

Projektseminar Echtzeitsysteme Abschlussbericht

Team TUrTlES

Seminar eingereicht von

Jonas Tamm-Morschel, Puria Izady, Tim Piscator, Francisco Medina, Christoph Weckert
am 8. April 2018



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: Géza Kulscar
Betreuer: Géza Kulscar

Erklärung zum Seminar

Hiermit versichere ich, das vorliegende Seminar selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die ich 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.

Ich erkläre mich damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 8. April 2018

(Jonas Tamm-Morschel, Puria Izady, Tim Piscator, Francisco Medina, Christoph Weckert)



Inhaltsverzeichnis

1 Einleitung	1
1.1 Aufgabenstellung	1
1.2 Überblick	1
2 Rundkurs	2
2.1 Modellbildung und Simulation	2
2.2 Regler	2
2.2.1 Kurvenerkennung mittels Ultraschallsensor	3
2.2.2 Kurvenerkennung mittels Gyroscope	3
2.3 Ergebnisse	4
3 Rundkurs mit Hindernissen	5
3.1 Das Konzept des Rundkurs mit Hindernissen	5
3.2 Der Navigationstack	6
3.2.1 Mapserver	6
3.2.2 Global planner und teb local planner	6
3.3 AMCL	7
3.4 SLAM	7
4 Schnittstelle zum Auto	8
4.1 Motorkennlinie	8
4.2 Lenkwinkelkennlinie	10
5 Einparken	12
5.1 Dynamic Wall-Follow im Detail	13
5.2 Verwendeter PID-Regler	13
5.3 Parklückenerkennung	14
5.3.1 Ultraschall	15
5.3.2 Hough Transform	15
5.4 Der eigentliche Einparkvorgang	16
6 Fazit und Ausblick	19
6.1 Rundkurs	19
6.2 Einparken	20
6.2.1 Kalman-Filter	20
6.2.2 Verbesserungen des Gyroskops	20
6.2.3 Mehr Sensorik	21
6.3 Allgemein	21
A Erster Anhang	23

Abbildungsverzeichnis

2.1	PD-Regler	3
2.2	PID-Regler	4
3.1	Pfadplanung	5
3.2	KOM	6
3.3	Gmapping	7
3.4	KOM	7
4.1	Ermittelte Datenpunkte und Motorkennlinie	9
4.2	Ermittelte Datenpunkte und Lenkwinkelkennlinie	11
5.1	Der Einparkvorgang [Roj10]	12
5.2	Funktionsweise des Dynamic-Wall-Follows	14
5.3	Probleme bei der Parklückenerkennung	15
5.4	Hough-Transformation	16
5.5	Einparkvorgang	17

Tabellenverzeichnis

5.1 Regelparameter	14
------------------------------	----

1 Einleitung

In diesem Kapitel soll die Aufgabenstellung und die Ziele unseres Teams erläutert werden und der Inhalt der folgenden Kapitel kurz zusammengefasst werden.

1.1 Aufgabenstellung

Die Aufgabenstellung des Projektseminars Echtzeitsysteme WS17/18 gliederte sich in drei Teile. Die erste Aufgabe bestand darin, der Einfachheit halber, den sogenannte Wall-Follow-Modus zu Implementieren. Dabei sollte der Roboter eigenständig einen Rundkurs absolvieren können indem er der Wand zu seiner Seite folgt. Wie wir das erste Teilziel „Rundkurs“ später lösen blieb uns überlassen. Der zweite Teil war das Absolvieren des Rundkurses mit Hindernissen denen der Roboter eigenständig ausweichen sollte. Teil drei war ein beliebiges zusätzliches Feature für den Roboter. Hierbei hat sich unser Team für autonomes Einparken entschieden.

1.2 Überblick

Ziel der folgenden Kapitel soll es insbesondere sein unser vorgehen während des Projekts und die Besonderheiten unserer Lösungsansätze zu beschreiben. Auch werden einige Probleme die während des Projekts aufgetreten sind benannt und deren Lösungen beschrieben. Eine genaue Beschreibung der Hardware des Roboters und von ROS sind in den Dokumenten der Einführungsveranstaltungen aufgeführt. In Kapitel 2 wird kurz auf den erwähnten Wall-Follow-Modus und den dafür benötigten Regler eingegangen. Kapitel 3 beschäftigt sich hauptsächlich mit dem Aufgabenteil Rundkurs mit Hindernissen und dem dazu verwendeten Navigation Stack. Im vierten Kapitel wird die Schnittstelle zum Auto beschrieben. Dies umfasst außerdem die Erstellung einer Kennlinie für den Motor und die Lenkung. In Kapitel 5 wird auf das autonome Einparken eingegangen und Kapitel 6 enthält unser Fazit und einen Ausblick.

2 Rundkurs

Im folgenden Kapitel wird auf den ersten Teil der Aufgabenstellung, den Rundkurs ohne Hindernisse, eingegangen. Unser erster Ansatz war, wie im vorherigen Kapitel bereits erwähnt, der sogenannte Wall-Follow-Modus. Dieser basiert auf einem einfachen Regler.

2.1 Modellbildung und Simulation

Um abschätzen zu können in welchem Bereich die Reglerparameter liegen sollten, haben wir uns entschlossen zuerst ein Modell des Roboters in Simulink zu erstellen. Ein einfaches mathematisches Modell für die Reglerauslegung zur Seitenführung ist das Ackermannmodell, welches ein rein kinematisches Modell ist. Es ist mit guter Näherung verwendbar, wenn sich der Schräglauf der Räder vernachlässigen lässt. Das Modell basiert auf den folgenden Gleichungen zur Berechnung der Geschwindigkeit [Len17]:

$$\dot{x} = v \cdot \cos(\varphi) - l \cdot \dot{\varphi} \cdot l \sin(\varphi) \quad (1)$$

$$\dot{y} = v \cdot \sin(\varphi) - l \cdot \dot{\varphi} \cdot l \cos(\varphi) \quad (2)$$

Aus diesen Gleichungen lässt sich, unter anderem durch Linearisierung, die Übertragungsfunktion herleiten. Die vollständige Herleitung ist dem Handout von Dr. Lenz zu entnehmen.

$$G(s) = \frac{v \cdot l_H}{l} \cdot \frac{\frac{v}{l_H} + s}{s^2}. \quad (3)$$

Aus der oben aufgeführten Übertragungsfunktion wurde ein Modell in Simulink erstellt. Zusätzlich wurde noch eine Stellgrößenbegrenzung und ein weiterer Block nach dem Regler eingebaut, welcher dazu dient den Zusammenhang zwischen der Lenkskala des Roboters und dem realen Lenkwinkel herzustellen.

Simuliert wurde zuerst ein PD-Regler mit einem Sollgrößensprung von 0.6 auf 0.8 m. Das Ergebnis ist in Abbildung 2.1 zu sehen. Der Regler ist, wie aus Abbildung 2.1 zu sehen, nicht in der Lage die bleibenden Regelabweichung auszuregeln. Daher wurde im folgenden ein PID-Regler simuliert.

Der PID-Regler zeigt, wie in Abbildung 2.2 zu sehen, ein gutes Verhalten. Alle Parameter wurden mithilfe der Tune-Funktion (und etwas anpassen) von Matlab Simulink bestimmt. Anschließend haben wir die Ergebnisse am realen Roboter überprüft.

2.2 Regler

Die durch die Simulation entstandenen Parameter ließen sich leider nur bedingt in der Realität verwenden. Nach dem Anpassen konnte der Roboter jedoch ohne zu große Überschwingungen der Wand folgen. Ein Problem stellten jedoch noch die Ecken dar.

