Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Ronja Güldenring

Bahnplanung, Selbstlokalisierung und Hindernisumfahrung eines mobilen Roboters

Fakultät Technik und Informatik
Department Maschinenbau und Produktion

Faculty of Engineering and Computer Science
Department of Mechanical Engineering and Production Management

Ronja Güldenring

Bahnplanung, Selbstlokalisierung und Hindernis-umfahrung eines mobilen Roboters

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Maschinenbau, Entwicklung und Konstruktion
am Department Maschinenbau und Produktion
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Erstprüfer/in: Herr Prof. Dr.-Ing. Frischgesell
Zweitprüfer/in : Herr Prof. Dr.-Ing. Schulz

Abgabedatum: 25.09.2015

Zusammenfassung

Ronja Güldenring

Thema der Bachelorthesis

Bahnplanung, Selbstlokalisierung und Hindernisumfahrung eines mobilen Roboters

Stichworte

mobile Robotik, Steuerung, modellbasierte Regelung, ICP-Algorithmus, Scanmatching, Potentialfeldmethode, Lokalisierung, MicroAutoBox, dSpace, Simulink/Matlab, Gitternetzkarte, Laserscanner, Differentialantrieb

Kurzzusammenfassung

Im Rahmen dieser Bachelorarbeit wurde die Steuerung und Regelung eines mobilen Roboters an einem Echtzeitsystem realisiert. Ziel war es, dass der Roboter einen Weg von Start- zu Zielpunkt ermittelt und diesen kollisionsfrei abfährt. Dabei befindet sich der Roboter in einer bekannten Umgebung – soll aber auch unbekannten, neuen Objekten ausweichen, sofern diese auf dem geplanten Pfad erscheinen.

Ronja Güldenring

Title of the paper

Path Planning, self-localisation and navigating around obstacles of a mobile robot

Keywords

mobile robotics, controlling, model-based control circuit, ICP-algorithm, scanmatching, distance transform method, localisation, microAutoBox, dSpace, Simulink/Matlab, grid map, laserscanner, differential drive

Abstract

This Bachelor thesis deals with the controlling and regulation of a mobile robot, integrated in a real-time system. The main objective is to make the robot calculate a path between a start position and a specified target position which it subsequently navigates without any collision. Even though the mobile robot manoeuvres through a familiar environment, it should be able to detect unknown obstacles and avoid them automatically.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Formelzeichenverzeichnis

B	Breite des Roboters
$B1$	Abstand zwischen den Rädern
D	Verschiebbare und verdrehbare Punktwolke
$\overline{D}$	Punktenwolkenschwerpunkt der Punktwolke D
f_{rising}	Frequenz von einer Schrittflanke zur nächsten Schrittflanke bei dem Encodersignal
i	Übersetzungsverhältnis von Motor zu Rad
M	Feste Punkwolke
$\overline{M}$	Punktenwolkenschwerpunkt der Punktwolke M
N_D	Anzahl der Punkte der Punktwolke D
$N_{Impulse}$	Anzahl der Impulse pro Umdrehung des Motors
N_M	Anzahl der Punkte der Punktwolke M
n_{max}	Maximale Drehzahl der Motoren
p_i	Punkt an der Stelle i des Vektors i
$p_{i,neu}$	Iterativ geglätteter Punkt
PWM	Pulsweitenverhältnis: Pulsweite/Periode
r	Radius der Räder des Roboters
R	Rotationsmatrix, die aus dem Scanmatching resultiert
s_{diff}	Wegedifferenz
t	Translationsmatrix, die aus dem Scanmatching resultiert
t_{aus}	Zeit, während das PWM-Signal null ist
t_{ein}	Zeit, während das PWM-Signal eins ist
T	Transformationsmatrix
U_{ein}	An den Motoren anliegende Spannung
U_m	Mittlere Spannung über das PWM-Signal
v_{diff}	Translationsgeschwindigkeitsdifferenz
v_{ges}	Translationsgeschwindigkeit des Roboters im Drehpunkt
v_L	Translationsgeschwindigkeit des linken Rades des Roboters
v_{max}	Translationsgeschwindigkeitsbegrenzung
v_R	Translationsgeschwindigkeit des rechten Rades des Roboters
W	Korrelationsmatrix
x_i	Position des Roboters in x-Richtung zum Zeitpunkt i
x_{ist}	Ist-Position des Roboters in x-Richtung
x_{soll}	Soll-Position des Roboters in x-Richtung
y_i	Position des Roboters in y-Richtung zum Zeitpunkt i
y_{ist}	Ist-Position des Roboters in y-Richtung
y_{soll}	Soll-Position des Roboters in y-Richtung

α	Gewichtungskoeffizient
Δt	Zeitdifferenz
θ_{diff}	Winkeldifferenz
θ_{ist}	Ist-Orientierung
θ_{soll}	Soll-Orientierung
φ_i	Drehwinkel des Roboters zum Zeitpunkt i
ω	Winkelgeschwindigkeit des Roboters um den Momentanpol
ω_{diff}	Winkelgeschwindigkeitsdifferenz
ω_L	Winkelgeschwindigkeit des linken Rades
ω_{max}	Rotationsgeschwindigkeitsbegrenzung
ω_R	Winkelgeschwindigkeit des rechten Rades

1 Einleitung

1.1 Motivation

Der Wirtschaftssektor der Robotertechnik weist in den letzten Jahren ein sehr starkes Wachstum auf. Zunächst lag der Fokus auf dem Einsatz der Industrierobotik. Der Industrieroboter arbeitet in einem klar definierten Umfeld und ist meistens abgeschirmt von unbekannten äußeren Einflüssen – insbesondere Menschen. Er übernimmt oftmals Fließband- oder Präzisionsarbeit (Bsp.: Schweißroboter). Mittlerweile hat sich auch der Bereich der mobilen Robotik enorm weiterentwickelt und spezialisiert sich immer mehr. Serviceroboter interagieren oft in sich verändernden Umgebungen, in denen Menschen vorkommen, und müssen dementsprechend auf diverse Situationen vorbereitet sein. Sie übernehmen schon viele verantwortungsvolle Aufgaben im Dienstleistungssektor, z.B: Reinigung von Gebäuden, Unterstützung bei chirurgischen Operationen, Transport von Speisen oder Medikamenten etc.. Auch in privaten Haushalten finden mobile Roboter als Staubsauger oder Rasenmäher ihre Anwendung. Serviceroboter werden in Zukunft immer mehr Aufgaben zur Sicherstellung eines menschengerechten Arbeitsumfeldes übernehmen – eine Branche, die von zunehmendem Wettbewerb bestimmt ist. Die Motivation für diese Bachelorarbeit findet ihren Ursprung in der starken Weiterentwicklung von Servicerobotern. Es werden bekannte Konzepte und Algorithmen untersucht und an einem realen Roboter realisiert.

Trotz der beeindruckenden Innovationen, die der Robotersektor mit sich bringt, sollte auf potentielle Risiken, Gefahren und den Missbrauch von Robotersystemen hingewiesen werden. Je stärker Roboter im menschlichen Umfeld eingesetzt werden, desto höher wird das Risiko, dass unter Roboter-Mensch-Interaktionen Menschen zu Schaden kommen können. Im Folgenden werden zwei extreme Beispiele zur Veranschaulichung vorgestellt. Bei dem ersten Beispiel handelt es sich um den zunehmenden Einsatz von autonomen Waffensystemen oder gar autonomen Kampfroboter. Die präzise Kriegsführung und programmatische Entscheidungsfindung wirft starke ethische Bedenken auf. Andererseits sei ein erschreckendes Beispiel, den Bereich menschlicher Emotionen betreffend, erwähnt. In New York sind Sexpuppen mit erstaunlich realitätsnahen menschlichen Zügen und Verhaltensweisen in Entwicklung [nytimes 2015].

1.2 Aufgabenstellung

Diese Bachelorarbeit thematisiert die Steuerung eines mobilen Roboters in einer bekannten Umgebung. Ziel ist es, dass der Roboter von A nach B fahren kann und dabei Hindernissen ausweicht.

