Simulation

Die Simulation zeit, dass der Stanley-Controller gegenüber dem simplen Carrot-Controller deutlich genauer folgt, keine Kurven schneidet und einfacher zu konfigurieren ist. Es muss jetzt mit dem Auto auf einer echten Strecke getestet werden, ob dies auch in der Realität so ist.

Dazu muss das im Rahmen dieser Arbeit erstellte Paket zur Folgeregelung mit dem bereits existierenden Paket zur Fahrbahnerkennung integriert werden.

9 Fazit

Im Weiteren wird beschrieben, was gut und was weniger gut während des Projektes funktioniert hat.

9.1 Simulation

Die bereitgestellte Simulation hat es sehr erleichtert, erste Ideen auszuprobieren, ohne das Auto benutzen zu müssen. Dies war etwa die Wanderkennung und das Fahrbahnfolgen. Für Aufgaben, die die echte Physik erfordern, etwa Auslegung des Reglers, war es nur für eine sehr grobe Parametrisierung nutzbar. Manche Aspekte mussten angepasst werden, wie etwa die Konfiguration der Karte über Launch-Parameter statt Anpassung von Dateien im Paket selber.

9.2 Probleme

Obwohl die aktuelle Hardware-Plattform des Fahrzeuges durchaus angemessen und zufriedenstellend ist, gibt es dennoch einige Probleme, die während des Projektes auffällig wurden.

Eines der auffälligeren Probleme war dabei das uc-Board, dass Motorsteuerung und Sensoren kontrolliert und ohne offensichtlich reproduzierbaren Grund abstürzte und nicht mehr auf Kommunikationsversuche reagierte. Dies erforderte einen vollständigen Neustart des Fahrzeuges (da uc-Board und Computer-Board zusammen mit Strom versorgt werden) und ist problematisch, wenn das Fahrzeug zum Absturzzeitpunkt gerade mit (erhöhter) Geschwindigkeit unterwegs ist (da keine Stopp-Befehle verarbeitet werden können).

Gleichzeitig wurde in diesem Zusammenhang ein weiteres Problem am Chassis des Fahrzeuges deutlich: Der vorhandene Schaumstoff-“Stoßdämpfer“ ist zu klein und falsch positioniert, da der Großteil aller Kollisionen des Fahrzeuges mit Wänden und Hindernissen an den vorderen Ecken erfolgt, die nicht geschützt sind.

Die Lenkung des Fahrzeuges ist zwar insgesamt gut, aber zieht spürbar zur Seite (bei unserem Fahrzeug nach rechts) und die erreichten Steuerwinkel sind nach Links und Rechts unterschiedlich.

Für die Sensorik wurde gegen Ende der Implementation deutlich, dass der in `pses_basis` verwendete Ansatz für Odometrie bei niedrigen Geschwindigkeiten deutlich abdrifted. Außerdem äußerte sich die stark variierende Störempfindlichkeit der Kinect in Abhängigkeit von der (Kunst-)Lichtsituation als problematisch, da Testfahrten abhängig von der Tagesszeit zu unterschiedlichen Ergebnissen führten.

Für zukünftige Hardware-Plattformen wäre außerdem ein Sensor nach hinten (etwa ein zusätzlicher Ultraschallsensor) hilfreich.

Abschließend gab es auch bei den Akkus der Fahrzeuge Probleme, so waren gegen Ende manche der Akkus defekt (sehr schnelle Entladezeit) oder wurden von anderen Gruppen leer hinterlassen. Außerdem störten die Ladegeräte, von denen einige sehr laut waren.

Für ein neues Projektseminar wäre es auch schön, wenn die ROS-Version auf dem Auto geupdatet werden könnte, da diese veraltet ist und auf neueren Ubuntu-Versionen nicht mehr installiert werden kann. Somit mussten auf den eigenen Laptops virtuelle Maschinen oder Docker-Container benutzt werden. Manche Gruppenmitglieder haben die bereitgestellten Pakete nach kleinen Anpassungen ohne Probleme unter Kinect benutzt.

9.3 Projektmanagement

Der Ansatz zum Projektmanagement funktionierte zu Beginn sehr gut, jedoch entwickelten sich im Verlauf der Implementation durch die aufeinander aufbauenden Ansätze für die Rundkurs-Aufgaben unübersichtliche Abhängigkeiten. Durch die Änderungen für eine Aufgabe gab es unerwartete Auswirkungen auf die anderen.

Dies wurde vor allem durch die reduzierte direkte Kommunikation im Team nach Beginn der Klausurenphase sichtbar und führte zu verlorener Zeit durch Fehlersuche.

Die Planung an sich hat jedoch gut funktioniert, die Kommunikation sollte verbessert werden.

Literatur

[Car16] Carolo-Cup: *Carolo-Cup Regelwerk 2017*. https://wiki.ifr.ing.tu-bs.de/carolocup/system/files/Regelwerk_Junior_2017_20161016.pdf. Version: 2016

[GA02] Galanis, George ; Anadranistakis, Manolis: A one-dimensional Kalman filter for the correction of near surface temperature forecasts. In: *Meteorological Applications* 9 (2002), Nr. 4, S. 437–441. http://dx.doi.org/10.1017/S1350482702004061. – DOI 10.1017/S1350482702004061

[Pro] Projektseminar Echtzeitsysteme: *Beschreibung Fahrzeug und ucboard*. https://github.com/tud-pses/ucboard/blob/master/ucboard.pdf

[Sep17] Sepulveda, Nicolas A.: *Fahrbahnerkennung und automatisierte Fahrbahnführung anhand einer Farbkamera mit einem Modellfahrzeug*, TU Darmstadt, Bachelorarbeit, Oktober 2017

[Sni09] Snider, Jarrod M.: *Automatic Steering Methods for Autonomous Automobile Path Tracking*. 2009

[TMD+06] Thrun, Sebastian ; Montemerlo, Mike ; Dahlkamp, Hendrik ; Stavens, David ; Aron, Andrei ; Diebel, James ; Fong, Philip ; Gale, John ; Halpenny, Morgan ; Hoffmann, Gabriel ; Lau, Kenny ; Oakley, Celia ; Palatucci, Mark ; Pratt, Vaughan ; Stang, Pascal ; Strohband, Sven ; Dupont, Cedric ; Jendrossek, Lars-Erik ; Koelen, Christian ; Markey, Charles ; Rummel, Carlo ; Niekerk, Joe van ; Jensen, Eric ; Alessandrini, Philippe ; Bradski, Gary ; Davies, Bob ; Ettinger, Scott ; Kaehler, Adrian ; Nefian, Ara ; Mahoney, Pamela: Stanley: The Robot That Won the DARPA Grand Challenge: Research Articles. In: *J. Robot. Syst.* 23 (2006), September, Nr. 9, 661–692. http://dx.doi.org/10.1002/rob.v23:9. – DOI 10.1002/rob.v23:9. – ISSN 0741–2223