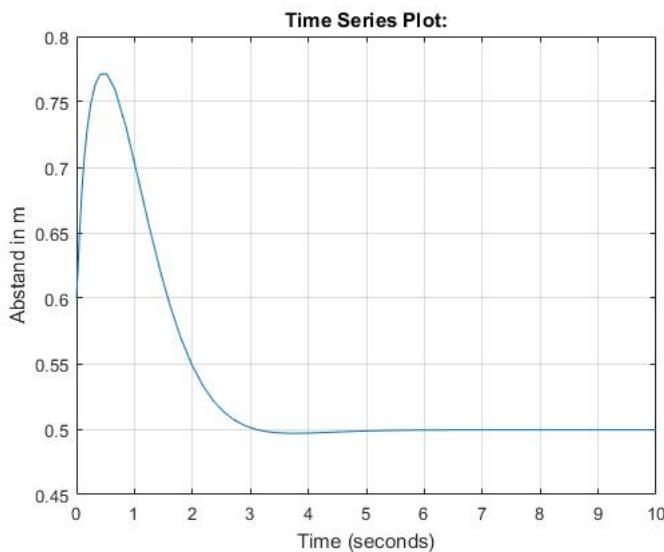


Abbildung 2.1: PD-Regler

Der Regler war nicht in der Lage darauf zufriedenstellend zu reagieren. Daher entschlossen wir uns eine Kurvenerkennung zu entwickeln. Wird eine Ecke erkannt soll der Regler deaktiviert werden und eine einfache Steuerung den Vorgang übernehmen. Nach beenden des Vorgangs wird der Regler wieder aktiviert und das Auto folgt wieder der Wand. Im folgenden sollen nun die beiden von uns erdachten Kurvenerkennungen kurz erklärt werden.

2.2.1 Kurvenerkennung mittels Ultraschallsensor

Die erste Idee bestand darin die Kurve einfach mittels der Ultraschallsensoren zu erkennen. Gibt der Sensor eine hohe Entfernung aus ist keine Wand mehr vorhanden, d.h. eine Ecke wurde erkannt. Dann wird der Regler deaktiviert und ein hoher Lenkwinkel eingestellt, bis der Sensor wieder eine Wand in der Nähe erkennt. Das Ergebnis war jedoch nicht zufriedenstellend. Der Roboter fuhr meistens trotz erkennen der Ecke gegen die Wand. Dies lag zum einen daran, dass der Roboter beim Erkennen der „neuen“ Wand meist schon zu schräg stand und der Regler nicht in der Lage war den Fehler auszuregeln. Ein weiteres Problem ist die Position des Ultraschallsensors, welcher durch die schräge Orientierung des Autos zur Wand nicht die richtige Distanz ausgibt.

Zum lösen des Problems war der erste Gedanke nach dem Erkennen der Ecke einfach eine bestimmte Zeit abzuwarten bis der Regler wieder aktiviert wird. Leider ist die Zeit, die der Roboter benötigt, um die Ecke zu umfahren, nicht immer gleich¹. Daher entstand die Idee das Gyroskop zu verwenden. Die Idee wird im nächsten Abschnitt beschrieben.

2.2.2 Kurvenerkennung mittels Gyroscope

Um das oben beschriebene Problem zu lösen, entschlossen wir uns auf das im Roboter verbaute Gyroscope zurückzugreifen. Zusätzlich werden die Daten des Ultraschallsen-

¹ Um genau zu sein, ist es der Zeitpunkt an dem die Kurve erkannt wurde. Da die Ausrichtung des Roboters zur Wand immer unterschiedlich ist.

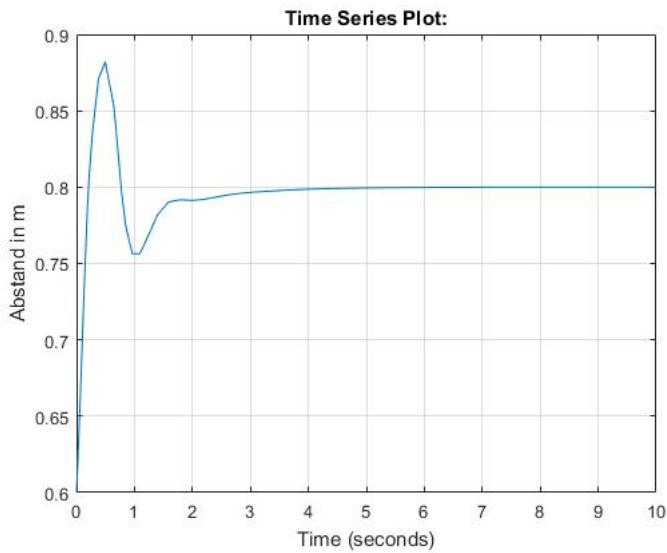


Abbildung 2.2: PID-Regler

sors wegen der enthaltenen Ausreißer Mittelwert gefiltert. Der Vorgang ist grundsätzlich der gleiche wie im vorherigen Abschnitt beschrieben, mit dem einzigen Unterschied, dass die Steuerung ab bzw. der Regler wieder eingeschaltet wird nachdem eine Drehung von 90° durch das Gyroscope gemessen wurde. Dies führte zu einem besseren Ergebnis.

2.3 Ergebnisse

Nach weiterem Optimieren der Reglerparameter wurde die oben erwähnte Kurvenerkennung bzw. Steuerung jedoch obsolet, da der Regler robust genug war den Roboter ohne zusätzliche Hilfe um die Kurve fahren zu lassen. Das Ergebnis war mehr als zufriedenstellend und funktionierte auch noch bei vergleichsweise hohen Geschwindigkeiten. Die verwendeten Reglerparameter sind im folgenden aufgelistet:

- $K_p = 4000$
- $K_d = 1500$

Letztendlich entschieden wir uns aber für den Rundkurs auf den Navigation Stack zu setzen. Dieser wird im folgenden Kapitel genauer beschrieben. Das war im Hinblick auf die Geschwindigkeit eine weniger gute Entscheidung. Allerdings schien uns die Verwendung des Navigation Stacks eine allgemeinere und verlässlichere Lösung zu sein.

3 Rundkurs mit Hindernissen

Im folgenden Kapitel wird auf den zweiten Teil der Aufgabenstellung, den Rundkurs mit Hindernissen, eingegangen. Unser Ansatz ist es dem Roboter mit dem Navigationsstack von ROS die Fähigkeit zu geben sich selbst und Hindernisse in einer Karte zu lokalisieren, um dadurch einen Rundkurs ohne Kollisionen erfolgreich zu bestehen. Abbildung 3.1 zeigt einen Ausschnitt aus Rviz in dem das Auto sich selbst in der vorgegeben statischen Karte des Fachbereiches Echtzeitsysteme der TU Darmstadt lokalisiert, Hindernisse markiert und einen Pfad zu einem ausgewählten Ziel, welches als roter Pfeil dargestellt ist, plant.