Die Bachelorarbeit teilt sich in zwei Teilbereiche: Simulation der Roboterfahrt und tatsächliche Steuerung eines mobilen Roboters, der mit einer dSpace-Box und mit zwei angetriebenen Rädern ausgestattet ist. Eine Sensorauswahl muss für den Roboter noch getroffen und in beiden Teilbereichen eingebunden werden.
Die Simulation der Roboterfahrt wird in Matlab/Simulink unter zusätzlicher Verwendung der Robotics Toolbox, die hilfreiche Methoden zur Robotersteuerung beinhaltet, realisiert. Die Simulation der Roboterfahrt teilt sich in folgende Punkte.

- Erstellung von Karten
- Implementierung der Kinematik des von der HAW zur Verfügung gestellten Roboters
- Simulation der ausgewählten Sensoren
- Bahnplanung und Lokalisierung
- Steuerung
- Visualisierung der Roboterfahrt

Bei der tatsächlichen Steuerung des Roboters muss zunächst die Hardware des Roboters verstanden und durch die Sensoren erweitert werden. Die Steuerung basiert auf der simulierten Steuerung und muss durch folgende Punkte erweitert werden:

- Einbindung der gesamten Steuerung in ein Simulinkmodell, sodass diese auf die dSpace-Box geladen werden kann.
- Einlesen und Auswertung der Sensordaten
- Ansteuerung der Motoren

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in sechs Hauptkapitel. Das erste Kapitel gibt eine Einleitung in die Bachelorarbeit, indem die Motivation, die Aufgabenstellung und der Aufbau der Arbeit vorgestellt werden. In Kapitel zwei wird die Ausstattung des von der HAW Hamburg zur Verfügung gestellten Roboters vorgestellt. Dabei werden die verwendeten Hardwarekomponenten und das Kinematikmodell des Roboters erläutert. Kapitel drei und Kapitel vier geben eine theoretische Einführung in die Thematik Navigation und Selbstlokalisierung durch Scanmatching von Robotern. Hierbei werden mehrere Lösungsvarianten vorgestellt, von denen am Ende eine Kombination realisiert wurde. Kapitel fünf und Kapitel sechs gehen anschließend auf die tatsächliche Realisierung des Roboters ein. In Kapitel fünf wird die Steuerung und Regelung des Roboters erläutert. Das sechste Kapitel beschäftigt sich mit dem eigentlichen Programm, das für den Roboter erstellt wurde. Dabei werden einzelne Teile des Programms genauer erläutert, das komplette selbstgeschriebene Programm befindet sich im Anhang. Außerdem werden besonders interessante Problemstellungen hervorgehoben und beschrieben. Im letzten siebten Kapitel wird eine Zusammenfassung über das Projekt gegeben, Schwachstellen und Optimierungsmöglichkeiten werden darin aufgeführt.

2 Aufbau des mobilen Roboters

2.1 Kinematik

Der Roboter hat einen Differentialantrieb, das heißt er besteht aus zwei angetriebenen Rädern und zwei passiv mitlaufenden Räder. Die mitlaufenden Räder sind zusätzlich auf einer drehbaren Achse gelagert, sodass eine Winkeländerung während der Fahrt gewährleistet ist. In der folgenden Abbildung 2-1 und zugehörigen Beziehungen wird die Kinematik des Differentialantriebes beschrieben.

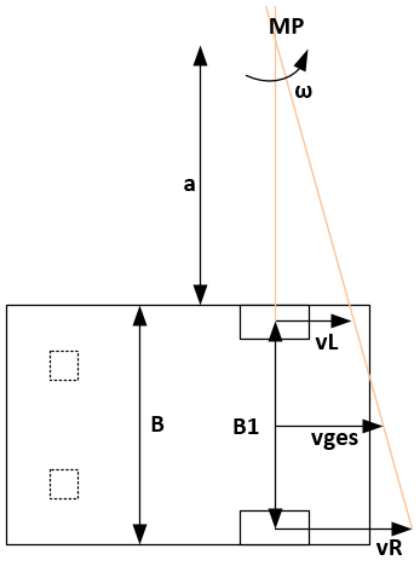

Abbildung 2-1: Kinematikmodell

$$v_R = v_{ges} + \frac{B1}{2} \cdot \omega \qquad 2\text{-}1$$

$$v_L = v_{ges} - \frac{B1}{2} \cdot \omega \qquad 2\text{-}2$$

$$\omega = \frac{v_R - v_L}{B} \qquad 2\text{-}3$$

$$v_{ges} = \frac{v_R + v_L}{2} \qquad 2\text{-}4$$

Hierbei stellt v_{ges} die Translationsgeschwindigkeit zwischen den beiden Rädern, ω die Winkelgeschwindigkeit um den Momentanpol (MP), v_L die Geschwindigkeit des linken Rades, v_R die Geschwindigkeit des rechten Rades und B1 den Abstand vom rechten Rad zum linken Rad dar.

Durch Integration über die Zeit t lassen sich die x_i-und y_i-Werte sowie der Drehwinkel θ_i des Roboters zum Zeitpunkt i ermitteln. Als Anfangswert wird die vorangehende Position $i-1$ gewählt.

$$\theta_i = \int \omega \; dt + \; \theta_{i-1} \qquad 2\text{-}5$$

$$x_i = \; \int v_{ges} \cdot \cos(\theta_i)\, dt + x_{i-1} \qquad 2\text{-}6$$

$$y_i = \int v_{ges} \cdot \sin(\theta_i)\, dt + y_{i-1} \qquad 2\text{-}7$$

2.2 Antriebsmotoren und Odometriedaten

Der Roboter ist mit zwei elektronisch kommutierten DC-Servomotoren der Firma Faulhaber Serie 3564K024B CS, die über ein PWM-Signal angesteuert werden, ausgestattet. Die Übersetzung i von Elektromotor zu angetriebenem Rad beträgt 1:14. Außerdem ist jeder Motor mit einem sogenannten Motion Controller verbunden. Bei den Motion Controller handelt es sich um digitale Signalprozessoren, mit denen die Motoren geregelt und konfiguriert werden können. Über eine serielle Schnittstelle und die zugehörige Software können so problemlos Parameter des Motors verändert werden [Faulhaber 2009]. Folgende Einstellungen wurden für den Roboter neu konfiguriert.

- Die maximale Drehzahl n_{max} des Roboters wurde auf 140 1/min gesetzt, da der Roboter sich nur mit geringen Geschwindigkeiten bewegen soll und somit eine präzisere Einstellung der Geschwindigkeit möglich ist.
- Der Fault-Pin des Motion Controllers wurde als Impulsausgang mit 255 Impulsen/Umdrehung definiert. Somit stehen Odometriedaten, d.h. Encoderdaten, zur Verfügung, mit denen man eine ungefähre Roboterposition ermitteln kann.

Ansteuerung der Motoren über Pulsweitenmodulation

Ein PWM-Signal besteht aus Rechtecksignalen, die eine feste Periodendauer besitzen. Das Pulssignal wird z.B. durch einen Tiefpassfilter demoduliert, sodass sich eine mittlere Spannung über die Periode ergibt, die an die Motoren weitergeleitet wird. Je größer das Verhältnis von Pulsweite zu Periodendauer ist, desto höher ist die Spannung, die bei den Motoren ankommt. Die folgende Abbildung 2-2 stellt ein PWM-Signal mit geringer Pulsweite und ein PWM-Signal mit großer Pulsweite dar.