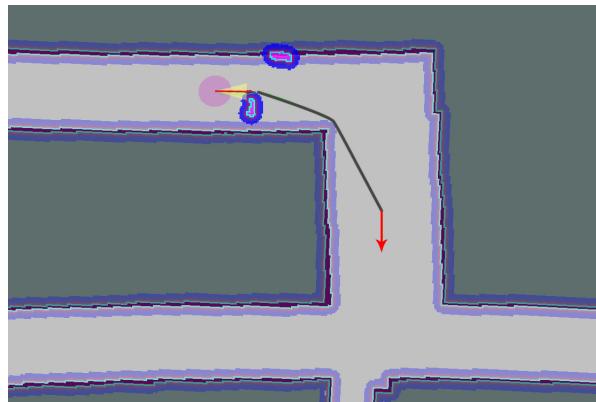


Abbildung 3.1: Pfadplanung

3.1 Das Konzept des Rundkurs mit Hindernissen

Die Kinect-Kamera des Autos ermöglicht über den Infrarot-Sensor Laserscans der Umgebung zu erstellen, was die Nutzung des Navigationstacks für uns möglich macht. Der Navigationstack von ROS benötigt Laserscan-Sensordaten, die Odometrie des Autos und ein gegebenes Ziel, wohin das Auto fahren soll. Als Ergebnis liefert der Navigationstack Geschwindigkeit und Lenkwinkel, welche an das Auto weitergesendet werden. [ros18a]

Abbildung 3.2 zeigt unser Konzept für den Rundkurs mit Hindernissen. Die Knoten auf diesem Bild stellen ROS-Nodes dar, welche jeweils einen von uns geschriebenen Node oder ROS-Paketen entsprechen. Die Nodes **map server**, **costmap 2d**, **global planner**, **teb local planner** gehören zum Navigationstack. Die Aufnahmen der Kinect-Kamera werden gefiltert und als Laserscan an AMCL übergeben. Dieser Knoten steht für adaptive Monte Carlo Lokalisierung (AMCL) und berechnet über Wahrscheinlichkeitsschätzungen die Position des Autos in einer gegeben Karte mithilfe von Laserscans. [amc] Das Paket tf dient zu Umrechnungen der Positionen der Kinect-Farb- und Infrarot-Kamera relativ zur Mitte des Autos, um bei den Schätzungen von AMCL genauere Ergebnisse zu erhalten. [rosa] Die Knoten **goal publisher** und **car node** sind dafür verantwortlich, dass das Auto Teilziele erhält um in einem Rundkurs zu fahren, und dass Motor- und Lenkbefehle an den Mikrocontroller gesendet werden.

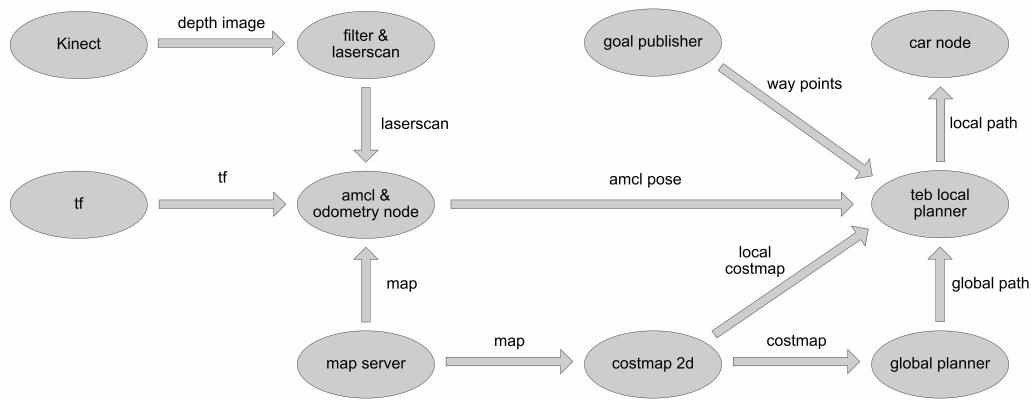


Abbildung 3.2: Konzept

3.2 Der Navigationstack

Der Navigationstack verwaltet die globalen und lokalen Karten der Umgebung des Roboters innerhalb des map servers und berechnet über den globalen und lokalen Planer eine Trajektorie zum gewünschten Ziel. [ros18a]

3.2.1 Mapserver

Für eine Trajektorie von einem Startpunkt des Autos zu einem Ziel werden zunächst Karten der Umgebung des Autos erstellt. Der map server beinhaltet eine statische Karte der Umgebung des Autos. [rosb] Hierfür wurde freundlicherweise im Rahmen dieses Seminars eine Karte der Flure des Fachbereiches Echtzeitssysteme zur Verfügung gestellt. Die costmap 2d berechnet für jede Position der statischen Karte Kosten, je nach ihrer Erreichbarkeit für das Auto. Dies dient dazu, um bei der Trajektorierplanung zu verhindern, dass das Auto gegen Wände oder andere Objekte fährt. [ros18b] Der local map server benutzt den Laserscan der Kinect-Kamera um lokale Hindernisse des Autos auf der lokalen Karte darzustellen. Diese werden ebenfalls als eine local costmap gespeichert, damit Objekte welche sich in der unmittelbaren Nähe des Roboters befinden, und nicht in der statischen Karte eingetragen sind, bei der späteren Planung zu vermeiden. Die Abbildung 3.1 zeigt die Visualisierung des map servers in Rviz. Die Karten bewerten Orte je nach ihrer Erreichbarkeit mit Kosten, wonach Wände in der costmap 2d teuer sind und ebenso Hindernisse der lokalen Karte. Der globale Planer kann mithilfe der Position des Autos und der globalen Karte eine Trajektorie zum ausgesuchten Ziel berechnen. Der teb local planner ist die entscheidende Instanz, welche dem Auto Befehle für die nächste Bewegung sendet, um das ausgewählte Ziel zu erreichen.

3.2.2 Global planner und teb local planner

Der globale Planner berechnet einen auf dem costmap 2d gültigen Pfad zu einem ausgewählten Ziel der Navigation. [glo] Der teb local planner optimiert diese Trajektorie und

fügt bei lokalen Hindernissen Umwege hinzu, um Kollisionen mit Hindernissen zu vermeiden. Als Endergebnis veröffentlicht der teb local planner Geschwindigkeiten und Lenkwinkel, welche dann später vom car node umgewandelt werden in Motor- und Lenkbefehle. [teb]

3.3 AMCL

AMCL verwendet den Laserscan der Kinect, den mapserver vom Navigationstack und die Transformationen von tf um eine Schätzung der aktuellen Position des Roboters und seiner Orientierung zu geben. Die Schätzung kann optimiert werden, wenn zu Beginn eine initiale Position des Roboters an AMCL übermittelt wird. [amc]

3.4 SLAM

Um für unbekannte Gebiete ebenfalls unseren Ansatz verwenden zu können, haben wir uns mit SLAM (simultaneous localization and mapping) auseinander gesetzt und das ROS-Paket **Gmapping** verwendet. Dieses Paket verwendet die Transformationen von tf und den Laserscan der Kinect und die Odometrie-Informationen des in diesem Seminar zur Verfügung gestellten Odometrie-Pakets, um eine Karte der Umgebung des Roboters zu erstellen. [rosc] Die Abbildung 3.3 zeigt einer unserer ersten Versuche die Flure des Fachbereiches aufzunehmen. Nach einigen Optimierungen ist es uns gelungen das Ergebnis von Abbildung 3.4 zu erzielen.

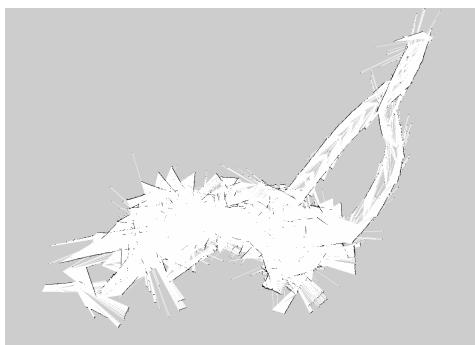


Abbildung 3.3: Ungenaues gmapping

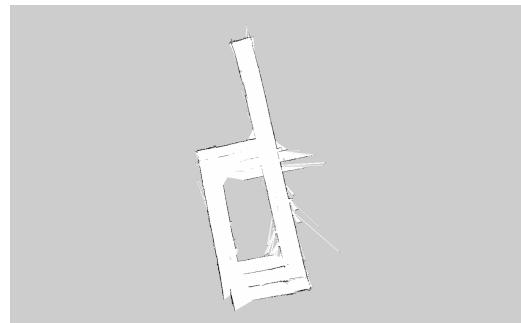


Abbildung 3.4: Optimiertes gmapping

Durch die Halbierung der Abstastfrequenz des Beschleunigungssensor sind die Messwerte hierbei genauer geworden und dies hat eine direkte Auswirkung auf die Berechnung der Odometrie. Folglich haben sich die Ergebnisse von Gmapping hierdurch von Abbildung 3.3 zu Abbildung 3.4 verbessert. Um die Abstastfrequenz der IMU, die den Beschleunigungssensor beinhaltet, zu verringern haben wir ein Shell-Skript geschrieben, um dies bei jedem Start unserer ROS-Knoten durchzuführen.

Listing 3.1: Konfigurationsskript

```
1 rosservice call /uc_bridge/toggle_daq False  
2 rosservice call /uc_bridge/set_imu 4 0 250 1  
3 rosservice call /uc_bridge/toggle_daq True  
4 rosservice call /uc_bridge/get_imu_info
```

4 Schnittstelle zum Auto

Die Ausgabe des Timed-Elastic-Band(Teb) Planners besteht aus einem Lenkwinkel und einer Geschwindigkeit. Diese Werte müssen in Befehle umgewandelt werden, welche der Mikrocontroller verarbeiten kann. Dazu haben wir einen Node geschrieben, der diese Aufgabe übernimmt. Der Hauptteil dieses Nodes ist die Berechnung der Motor- und Lenkwinkelbefehle mit einer zuvor, experimentell ermittelten Kennlinie. Dazu haben wir mehrere ROS-Bags aufgenommen und mit einem Python Skript in CSV-Dateien umgewandelt. Alle zur Laufzeit verfügbaren ROS-Topics werden so in einer CSV-Datei gespeichert. Diese ROS-Topics beinhalten unter anderem alle Sensorwerte, sowie Werte die an den Mikrocontroller gesendet worden sind. Für die weitere Berechnung sind besonders der Hallsensor, das Gyroskop und Befehle für Motor und Lenkung interessant. Genaue Informationen zu den Sensoren und den ROS-Topics sind in den bereitgestellten Dokumentationen verfügbar. Die entsprechenden CSV-Dateien lassen sich einfach in Matlab importieren. Mit Matlab haben wir die Kennlinien für den Motor und die Lenkung berechnet.

4.1 Motorkennlinie

Die Motorkennlinie beschreibt den Zusammenhang zwischen der Geschwindigkeit des Autos und dem Befehl der an den Mikrocontroller gesendet wird. Die Geschwindigkeit des Autos lässt sich mit dem am linken hinteren Rad verbauten Hallsensor berechnen. In dem Rad sind acht Magnete verbaut, sodass der Hallsensor bei einer Umdrehung des Rades acht Zeitintervalle misst. Diese Zeitintervalle können zusammen mit dem Radumfang für die Berechnung der Geschwindigkeit,

$$v = \frac{U}{8t}, \quad (4)$$

genutzt werden. Dabei ist U der Radumfang und t ein gemessenes Zeitintervall. Aus diesen Zeitintervallen lässt sich allerdings nicht unterscheiden, ob das Auto vorwärts oder rückwärts gefahren ist.

Die Motorbefehle, die während der Aufzeichnung der ROS-Bag an das Auto gesendet worden sind, wurden ebenfalls in eine CSV-Datei konvertiert und in Matlab importiert. Dabei haben die Motorbefehle und die Werte des Hallsensors eine verschiedene Zeitachse. Um die entsprechenden Werte verknüpfen zu können ist es notwendig die Zeit zu synchronisieren.

Mit der Annahme, dass das Auto bei einem positiven Motorbefehl vorwärts fährt und rückwärts bei einem negativen Motorbefehl, lässt sich ebenfalls die Fahrtrichtung für jeden Wert des Hallsensors ermitteln. Damit die Annahme gemacht werden kann, sollte das Auto bei der Aufzeichnung der ROS-Bag nur sehr langsam Beschleunigen und nicht abrupt zwischen vorwärts und rückwärts Fahren wechseln. Außerdem können durch die Massenträgheit und große Beschleunigungen falsche Datenpunkte entstehen.

Die so ermittelten Datenpunkte sind in der Grafik 4.1 zu sehen. Aus den so gewonnenen Datenpunkten muss nun eine Kennlinie berechnet werden. Die Problematik dabei besteht darin, dass die Messungen verrauscht waren. Durch die falschen Datenpunkte ist es nicht möglich mit der Methode der kleinsten Quadrate eine Gerade zu ermitteln, welche die Datenpunkte möglichst gut beschreibt. Die Methode der kleinsten Quadrate nutzt quadratische Fehler, sodass die vielen Ausreißer die Kennlinie zu stark beeinflussen würden. Wir haben uns entschieden den RANSAC Algorithmus zu verwenden, um eine genaue Kennlinie zu ermitteln. Der RANSAC Algorithmus ist dafür besonders gut geeignet, da er sehr robust ist bezüglich Ausreißern.

Der Algorithmus nutzt für die Berechnung einer Geraden zwei zufällig aus allen Datenpunkten ausgewählte Punkte. Diese Punkte werden durch eine Gerade beschrieben. Danach wird der Abstand aller anderen Datenpunkte zu der Geraden berechnet und alle Datenpunkte gezählt, deren Abstand zu der Geraden kleiner als ein Schwellenwert ist. Dadurch lässt sich erkennen, wie viele Datenpunkte ungefähr durch die Gerade beschrieben werden können. Dieser Vorgang wird mehrfach wiederholt und die beste Hypothese gespeichert. Am Ende wird aus allen Datenpunkten, die sich im Bereich des Schwellenwerts um die beste Hypothese befinden, die Kennlinie berechnet. Die mit dem RANSAC Algorithmus ermittelte Kennlinie ist ebenfalls in Grafik 4.1 zu sehen.

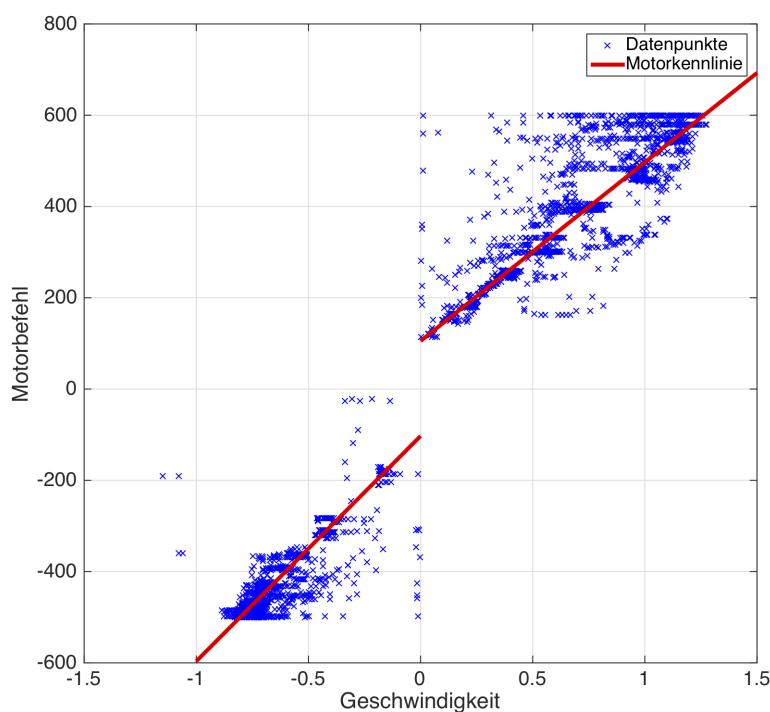


Abbildung 4.1: Ermittelte Datenpunkte und Motorkennlinie

Aus der Grafik ist zu erkennen, dass das Auto nicht direkt losfährt sobald der Motorbefehl größer als Null ist. Erst ab einem Wert von ca. 100-130 bringt der Motor genug Kraft auf, damit sich das Auto in Bewegung setzt. Durch die so ermittelte Kennlinien ist besonders der Anfahrvorgang sichtlich verbessert worden.