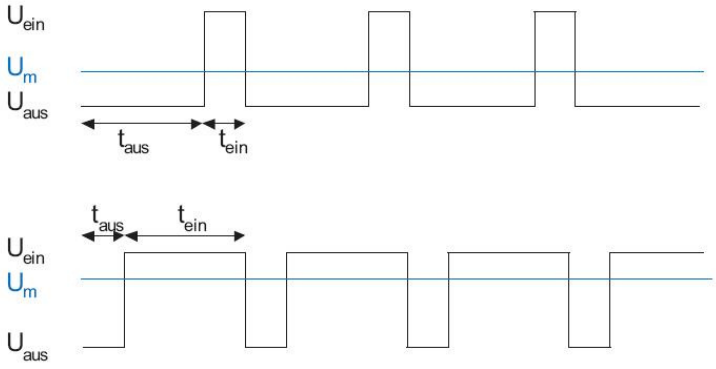

Abbildung 2-2: niedriges PWM-Signal (oben), hohes PWM-Signal (unten) aus [Robertvon-Goess 2009]

Die folgende Gleichung bestimmt die Mittelspannung U_m über das PWM-Signal. Dabei entspricht U_{ein} der anliegenden Spannung, t_{ein} und t_{aus} der Zeit, in der das Signal eins oder null ist.

$$U_m = U_{ein} \cdot \frac{t_{ein}}{t_{ein}+t_{aus}} \qquad 2\text{-}8$$

Die Motoren von Faulhaber sind so konfiguriert, dass bei einem Verhältnis von Pulsweite zu Periodendauer von 50% Stillstand herrscht. Ist das Verhältnis kleiner als 50% drehen sich die Motoren in die Vorwärts-Richtung des Roboters, ist das Verhältnis größer als 50% drehen sie sich in die andere Richtung. So ist ein Fahren in beide Richtungen gewährleistet. Setzt man das Verhältnis also auf 0% fährt der Roboter mit der maximalen Drehzahl von 140 1/min vorwärts [Faulhaber 2009]. Das PWM-Signal wird von der Steuereinheit *MicroAutoBox* erzeugt. dSpace stellt einen Simulink-Block „DIO_TYPE1_PWM_TPU_Mx_Cy“ zur Verfügung, der als Eingang die Periodendauer sowie das Verhältnis von Pulsweite zu Periode verlangt [dSpaceHelpDesk]. Aus diesen beiden Informationen wird dann ein PWM-Signal in den zugehörigen Ausgängen generiert. Das Pulsweitenverhältnis PWM lässt sich aus der Geschwindigkeit des linken oder rechten Rad $v_{L/R}$ des Roboters folgendermaßen berechnen.

$$PWM = -\frac{0.5 \cdot i \cdot 60 \cdot v_{R/L}}{n_{max} \cdot r \cdot 2\pi} + 0.5 \qquad 2\text{-}9$$

Ermittlung der Roboterposition mittels Encoderdaten

Die MicroAutoBox bietet digitale Eingänge mit integrierten Pull-Up-Widerständen. Ein Pull-Up-Widerstand ist ein hochohmiger Widerstand, der das Signal auf das nächst höhere Potential bringt, sofern das Signal vom Ausgang nicht aktiv auf das niedrigere Signal gebracht wird. Das Signal wird sozusagen aufgerundet, sodass das Encodersignal nur aus Low und High besteht. Zusätzlich stellt dSpace einen entsprechenden Simulink-Block „DI-

O_TYPE1_FPW2D_CTM_Mx_Bly“ zur Verfügung, der laufend die Frequenz von einer Schrittflanke zur nächsten Schrittflanke, wie in Abbildung 2-3 dargestellt, liefert [dSpaceHelpDesk].

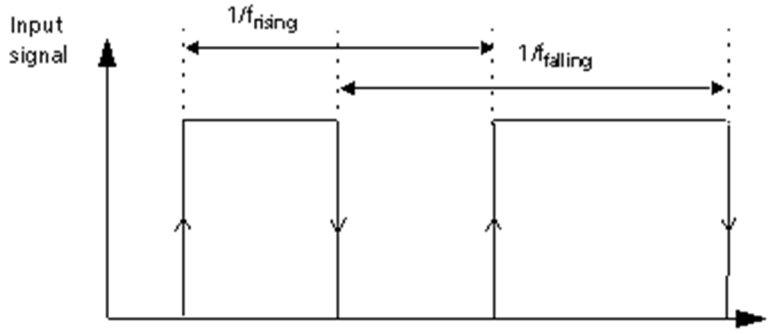

Abbildung 2-3: Encodersignal in Simulink aus [dSpaceHelpDesk]

Für jedes Rad gibt es ein unabhängiges Encodersignal, d.h. es werden zwei Eingänge mit Pull-Up-Widerstand verwendet. Diese Signale müssen nun kombiniert werden und in die x- und y-Werte des globalen Koordinatensystems umgerechnet werden. Dafür wird die Frequenz von der einen Schrittflanke zur nächsten Schrittflanke f_{rising} auf eine Momentanwinkelgeschwindigkeit $\omega_{R/L}$ des linken und rechten Rades umgerechnet.

$$\omega_{R/L} = 2 * pi * \frac{f_{rising}}{N_{impulse} * i} \qquad 2\text{-}10$$

$N_{Impulse}$ entspricht dabei der Anzahl der Impulse pro Umdrehung des Motors und i der Übersetzung von Elektromotor zu Rad. Pro Radumdrehung werden also $N_{Impulse} * i = 255\ Impulse * 14 = 3570$ Impulse durchlaufen. Die Momentanwinkelgeschwindigkeit kann mittels Radius r des Rades auf die an dem Rad anliegende Geschwindigkeit umgerechnet werden.

$$v_{R/L} = \omega_{R/L} * r \qquad 2\text{-}11$$

Aus den Geschwindigkeiten des linken und rechten Rades kann nun nach Kapitel 2.1 die Ist-Position $[x_i, y_i, \theta_i]$ des Roboters gemäß Encodersensoren bestimmt werden.

2.3 MicroAutoBox als Steuergerät

Hardware

Als Steuergerät wird die MicroAutoBox 1401/1505/1507 von der Firma dSpace verwendet. Die MicroAutoBox ist ein Echtzeitsystem, das häufig für Steuerungsaufgaben in der Automo-

bilbranche eingesetzt wird. Die folgenden technischen Daten wurden dem Datenblatt [MicroAutoBoxOverviewofBoardRevisions_Release2015A] entnommen.

- Prozessor: PPC750FX
- Taktfrequenz der CPU: 800 MHz
- Flashspeicher: 8MB
- Spannungsversorgung: 6- 40 V

Außerdem stellt die MicroAutoBox verschiedene analoge und digitale Ein- und Ausgänge zur Verfügung. Die folgende Abbildung 2-4 und Tabelle 2-1 zeigen welche Ein- und Ausgänge bei dem Roboter besetzt wurden.

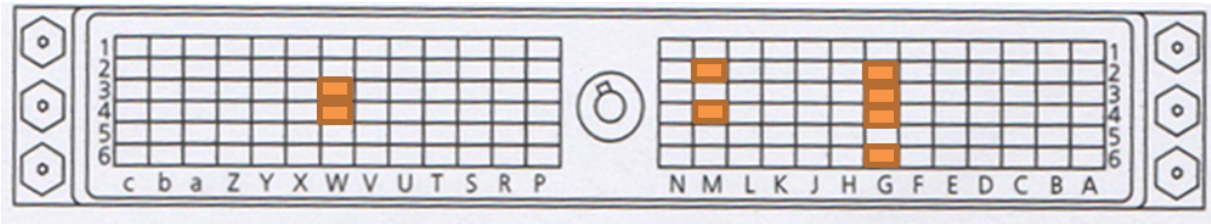

Abbildung 2-4: Pinbelegung des Anschlussfeldes der MicroAutoBox (in Anlehnung an [MicroAutoBox 2013])

Tabelle 2-1: Pinbelegung der MicroAutBox und ihre Beschreiung

PIN	**Beschreibung**
M2	Digitaler Eingang mit Pull-Up-Widerstand für Frequenz- oder Pulsweitenmessung für das rechte Rad
M4	Digitaler Eingang mit Pull-Up-Widerstand für Frequenz- oder Pulsweitenmessung für das linke Rad
G3	PWM-Ausgangssignal für das linke Rad
G6	PWM-Ausgangssignal für das rechte Rad
W3, W4	Spannungsversorgung (24 V)
G3,G4	Masse/Ground

Software

Ein großer Vorteil ist, dass dSpace einen Compiler liefert, der Matlab/Simulink-Modelle in C/C++ Code übersetzt. Dieser Code kann dann problemlos über eine PCMCIA-Steckkarte auf die Steuereinheit geladen werden. Mit der zugehörigen Software ControlDesk können Parameter des Echtzeit-Modells eingesehen und verändert werden. Das ist zur Überprüfung des Echtzeitmodells hilfreich.

dSpace bietet zusätzlich einen dSpace HIL API .NET server, mit dem man über Matlab auf das Modell der Steuereinheit zugreifen kann. Diese API arbeitet objektorientiert und bietet Klassen und Methoden, mit denen man Daten auslesen und schreiben kann. So ist es möglich, parallel zum laufenden Modell ein weiteres Programm auszuführen, das zur Steuerung beiträgt [Migration Guide 2013]. In dieser Arbeit wurden die Daten des Lasersensors über eine USB-Schnittstelle auf den Laptop übertragen, in einem Matlab-Skript verarbeitet und die notwendigen Informationen wie Ist-Position oder Pfadabänderungen an die MicroAutoBox übergeben.