4.2 Lenkwinkelkennlinie

Für die Berechnung der Lenkwinkelkennlinie haben wir ROS-Bags aufgenommen, bei denen das Auto mit konstanter Geschwindigkeit im Kreis fährt. Dabei wurden Kreise mit verschiedenen Radien (Lenkwinkelbefehlen) gefahren.

Das Vorgehen zum Ermitteln der Lenkwinkelkennlinie ist ähnlich zu dem der Motorkennlinie. Dazu ist es ebenfalls notwendig die Geschwindigkeit aus den Werten des Hallsensors zu berechnen. Der Lenkwinkel lässt sich aus Zusammenhängen des Ackermannmodells berechnen [Len17]. Für die Berechnung wird die Drehgeschwindigkeit, die mit dem Gyro gemessen werden kann, des Autos um die z-Achse benötigt. Damit die Sensorwerte verwendet werden können muss auch die Zeitachse des Gyrosensors mit der des Hallsensors synchronisiert werden.

Der Lenkwinkel lässt sich nun mit:

$$\dot{\phi}_L = \arctan \frac{\dot{\phi}_k l}{v} \quad (5)$$

berechnen. Dabei ist $\dot{\phi}_k$ die Änderung des Kurswinkels. Diese Winkelgeschwindigkeit wurde mit dem Gyro gemessen. v ist die zuvor berechnete Geschwindigkeit und l der Radabstand. In der Grafik 4.2 sind die berechneten Lenkwinkel mit den Lenkwinkelbefehlen aus der ROS-Bag dargestellt. Die Lenkwinkelkennlinie wurde ebenfalls mit dem RANSAC-Algorithmus ermittelt.

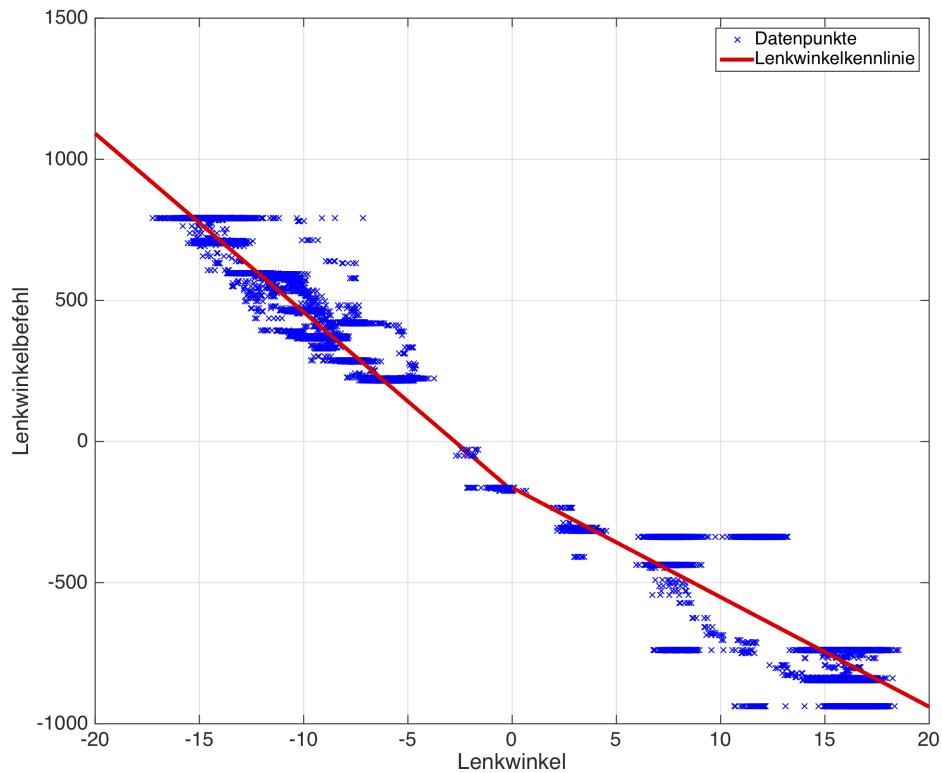


Abbildung 4.2: Ermittelte Datenpunkte und Lenkwinkelkennlinie

Aus der Kennlinie lässt sich erkennen, dass für die Geradeausfahrt etwa ein Lenkwinkelbefehl von 200 benötigt wird. Diesen Offset in der Lenkung haben wir bereits zuvor am Auto festgestellt. Die so ermittelten Kennlinien haben wir in einem Node implementiert. Die Fahrdynamik des Autos hat sich dadurch sichtlich verbessert. Dies ist besonders beim Anfahren und bei der Geradeausfahrt zu bemerken.

5 Einparken

Das Einparken ist das zweite, große Themengebiet mit dem wir uns beschäftigt haben und Thema dieses Kapitels. Wie bei jedem echten Auto auch, lässt sich der Einparkvorgang auch bei unserem Roboter durch das Rückwärtsfahren zweier Kreise beschreiben. Abbildung 5.1 sollte den Vorgang etwas deutlicher machen.

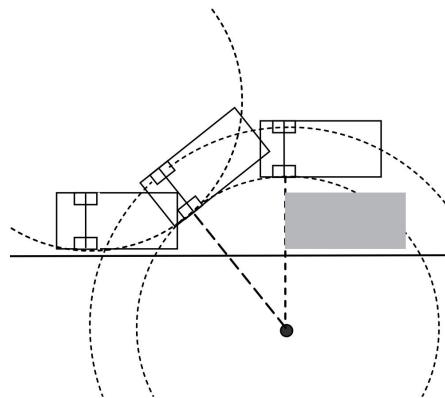


Abbildung 5.1: Der Einparkvorgang [Roj10]

Auf dessen genaue Realisierung wird später eingegangen. Die größte Schwierigkeit beim Einparken mit den uns zur Verfügung gestellten Robotern ist die genaue Positionierung des Roboters neben der Parklücke. Aufgrund der beschränkten Sensorik muss der Roboter "blind" parken, er hat keinerlei Möglichkeiten, die Umgebung hinter ihm genau zu observieren und auf diese entsprechend zu reagieren. Der Einparkvorgang ist also statisch, sprich der Roboter muss davor in einer genau bestimmten Position zur Parklücke stehen, was in unserem Fall bedeutet, möglichst parallel neben dem Auto zu stehen, hinter dem man einparken will. Wie dies zu erreichen ist, wird weiter unten eingegangen.

Der Einparkvorgang unserer Implementierung lässt sich grob in folgende Schritte unterteilen:

- Dynamic-Wall-Follow
- Erkennung und Bewertung der Parklücke
- Genaue Positionierung des Autos
- Starten des Einparkvorgangs
- Positions korrektur

Diese werden nun in den folgenden Unterkapiteln erläutert.

5.1 Dynamic Wall-Follow im Detail

Nach dem Starten des Park-Prozesses macht der Roboter zunächst nichts weiteres als stumpf der rechten Wand in einem festen Abstand zu folgen. Dies wird mittels eines unten erläuternden PID-Reglers und dem rechten Ultraschallsensor realisiert. Hier ist es sehr wichtig, dass der Regler so genau wie möglich arbeitet, um so gut es geht zu vermeiden, dass der Roboter in Schlangenlinien fährt, da dies sowohl die Messung, als auch die Erkennung möglicher Parklücken erschwert, teilweise sogar zu völlig falschen Ergebnissen führt.

Ein normaler Wall-Follow-Algorithmus reicht hier aber nicht aus, da die Wand, die der Roboter folgt, nie exakt gerade ist, da sich zum Beispiel zwischen dem Roboter und der Wand andere Hindernisse befinden, wie beispielsweise andere parkende Autos. Ein normaler Wall-Follow-Algorithmus würde bei jedem Hindernis nach links ausweichen, sprich von der Wand weg, um die vorgegebene Distanz zu halten. Das ist natürlich für das Einparken sehr unpraktisch, da der Roboter hier recht nah an den anderen parkenden Autos fahren muss.

Unsere Lösung war es, den statischen Wall-Follow-Algorithmus dynamisch zu machen. Das heißt er erkennt plötzlich auftretende Hindernisse, und versucht dann zu diesem die bereits bestehende Distanz zu halten, so werden die oben beschriebenen Ausweichbewegungen des Roboters vermieden (siehe Abbildung 5.2). Die Implementierung dessen hat uns aber vor große Herausforderungen gestellt, da der rechte Ultraschallsensor sowohl zur Distanzhaltung vom PID-Regler verwendet wird, als auch zur Erkennung von Hindernissen und wie später noch erläutert wird, zur Erkennung und Messung der Parklücken. Hier musste viel experimentiert werden und der Algorithmus erforderte sehr viel Feintuning. Um bei Hindernissen den PID-Regler nicht sofort zum Auslenken zu bringen, wurde ein Tick-System eingeführt, das sicherstellt, dass der PID-Regler nur alle 3 Ticks arbeitet, während die Hindernis- und Parklückenerkennung immer aktiv ist. Unser gesamter Park-Prozess arbeitet intern mit 10 Ticks pro Sekunde.

5.2 Verwendeter PID-Regler

Zur Regelung der Lenkung des Roboters wurde ein PID-Regler verwendet, da ein PD-Regler nicht ausreichend zufriedenstellend arbeitete. Wie oben beschrieben ist eine stabile und robuste Regelung sehr wichtig für den eigentlichen Einparkprozess. Das System muss eine kurze Einschwingzeit und eine kleine Regelabweichung besitzen, außerdem muss es schnell auf die sich ständig ändernden Sollwerte vom Dynamic-Wall-Follow-Algorithmus reagieren können. Besonders letzteres war das Hauptproblem beim Reglerentwurf.

Die Parameter des PID-Reglers wurden experimentell bestimmt. Zuerst wurde der P-Anteil so eingestellt, dass der Istwert so langsam wie möglich, aber immer noch ausreichend schnell für den Dynamic-Wall-Follow-Algorithmus den Sollwert annimmt. Anschließend wurde der I-Anteil erhöht bis Überschwingungen des Reglers von bis zu 20%

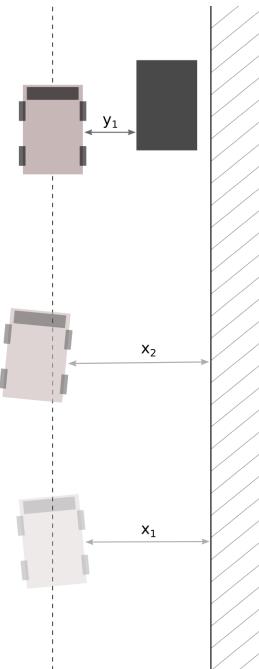


Abbildung 5.2: Funktionsweise des Dynamic-Wall-Follows

beobachtet werden konnten. Zuletzt wurde der D-Anteil so weit vergrößert, bis die durch den I-Anteil erzeugten Überschwingungen ausreichend gedämpft wurden.

Die Stellgröße des Reglers wurde auf ± 700 beschränkt, um die Servolenkung des Roboters nicht zu überlasten. Zur Vermeidung eines Überlaufs der Regler-Internen Variablen wurde ein Anti-Windup Schutz in den Regler implementiert.

K_p	K_i	K_d
70	5	80

Tabelle 5.1: Regelparameter

5.3 Parklückenerkennung

Bis jetzt wurde erläutert wie der Roboter möglichst präzise der rechten Wand folgt und trotz etwaiger Hindernisse, wie zum Beispiel anderer parkender Autos, immer einen festen Abstand zur Wand hält. Nun stellt sich die Frage, wie der Roboter eine Parklücke erkennt, und diese auch entsprechend ihrer Daten bewertet. Zum Beispiel sollte der Roboter nicht versuchen in jede beliebige Lücke zwischen zwei Hindernissen zu fahren, stattdessen soll weiter nach einer ausreichend großen Lücke gesucht werden, die sich zum einparken eignet. Dafür haben wir zwei Ansätze verfolgt, die Erkennung per Ultraschall-Sensor und die Hough-Transformation, die nun im Folgenden erläutert werden sollen.

5.3.1 Ultraschall

Die Erkennung der Parklücke mit dem Ultraschall-Sensor ist simpel, und kann einfach in den Dynamic Wall-Follow Algorithmus eingebaut werden. Ändert sich die Messung des rechten Ultraschall-Sensors schlagartig über einen bestimmten Schwellwert T , so ist der Roboter je nachdem, ob die Distanz sich vergößert oder verkleinert hat, gerade an einem Hindernis vorbei gefahren (Distanz ist nun wieder größer) oder fährt jetzt neben einem Hindernis (Distanz ist nun kleiner). So lassen sich Beginn (vorbeifahren an einem Hindernis) und das Ende (neues Hindernis erkannt) feststellen. Abbildung 5.3 veranschaulicht den Ansatz und zeigt wie der Roboter eine Parklücke erkennen kann, natürlich muss diese noch entsprechend ihrer Parameter bewertet werden. Die Distanz zwischen den beiden Ereignissen wird durch den Hall-Sensor im Reifen des Roboters gemessen. Anhand dieser und der Tiefe der Parklücke wird entschieden, ob die Parklücke für den Roboter groß genug ist, oder ob weiter gesucht werden muss.

Die Idee ist sehr simpel, aber dennoch erstaunlich präzise, vorausgesetzt das vorher besprochene Dynamic-Wall Follow arbeitet ausreichend genau. Fährt der Roboter bei zu unpräziser Regelung Schlangenlinien, so ändert sich entsprechend auch die Distanz zur Wand und es kann passieren, das falls diese ungewollte Distanzänderung größer als der Schwellwert T ist, der Roboter fälschlicherweise ein Hindernis erkennt oder denkt er wäre bereits an einem vorbei gefahren. Dreh- und Angelpunkt dieser Erkennungs methode ist also ein robuster Regler für die Geradeausfahrt, der Roboter sollte sich immer so parallel wie möglich zur Wand befinden. Schlangenlinien bei der Fahrt des Roboters sollten so gut wie möglich vermieden werden.

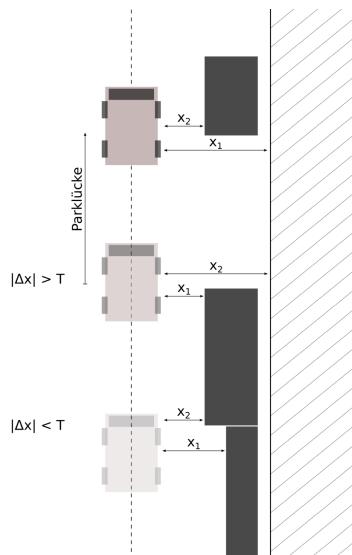


Abbildung 5.3: Probleme bei der Parklückenerkennung

5.3.2 Hough Transform

Ein anderer, deutlich komplexerer Ansatz zur Parklückenerkennung ist die Verwendung der Kinect-Kamera, bzw. des aus ihrem Tiefenbild gewonnenen Laserscans. Mit Hilfe der

Hough-Transformation ist es möglich, aus dem Laserscan Linien im Field of View des Roboters zu extrahieren. Anschließend werden die gewonnenen Linien zu geometrischen Figuren gruppiert, in diesem Falle Rechtecken, die dann potentielle Parklücken darstellen können. Ein Beispiel, wie das Ergebnis der Hough-Transformation angewendet auf einen Laserscan des Roboters im Gang des HBI aussieht, ist in Abbildung 5.4 zu sehen.