2.4 2D-Lasersensor

Zu Beginn des Projektes standen folgende Sensoren zur Verfügung: Sick TIM310-1030000, PMD Nano, Kinect v1, Hokuyo URG-04LX-UG01. Bei dem Lasersensor der Firma Sick handelt es sich um einen Feldüberwachungssensor, d.h. der Benutzer wird informiert, wenn ein Objekt in ein definiertes Feld eintritt. Er liefert im Gegensatz zum Sensor der Firma Hokuyo keine Abstandswerte zu den Objekten in seiner Umgebung. Da zur Selbstlokalisierung die genauen Abstandswerte zu den Objekten der Umgebung benötigt werden, wurde der Hokuyo URG-04LX-UG01 ausgewählt. Zusätzlich sollte noch eine Kamera in das Robotersystem eingebunden werden, sodass Hindernisse erkannt werden können, die sich nicht in der Arbeitsebene des Laserscanners befinden. Zunächst fiel der Fokus auf die Time-Of-Flight Kamera PMD Nano, da sie sehr schnelle Prozesszeiten hat und somit optimal bei Echtzeitanwendungen eingesetzt werden kann. Nach einer Einarbeitungsphase fiel aber auf, dass die PMD Nano nur Tiefenbilder und keine Grauwerte der Bildaufnahme liefert. Die Graubilder sind allerdings notwendig, um mittels Canny-Filter Kanten der Bildaufnahme zu extrahieren, sodass die Datenmenge der 3D-Bilder heruntergesetzt und die Verarbeitungszeit minimiert wird. Folglich wurde die Kinect v1, die sowohl Tiefen- als auch Farbbilder liefert, in Betracht gezogen. Die Herausforderung hierbei war es, die Kamera zu kalibrieren und die rohen Messdaten in Matlab einzubinden. Im Internet stehen diverse Treiber und Libraries zur Verfügung: Kinect for Windows SDK 2.0, OpenNI und Nite, OpenKinect/libfreenect. Die Ermittlung der Kalibrierungsparameter stellte sich jedoch als komplex dar und hätte den vorgesehen Zeitrahmen gesprengt. Aus diesem Grund wurde der Roboter nur mit einem 2D-Lasersensor URG-04LX von der Firma Hokuyo ausgestattet.

Der gewählte Sensor misst in der Ebene die Abstände zu Wänden und Objekten in einem Winkelbereich von 240°, einer Schrittweite von 0.36° und einer maximale Reichweite von 4,095m. Abbildung 2-5 und Abbildung 2-6 zeigen den in dieser Arbeit verwendeten Laserscanner mit seinen Abmaßen und seinem Messbereich.

Abbildung 2-5: Lasersensor URG-04LX von der Firma Hokuyo

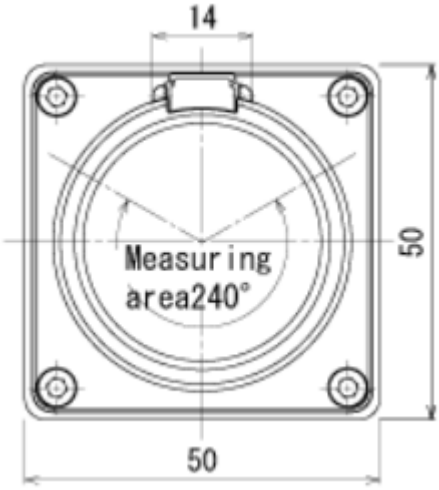

Abbildung 2-6: Messbereich

Das Messprinzip eines Lasersensors funktioniert, indem ein Sender einen Laserstrahl aussendet. Der Laserstrahl trifft auf ein Objekt und wird reflektiert, sodass das Lasersignal wieder zum Sensor zurückgeleitet wird. Der Strahl wird durch einen Spiegel umgeleitet und vom Empfänger registriert. Die gemessene Zeit von Sender bis Empfänger des Lasersignals kann dann in den entsprechenden Abstand umgerechnet werden. Um nun im 2D-Bereich messen zu können, gibt es im Sensor eine rotierende Spiegeleinrichtung, die den Laserstrahl in verschiedene Richtungen aussenden kann.
Es ist wichtig, dass der Sensorstrahl durch einen sehr kleinen Öffnungswinkel ausgesendet wird, da dieser kegelförmig ist und bei einem zu großen Durchmesser mehrere Objekte detektieren könnte. Weitere Probleme können bei Objekten mit stark absorbierenden Oberflächen, Spiegel oder Glas auftreten. Bei stark absorbierenden Oberflächen wird das Signal gar nicht erst reflektiert, bei Glas geht der Sensorstrahl meistens hindurch, sodass nur die Objekte hinter dem Glas registriert werden, und bei Spiegeln wird der Sensorstrahl in eine andere Richtung reflektiert.

Datenübertragung mittels serieller Schnittstelle

Bei der Datenübertragung handelt es sich um eine serielle Schnittstelle, die mittels virtuellen Com Port realisiert wird. Die Firma Hokuyo stellt ein Kommunikationsprotokoll [Hoyuko, Kawata 2008] zur Verfügung, in dem alle notwendigen Befehle sowie die Interpretation der Datensätze beschrieben werden. Der Benutzer schickt dem Laserscanner einen Befehl und bekommt als Antwort einen ASCII-Code, der die angefragten Informationen enthält. Im Folgenden werden der GD-Befehl und die Interpretation seines Echos exemplarisch beschrieben. Der GD-Befehl fordert die letzte erstellte Messung im Bereich von Startwinkel und Endwinkel an. Für den Roboter soll die Messung immer für den kompletten Winkelbereich ausgewer-

tet werden. Nach Abbildung 2-7 ergibt sich folgender Befehl, der über die serielle Schnittstelle an den Laserscanner gesendet wird.

```
fprintf(lidar,'GD0044072500\n')
```

Die Funktion fprintf stammt aus der Klasse Serial in Matlab, die eine serielle Kommunikation ermöglicht. Die Funktion sendet den Befehl GD0044072500\n an den Laserscanner, wobei 0044 den Startwinkel, 0725 den Endwinkel und die 00 den sogenannten Cluster Count darstellt. Der Cluster Count gibt an, wie viele, aufeinander folgende Messwerte zu einem Wert zusammengefasst werden sollen. Mit dem Zeilenumbruch \n wird der Befehl abgeschlossen. Der Befehl könnte ebenso mit einem Zeilenvorschub abgeschlossen werden.

(HOST→ SENSOR)

					optional	
G (47H)	D (44H) or S (53H)	Starting Step (4bytes)	End Step (4 bytes)	Cluster Count (2bytes)	String Characters	LF

Abbildung 2-7: GD-Befehl in Anlehnung an [Hoyuko, Kawata 2008]

Das Echo des Laserscanners muss anschließend ausgelesen und interpretiert werden. Die Abbildung 2-8 zeigt die Form, mit der das Echo an den Benutzer gesendet wird. Dabei werden alle Daten aneinandergereiht und in einer ASCII-Abfolge zur Verfügung gestellt.

G	D or S	Starting Step	End Step	Cluster Count	String Characters	LF
0	0	P	LF	Time Stamp	Sum	LF
Data Block 1 (64 bytes)			Sum	LF		
----------			Sum	LF		
Data Block N-1 (64 bytes)			Sum	LF		
Data Block N (n bytes)			Sum	LF	LF	

Abbildung 2-8: Echo des GD-Befehls aus [Hoyuko, Kawata 2008]

Interessant sind nun die Datenblöcke von 1 bis N, die aus dem Informationsblock herausgefiltert werden müssen. Dabei sind die Zeilenvorschübe (LF) bei der Filterung hilfreich. Da die Datenblöcke in verschlüsseltem ASCII-Code vorliegen, müssen diese dekodiert werden. Die folgende Abbildung 2-9 zeigt das 3-Zeichen-Entschlüsseln von ASCII-Code in eine Dezimalzahl mit der Einheit mm. Drei aufeinanderfolgende Zeichen der Datenblöcke repräsentieren also immer einen Abstand in mm.