Wir haben zwei Implementierungen der Hough-Transformation getestet, eine selbst entwickelte und eine in OpenCV enthaltene. Letztere arbeitete zufriedenstellender und performanter, daher wurde diese Implementierung für weitere Tests verwendet. Nach mehreren Tests in verschiedenen Umgebungen und vielem Feintuning an den Algorithmen funktionierte die Erkennung von Wand und potentiellen Parklücken zufriedenstellend. Auf unserem Roboter konnte der Algorithmus circa 8-10 FPS liefern, was genug für eine simple Navigation wäre, allerdings hatte der Algorithmus mit kleinen Objekten unter 1 Meter Probleme. Hier würde weitere Bildverarbeitung den Algorithmus wahrscheinlich noch zuverlässiger und robuster machen, jedoch würde dies eine Verschlechterung der Performance mit sich bringen.

Wir entschieden uns nach diesen Experimenten gegen den Einsatz der Hough-Transformation bei der Parklückenerkennung, da bei weiterer Bildverarbeitung mit weiteren Performance-Einbußen zu rechnen ist, und die 8-10 FPS des Grundalgoritmus keinen ausreichend großen Puffer dafür bereitstellen. Daher wurde die Parklückenerkennung per Ultraschallsensor im finalen Code verwendet.

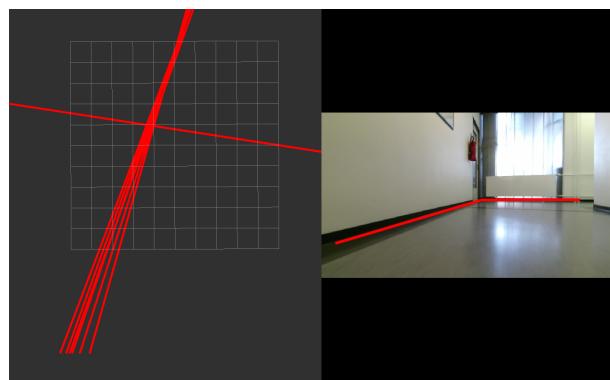


Abbildung 5.4: Hough-Transformation

5.4 Der eigentliche Einparkvorgang

Steht der Roboter nun parallel zu dem Hindernis hinter dem geparkt werden soll, so ist der Einparkvorgang ganz einfach. Die Lenkung wird ganz nach rechts eingeschlagen und rückwärts gefahren, bis sich der Roboter um θ -Grad gedreht hat, nun wird die Lenkung nach links ganz eingeschlagen und wieder rückwärts gefahren, bis der Roboter parallel zur Wand steht, danach stoppt er. Nun fährt er noch soweit vor, bis er in der Mitte der Parklücke steht.

Die Berechnung des Parameters θ ist in Abbildung 5.5 dargestellt. Die Konstante R im Bild ist der minimale Radius, den der Roboter bei voll eingeschlagener Lenkung fahren kann. Der Parameter a berechnet sich aus dem aktuellen Ultraschallsensorwertes und der gemessenen Parklückentiefe. b ist eine Konstante, die die Distanz der hinteren Achse zur vordersten Spitze des Fahrzeugs darstellt. Die restlichen Schritte sollten durch die Abbildung selbsterklärend sein. [Roj10]

Nun steht der Roboter in der Parklücke, aber wie weit er noch nach vorne fahren muss, um in der Mitte der Parklücke zu stehen ist noch unklar. Dazu wird die Distanz zum vorderen Hindernis mit dem vorderen Ultraschallsensor gemessen. Die gewünschte Distanz, nennen wir sie x , wird wie folgt berechnet:

$$x = \frac{(s - l)}{2.0} \quad (6)$$

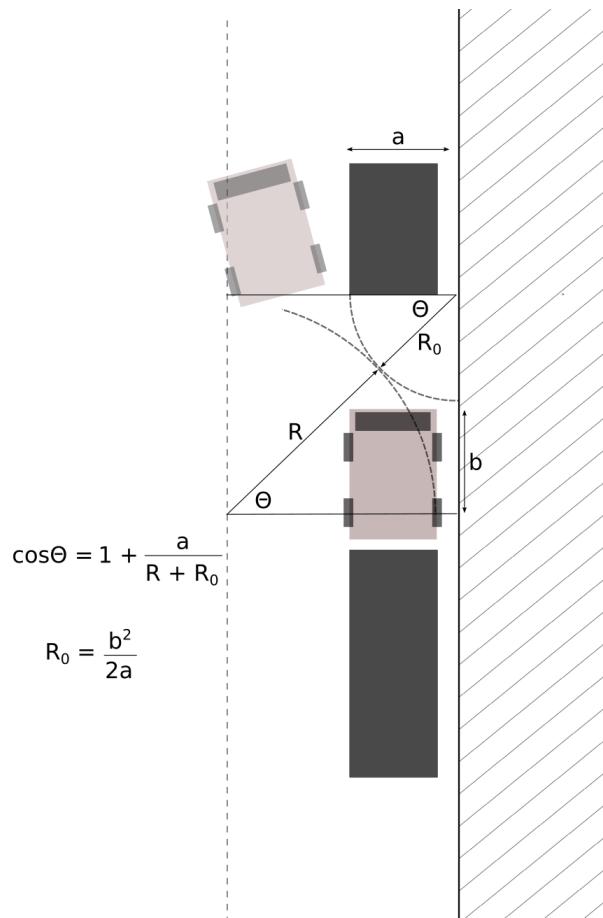


Abbildung 5.5: Einparkvorgang

Wobei s die gemessene Parklückentiefe und l die Länge des Roboters ist. Nun fährt der Roboter solange vorwärts, bis die Distanz zum vorderen Auto kleiner gleich der gewünschten Distanz x ist. Dabei wird für eine optimale Trajektorie der oben erwähnte PID-Regler wiederverwendet. Nun ist der Roboter fertig eingeparkt.

Ausparken ist aufgrund mangelnder Sensorik hinten am Roboter nicht ohne weiteres möglich, nur wenn die Parklückenlänge vom Einparken noch bekannt ist, anhand dieser könnte der Roboter mit dem vorderen Ultraschallsensor ungefähr abschätzen wo er sich in der Parklücke befindet, allerdings darf sich die Parklücke so natürlich nicht ändern. Aufgrund dieser Einschränkungen haben wir uns gegen eine Implementierung des Ausparkens entschieden, allerdings lässt sich diese, nach dem erweitern der Sensorik, einfach hinzufügen, da wir stets darauf geachtet haben das unser gesamter Code sehr einfach erweiterbar ist. Es müsste nur ein entsprechender Zustand hinzugefügt werden.

6 Fazit und Ausblick

6.1 Rundkurs

Wie bereits aus den vorherigen Kapiteln zu erkennen ist, haben wir uns dafür entschieden den Navigationstack zu verwenden. Der Navigationsack ermöglicht eine sehr allgemeine Lösung und es ist möglich den Rundkurs ohne und mit Hindernissen damit zu bewältigen.

Außerdem kann der Navigationstack als Basis für weitere Aufgaben genutzt werden. Das Auto benötigt lediglich ein Ziel auf der Karte und ist dann in der Lage selbstständig zu diesem Ziel zu fahren.

Damit der Navigation Stack gut funktioniert ist es notwendig sehr viele Parameter anzupassen. Dies nimmt viel Zeit in Anspruch und oft wird der Effekt nur durch Ausprobieren sichtbar.

Wir haben uns nach kurzer Zeit dazu entschieden den Navigationstack mit dem Teb Planner zu erweitern. Dadurch wurden die Ergebnisse verbessert, da der Teb Planner speziell für Fahrzeuge gut geeignet ist.