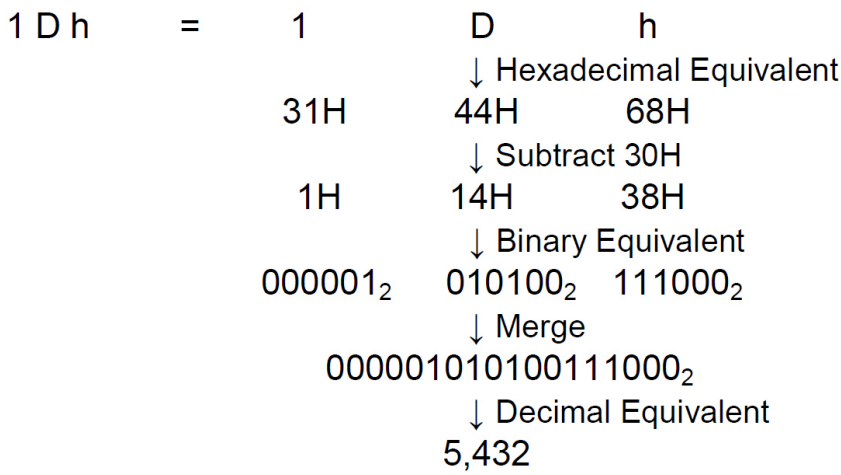

Abbildung 2-9: 3-Character Decoding Example von [Hoyuko, Kawata 2008]

In der folgenden Abbildung 2-10 ist zum besseren Verständnis ein Beispiel-Scan abgebildet. Dieser wurde in der Testumgebung aufgenommen, der Roboter befindet sich hierbei an der Position des roten Kreuzes.

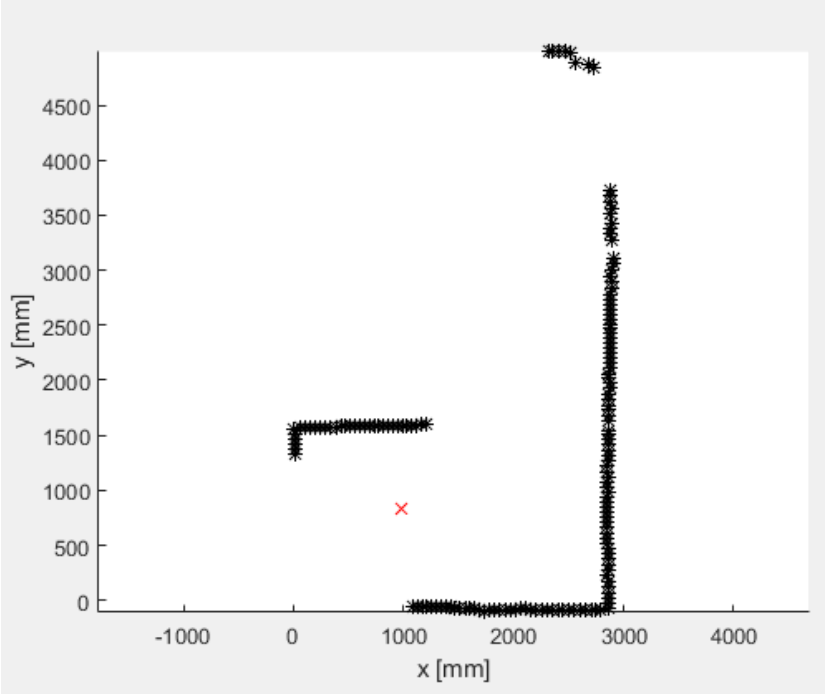

Abbildung 2-10: Beispiel-Scan der Testumgebung

3 Navigation

3.1 Karten

3.1.1 Gitternetzkarte

Bei der Gitternetzkarte wird die Umgebung in gleichgroße Zellen aufgeteilt, sodass ein Gitter entsteht. Die Größe der Zellen bestimmt dabei die Genauigkeit der Karte. Diskrete Gitternetzkarten füllen die Zellen mit einer eins, wenn diese besetzt sind, und mit einer null, wenn es sich um Freiraum handelt. Eine Erweiterung dieser diskreten Gitternetzkarte ist das „Occupancy Grid“. Bei der Belegtheitskarte wird in jeder Zelle die Wahrscheinlichkeit gespeichert, mit der die Zelle besetzt ist.

Die Gitternetzkarte hat große Vorteile bei der Pfadplanung und Lokalisierung eines mobilen Roboters. Der geplante Pfad wird durch eine Menge von Punkten beschrieben, wobei jeder Punkt einer Zelle in der Gitternetzkarte entspricht. Bei der Lokalisierung können die gemessenen Punkte der Sensordaten ebenfalls in die Zellen der Gitternetzkarte eingeteilt und somit einfach mit der Karte abgeglichen werden.

Ein Nachteil von Gitternetzkarten ist der große Speicheraufwand. Dieser ist insbesondere relevant, wenn es sich um komplexe und weitläufige Umgebungen handelt. Um die Speichergröße zu reduzieren, können Rasterkarten aus dem 2D-Raum nach dem Prinzip der Quadtrees erstellt werden. Das bedeutet, dass große freie Flächen auch mit großen Zellen beschrieben werden. Je näher die Zelle sich an einem Objekt befindet, desto kleiner wird sie, um so eine hohe Genauigkeit zu gewährleisten [Hertzberg, Lingemann und Nüchter 2012].

In Abbildung 3-1 sind eine einfache Gitternetzkarte und eine Gitternetzkarte nach dem Quadtree-Algorithmus im Vergleich dargestellt.

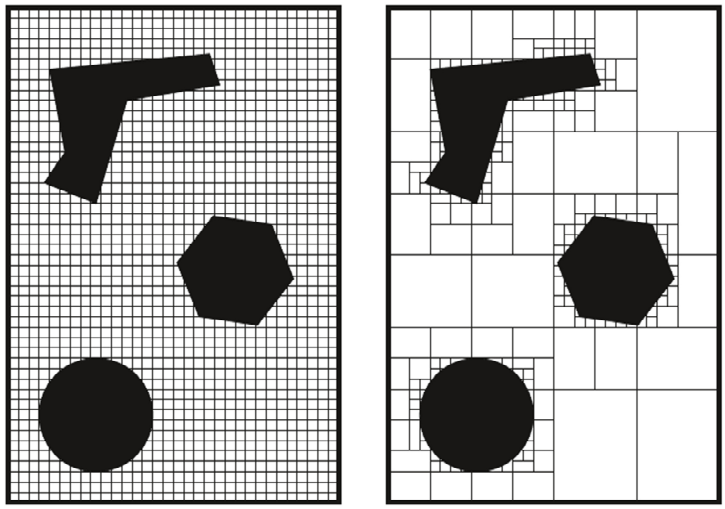

Abbildung 3-1: diskrete Gitternetzkarte (rechts), diskrete Gitternetzkarte unter Verwendung von Quadtrees (links) von [Hertzberg, Lingemann und Nüchter 2012].

In dieser Bachelorthesis wurde eine diskrete Gitternetzkarte mit gleichgroßen Zellen gewählt. Die Wahl ist durch die gewählten Bahnplanungs- und Lokalisierungsalgorithmen, die sich am besten mit diesem Kartentyp verbinden lassen, begründet. Außerdem ist die Testumgebung auf einen kleinen Raum beschränkt, sodass der Speicheraufwand gering gehalten wird.