Ein Problem ist, dass das Auto in bestimmten Situationen Hindernisse nicht richtig erkennt. Dieser Effekt ist uns besonders deutlich bei der Kurvenfahrt aufgefallen. Dabei erkennt das Auto erst sehr spät Hindernisse die sich im Kurveninneren befinden. Das Problem ist auf die Kinect Kamera zurück zu führen. Das Sichtfenster der Kinect Kamera ist nicht sehr groß und die Hindernisse werden dadurch nicht vom Auto gesehen. Dieses Problem kann durch die Verwendung von anderen Sensoren behoben werden. Das Auto verfügt außerdem über Ultraschallsensoren. Diese Sensoren könnten den Bereich außerhalb des Sichtfensters der Kinect Kamera abtasten. Dies haben wir versucht, allerdings fehlte uns die Zeit, um eine zufriedenstellende Lösung zu erzielen.

Ein weiteres Problem das uns während der Inbetriebnahme des Navigationstacks aufgefallen ist war, das auftreten von „Geisterobjekten“ während des fahrens des Rundkurses. Dabei entstanden aufgrund von Reflexionen die der Kinect Sensor aufzeichnet auch Punkte im Laserscan. Diese wiederum wurden von der Local Coastmap als Objekte interpretiert und daher mit Kosten versehen. Dies führte dazu das der Roboter ausgebremst wurde und versuchte diesen, eigentlich nicht existenten Objekten, auszuweichen. Er musste teils ausweichen und das Objekt umständlich umfahren, im schlimmsten Fall war eine weiter fahrt nicht möglich. Nach einigem Testen lies sich das Problem auf drei Punkte zurückführen.

Die erste Ursache hängt mit der Reichweite des Sensors zusammen. Schaut der Sensor auf ein Gebiet indem sich keine Objekte bzw. Wände innerhalb des maximalen Sichtbereichs befinden (dies tritt vor allem in den längeren Fluren auf) sorgt dies vermehrt für Fehler bei der Detektion. Dieses Problem lies sich, zumindest teilweise, durch leichtes ausrichten der Kamera Richtung Boden lösen.

Die zweite Ursache ist Glas. Glas führt aufgrund der aktiven Infrarot Beleuchtung der Kinectkamera zu erheblichen Reflexionen bzw. Spiegelungen. Dieses Problem lies sich

nur doch abdecken von Fenstern und Großen Glasflächen etc. lösen. Der dritte Grund sind allgemeine Störungen bzw. Reflexionen ohne spezielle Ursache. Um die Störungen zu reduzieren verwendeten wir den zur Verfügung gestellten Filter Node² für das Tiefenbild der Kinect. Das Ergebnis war zwar bemerkbar aber nicht wirklich zufriedenstellend. Ein Ansatz zur Verbesserung der Performance ist zum Beispiel das erstellen eines eignen, besseren Filters um Reflexionen im Laserscan auf ein minimum zu beschränken. Hierzu fehlten uns jedoch leider die Zeit. Wir reduzierten den Einfluss durch anpassen der Parameter der Coastmap. Insbesondere ist es wichtig darauf zu achten das die „Geisterobjekte“ in der Local Coastmap nicht auf die Global Coastmap übernommen werden. Dadurch werden sie relativ schnell wieder herausgelöscht (bei erneutem einsehen des „blockierten“ Bereichs) und sorgen nicht, spätestens bei der zweiten Runde, für weitere Probleme.

In Kapitel 4 haben wir erklärt, wie wir die Kennlinie für die Lenkung und den Motor aufgenommen haben. Diese beiden Kennlinien haben eine deutliche Verbesserung gebracht. Die Fahrmanöver sind dadurch sichtbar flüssiger geworden.

Gegebenenfalls könnten für die Berechnung der Kennlinie noch weitere ROS-Bags verwendet werden. Bei der Aufnahme der verwendeten ROS-Bags waren wir vom Platz her beschränkt. Würden die ROS-Bags z.B. in einer Turnhalle aufgenommen, wären die Ergebnisse vermutlich genauer und die Anzahl der falschen Messwerte geringer.

6.2 Einparken

6.2.1 Kalman-Filter

Der größte Schwachpunkt des Roboters ist seine begrenzte Anzahl an Sensoren und deren Genauigkeit, insbesondere Rauschen ist hier ein Problem. Eine Ursache für die mangelnde Genauigkeit ist das viele Daten von den Sensoren ohne Nachbearbeitung und Filterung verwendet werden. Ein möglicher Ansatz ohne Hardware-Änderungen wäre ein Kalman-Filter, der die Daten von den verfügbaren Sensoren zusammen betrachtet und daraus eine genauere Position des Roboters im Raum ermöglicht, als es beispielsweise das zur Verfügung gestellte Odometrie-Paket tut. Dies würde helfen die Trajektorie des Roboters parallel zur Wand zu verbessern und damit auch die Erfolgsquote des Einparkvorgangs, da dieser wie in 4.5 erwähnt stark von der Startposition abhängt, und diese bei ungenauer Steuerung teils stark schwanken kann.

6.2.2 Verbesserungen des Gyroskops

Für ein erfolgreiches autonomes parken ist eine verlässliche Messung der Rotation des Roboters unabdingbar. Während dieses Projektes ist uns ein Drift der Gyroskop-Messwerte aufgefallen, selbst wenn der Roboter still steht. Aus Zeitgründen haben wir einen einfachen Schwellwert implementiert, der geringe Veränderungen der Gyroskop-Messwerte einfach verwirft, allerdings auf Kosten der Genauigkeit bei sehr langsamem

² enthalten im Package pses kinect utilities

Bewegungen. Eine besser Lösung wäre ein Tiepass direkt auf dem Microcontroller wo die Abtastrate der Gyroskop-Daten bekannt ist. Auch der oben erwähnte Kalman-Filter würde hier helfen, am besten wäre eine Kombination aus beiden Anäten.

6.2.3 Mehr Sensorik

Schon der Wall-Follow Algorithmus als auch das Parken selbst würde stark von mehr Ultraschall-Senoren profitieren. Ein zusätzlicher Ultraschall-Sensor an der Seite des Roboters würde helfen einen festen Winkel und Abstand zur Wand zu halten. Außerdem würde er das Erkennen der Parklücken deutlich vereinfachen und verbessern. Auch hinten am Roboter wären weitere Sensoren sinnvoll, da der Roboter aktuell "blind" parkt, sprich die Parklücke wird ausgemessen und darf sich danach nicht mehr verändern. Läuft nun beispielsweise ein Kind die Parklücke, so könnte der Roboter nicht mehr reagieren. Generell würden mehr Sensoren am Roboter die Kollisionswahrscheinlichkeit des Roboters verringern.

6.3 Allgemein

Das Projektseminar hat uns sehr gut gefallen, besonders die freie Aufgabenstellung lässt sich gut mit den eigenen Interessen kombinieren. Wir haben dabei den Umgang mit ROS gelernt und uns im Team zu Organisieren. Zur Absprache und Organisation haben wir Trello, Slack und Github genutzt.

ROS ist ein sehr umfassendes Software-Framework und die erlernten Methoden und Vorgehensweisen lassen sich dank des Modularen Aufbaus auch in anderen Projekten nutzen.

Außerdem haben wir unsere Programmierkenntnisse, besonders in Kombination mit ROS, verbessert.

Literatur

- [amc] <http://wiki.ros.org/amcl>
- [glo] http://wiki.ros.org/global_planner
- [Len17] LENZ, Eric: *Handout*. 2017
- [Roj10] ROJAS, Raul: *Die Weltformel: Wie man ein Auto einparkt.* http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/Weltformel.pdf. Version: 2010
- [rosa] <http://wiki.ros.org/tf>
- [rosb] http://wiki.ros.org/map_server?distro=kinetic
- [rosc] <http://wiki.ros.org/gmapping>
- [ros18a] <http://wiki.ros.org/navigation>
- [ros18b] http://wiki.ros.org/costmap_2d
- [teb] http://wiki.ros.org/teb_local_planner

A Erster Anhang

Im Anhang dieses Dokumentes befinden sich der Code der Gruppe TUrtlES, die Endpräsentation und das Vorstellungs-Video.