3.1.2 Merkmalsbasierte Karte

Merkmalsbasierte Karten basieren auf der Speicherung von Merkmalen der Umgebung. Es kann sich hierbei einerseits um natürliche Merkmale handeln wie z.B. Ecken, Kanten oder Objekte wie z.B. ein Mülleimer. Andererseits kann die Umgebung auch mit künstlichen Landmarken präpariert sein wie z.B. reflektierende oder farbige Streifen. Die Merkmale, die in der Karte aufgeführt sind, müssen an die Sensorausstattung des Roboters angepasst werden, da bestimmte Sensoren nur bestimmte Merkmale registrieren. Die folgende Abbildung 3-2 soll eine merkmalsbasierte Karte visualisieren. Die Farbe und Form der Objekte entsprechen dabei den Merkmalen. Die zugehörige Position in der Umgebung ist ebenfalls gespeichert. Registriert der Roboter ausreichend viele Landmarken, kann dieser über Triangulation, die in [Hertzberg, Lingemann und Nüchter 2012] genauer erläutert wird, seine eigene Position und Orientierung ermitteln.

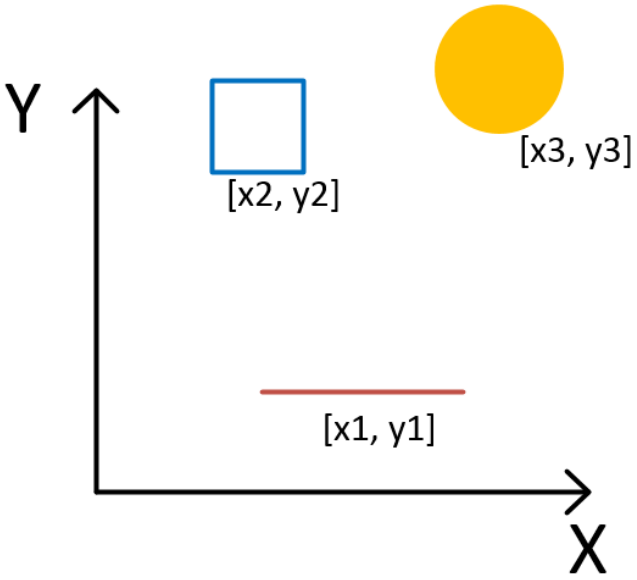

Abbildung 3-2: Visualisierung einer merkmalsbasierten Karte

Die merkmalsbasierte Karte benötigt im Gegensatz zur Rasterkarte einen geringeren Speicherplatz, da nur die Orte der Merkmale gespeichert werden, d.h. die restlichen Flächen der Umgebung sind unbekannt [Thrun, Burgard und Fox 2006].

3.1.3 Topologische Karten

Topologische Karten besitzen in der Regel keine geometrischen Eigenschaften, sondern basieren ausschließlich auf Informationen von einzelnen Plätzen, die sensorisch eindeutig erfasst werden können.

Eine topologische Karte besteht aus Knotenpunkten und Linien, die diese verbinden. In den Knotenpunkten sind Attribute gespeichert, die den Standort charakterisieren. Dabei kann es sich um gespeicherte Sensoraufnahmen wie zum Beispiel Kamerabilder handeln, die an diesem Ort aufgenommen wurden. Die Knotenpunkte können aber auch signifikante Stellen wie Türrahmen, Kreuzungen oder Sackgassen beschreiben. Auch die verbindenden Linien können Eigenschaften speichern, die den Weg von Punkt A zu Punkt B beschreiben (z.B.: Länge oder minimale Durchgangsbreite) [Hertzberg, Lingemann und Nüchter 2012].

Bei dem HVV-Netz-Plan, der in Abbildung 3-3 abgebildet ist, handelt es sich ebenso um eine topologische Karte. Die Knotenpunkte stellen die einzelnen Stationen dar. Ihre Eigenschaften beschreiben durch Verwendung von Straßennamen oder relevanten Plätzen eine relativ genaue Position in Hamburg. Die verbindenden Linien geben dem Bahnfahrer nur eine ungefähre Orientierung, wo die Bahn zwischen den Knotenpunkten entlang läuft. Außerdem beinhalten sie durch ihre farbliche Einteilung allgemeine Informationen darüber, welche Bahn auf welcher Strecke fährt.

Abbildung 3-3: topologische Karte von [HVV Schnellbahnplan]

3.2 Pfadfindung

In den meisten Anwendungen kennt der Roboter seine Startposition und das Ziel, wo er hinfahren möchte. Die Pfadplanung beschäftigt sich mit der Problematik, den besten Weg vom Start zum Ziel zu bestimmen.

3.2.1 Sichtbarkeitsgraphen

Bei der Sichtbarkeitsgraphen-Methode handelt es sich um eine Pfadplanung in bekannter Umgebung, die eine Menge von Polygonen enthält. Es werden Start- und Zielpunkt mit allen Ecken der Hindernisse verbunden. Wichtig ist, dass die Verbindungslinie dabei kein Hindernis schneidet, d.h. die Sichtbarkeit von Start- oder Zielpunkt bis zur Ecke muss gewährleistet sein. Die Ecken der Hindernisse untereinander werden auf die gleiche Weise verbunden. Aus der Sammlung an Linien wird dann der kürzeste Weg ermittelt. Problematisch ist hierbei, dass der Roboter bei diesem Algorithmus sehr nah an den Hindernissen vorbeifährt. Bei der Anwendung des Sichtbarkeitsgraphen-Algorithmus ist es also wichtig, dass man die Hindernisse um mehr als den halben Roboterdurchmesser vergrößert, damit Kollisionen vermieden werden [Bräunl 2008].

Die Abbildung 3-4 zeigt den Sichtbarkeitsgraphen für ein beispielhaftes Start-Ziel-Szenario.

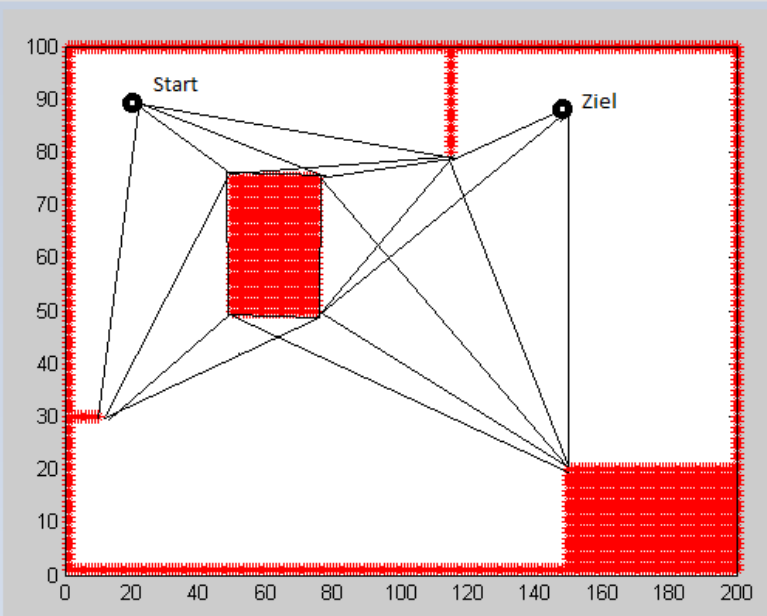

Abbildung 3-4: Sichtbarkeitsgraph-Methode

3.2.2 Potentialfeldmethode

Die Potentialfeldmethode findet ihre Anwendung in einer Gitternetzkarte. Dabei wird für jede Zelle des Gitternetzes der Abstand von der entsprechenden Zelle bis zum Zielpunkt berechnet und gespeichert. Es entsteht ein Potentialfeld, dessen globales Minimum sich beim Zielpunkt befindet. So kann sich der Roboter von Zelle zu Zelle vorarbeiten, in dem dieser immer in die Nachbarzelle mit dem geringsten Abstand zur Zielposition fährt. Auch bei diesem Algorithmus führt der Pfad meistens sehr nah an den Hindernissen entlang, weshalb diese bei der Pfadberechnung ebenfalls mindestens um den halben Roboterdurchmesser aufgebläht werden müssen. Der Algorithmus birgt zusätzlich die Gefahr, dass der Roboter in einem lokalen Minimum landet, aus dem er dann nicht mehr rauskommt.

Die Potentialmethode ist gut geeignet, wenn wiederholt die gleiche Zielposition angefahren werden muss. Die Abstände für die Zellen zum Ziel müssen nur einmal berechnet werden und

können aus allen Start- oder Zwischenpositionen verwendet werden [Hertzberg, Lingemann und Nüchter 2012].
Die folgende Abbildung 3-5 zeigt ein beispielhaftes Gitternetz mit den zugehörigen Abständen zur Zielposition.

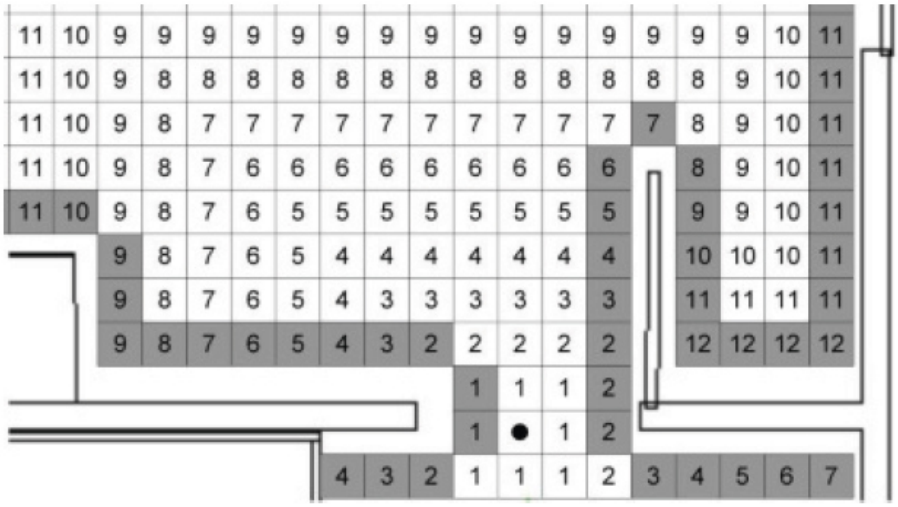

Abbildung 3-5: Potentialfeld im Occupancy-Grid von [Hertzberg, Lingemann und Nüchter 2012]

Die Pfadplanung in dieser Arbeit basiert auf der Potentialmethode, da sie sich optimal mit der Gitternetzkarte und den Sensorinformationen vereinen lässt. Ein weiterer Vorteil ist, dass das Potentialfeld für den leeren Raum nur einmal berechnet werden muss und im Laufe der Roboterfahrt immer wieder verwendet kann. Sobald unbekannte Hindernisse auftreten, muss allerdings ein weiteres Potentialfeld berechnet werden, was sehr zeitaufwändig und mit Nachteilen verbunden ist.

3.2.3 Voronoi-Methode

Die Voronoi-Methode versucht, im Gegensatz zu den beiden vorrangegangenen Methoden, den Pfad immer möglichst weit weg von den vorliegenden Objekten zu entwickeln. Voronoilinien sind dadurch gekennzeichnet, dass sie immer den gleichen Abstand zu mindestens zwei Objekten haben. Ihr Ursprung liegt daher in den nach innen zeigenden Ecken eines Raumes. Die Voronoilinien sind zum besseren Verständnis in Abbildung 3-6 für einen beispielhaften Raum dargestellt. Der Pfad entsteht dadurch, dass der Roboter vom Startpunkt möglichst schnell auf die nächstliegende Voronoilinie fährt. Von dort aus folgt der Roboter denjenigen Voronoilinien, die die kürzeste Strecke in Richtung Ziel bilden. Befindet sich der Zielpunkt nicht exakt auf einer der Voronoilinien, müssen diese zum Erreichen des Ziels wieder verlassen werden.

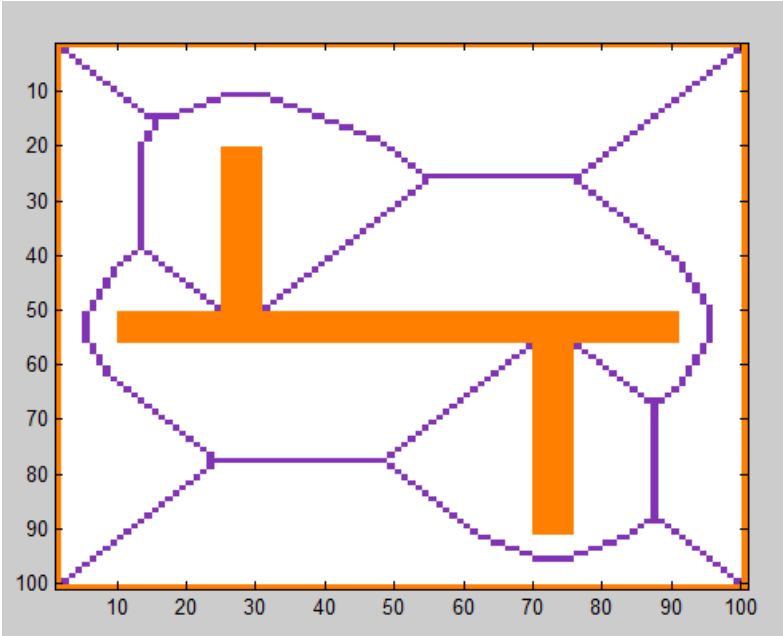

Abbildung 3-6: Voronoilinien (violett)

3.2.4 Bug-Algorithmus

Der Bug-Algorithmus ist eine reaktive Navigationsmethode, deshalb findet dieser Einsatz, wenn sich während der Roboterbahnfahrt ein Objekt in den Weg stellt. Es gibt mehrere Bug-Algorithmen (Bug1, Bug2, Bug3, DistBug), die alle darauf basieren, dass der Roboter sich entlang der Hinderniskontur bewegt. In dieser Arbeit wird nur der effizientere Bug3-Algorithmus näher erläutert: Stößt der Roboter auf ein Hindernis, dreht dieser sich parallel zur Hinderniskontur und folgt der Kontur solange, bis der Raum in Richtung Ziel wieder frei ist. In diesem Moment wird das Hindernis wieder verlassen und direkt auf das Ziel hinzugesteuert. Der im Vorhinein geplante Weg durch (z.B. die Potentialmethode) wird nicht wieder aufgenommen [Hertzberg, Lingemann und Nüchter 2012]. Abbildung 3-7 zeigt die Hindernisumfahrung eines Roboters mittels Bug3-Algorithmus.

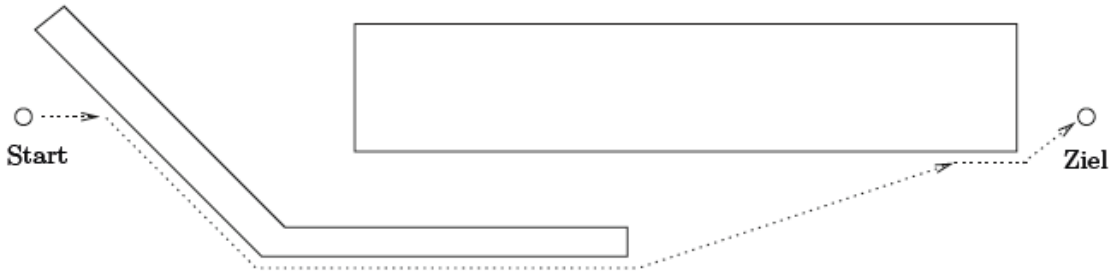

Abbildung 3-7: Bug3-Algorithmus von [Hertzberg, Lingemann und Nüchter 2012]

3.2.5 Glättung der Pfade

Die gewählte Pfadplanung nach der Potentialfeldmethode basiert auf Gitternetzkarten. Der Pfad besteht aus einer Abfolge von Punkten, wobei jeder Punkt eine benachbarte Zelle des vorangehenden Punktes beschreibt. Durch dieses Prinzip der Pfaderstellung entstehen sehr ruckartige Pfade, bei denen die Richtungsänderungen in einem Punkt beschrieben werden. Der Roboter würde im Punkt der Richtungsänderung eine sehr ruckartige Bewegung mit hohen Beschleunigungen durchführen, was aufgrund der Massenträgheitsmomente zu einer ungenauen Abfahrt des Pfades führen würde. Aus diesem Grund ist eine Glättung der Bahn, wie in Abbildung 3-8 dargestellt, notwendig.

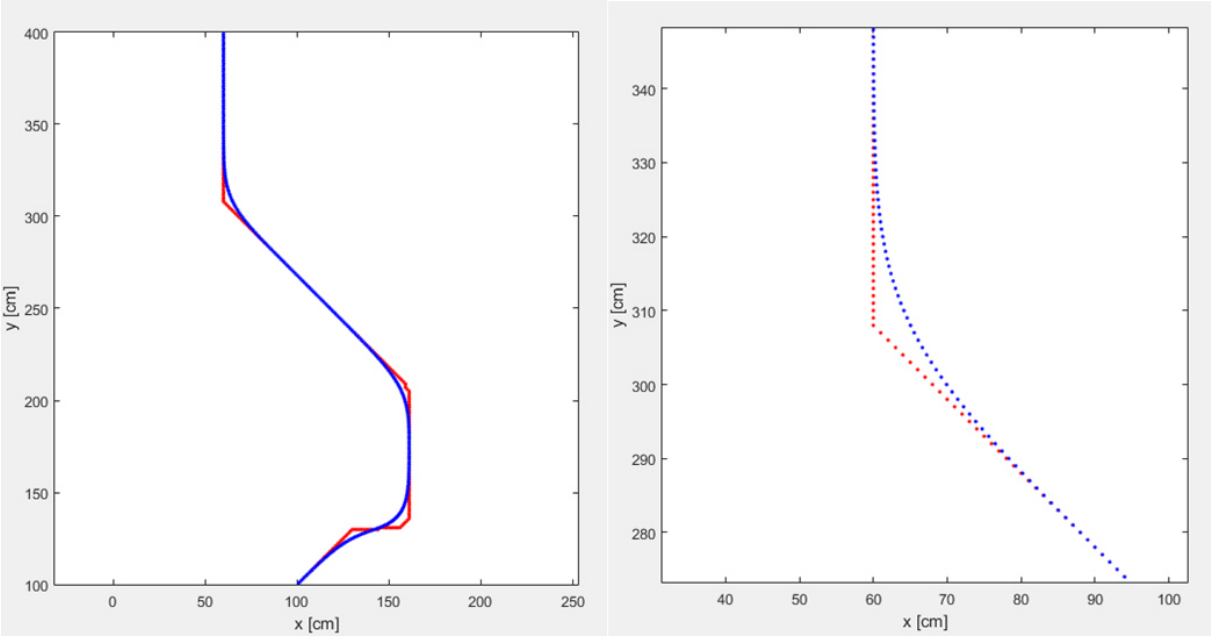

Abbildung 3-8: geplanter Pfad (rot), geglätteter Pfad (blau)

Die Glättung des geplanten Pfades wurde mittels eines iterativen Algorithmus in Anlehnung an [Robot Path Smoothing] realisiert. Dabei wird für jeden Punkt p_i der Abstand zu seinem vorangehenden Punkt p_{i-1} und zu seinem nachfolgenden Punkt p_{i+1} bestimmt. Der Punkt p_i wird dann so verschoben, dass sich der Abstand zu seinen Nachbarpunkten minimiert. Der Grad der Verschiebung pro Iteration wird in einem Gewichtungskoeffizient angegeben. Folgende Gleichung beschreibt die Verschiebung eines Punktes, dabei stellt α den Gewichtungskoeffizient dar.

$$p_{i,neu} = p_i + \alpha * ((p_{i-1} - p_i) + (p_{i+1} - p_i)) \qquad 3\text{-}1$$

Es ist zu erwähnen, dass Start- und Zielpunkt nicht verschoben werden sollten, da es sich um fest vorgegebene Werte handelt. Die Verschiebung der restlichen Punkte führt man nun so oft durch bis der gewünschte Glättungsgrad erreicht ist. Der Algorithmus wurde in der Funktion *smooth()*, die im Anhang A.1 aufgeführt ist, implementiert.

4 Scanmatching

In der mobilen Robotik kann das Prinzip des Scanmatching einerseits zur Kartenerstellung und anderseits zur Selbstlokalisierung genutzt werden. Bei der Kartenerstellung wird jeweils der aktuelle und vorige Scan gematcht. Durch das Aneinanderreihen der Sensorscans entsteht eine Karte. Bei der Selbstlokalisierung wird der Sensordatenscan mit einem Scan einer bekannten Karte verglichen. Liegen ausreichend viele natürliche Landmarken, also Ecken oder spezifische Objekte, vor, ist ein Matching und somit eine Lokalisierung möglich.

Mit „Scan“ ist eine Menge aus Punkten bezeichnet. Diese Punktmenge kann z.B. die Daten eines Lasersensors, der sich regelmäßig verändert, darstellen. Aber auch eine Karte einer bekannten Umgebung kann durch eine Punktwolke, also einen Scan, beschrieben werden. Dabei entspricht jeder Punkt einer besetzten Zelle in der Gitternetzkarte.

Mit „Matching“ ist das Übereinanderlegen zweier Scans bezeichnet. Dabei wird der eine Scan so verschoben und verdreht, dass die beiden Scans möglichst optimal übereinander liegen. Abbildung 4-1 zeigt zwei Scans (rot und blau), die durch den ICP-Algorithmus überlagert wurden.

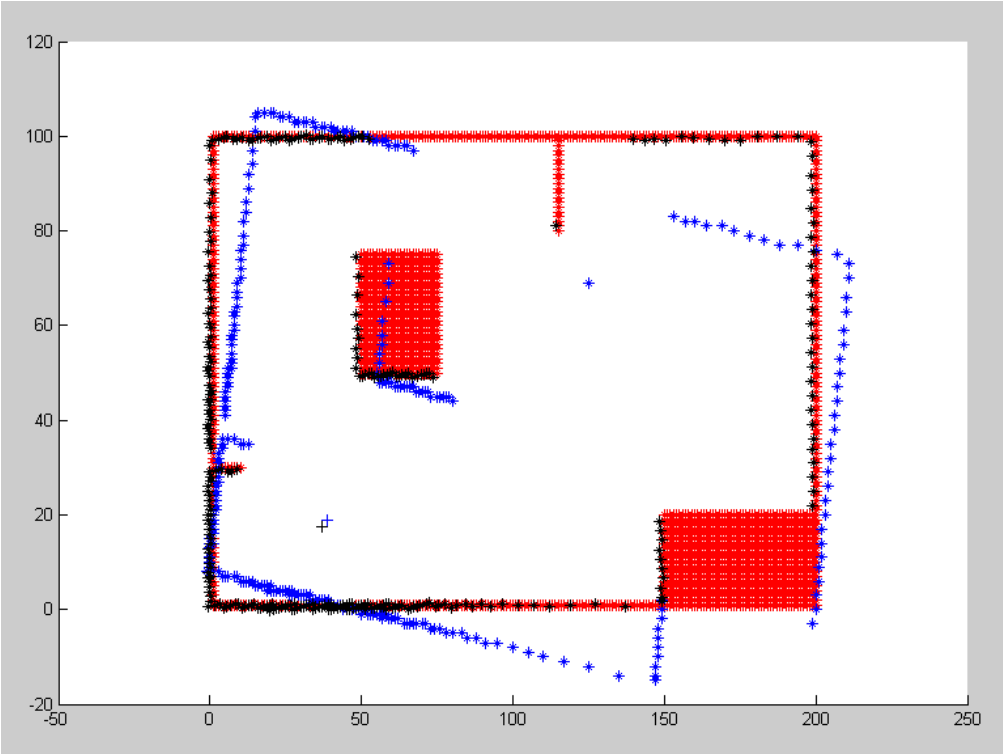

Abbildung 4-1: Scan D (blau), Scan M (rot), transformierter Scan (schwarz)

4.1 ICP-Algorithmus

Man nehme an, es liegen eine Punktwolke einer Umgebung (fest, M) und eine Punktwolke eines Sensors (verschiebbar und verdrehbar, D) vor. Die Punktwolken können eine unterschiedliche Anzahl an Punkten besitzen.

Nach [Corke 2013] sucht der *Iterative Closest Points Algorithmus* für jeden Punkt aus der zu verschiebenden/verdrehenden Punktwolke D den Punkt aus der festen Punktwolke M, der den geringsten Abstand hat. Es wird eine Transformationsmatrix ermittelt, die die Summe der Quadrate der Abstände minimiert. Die Transformationsmatrix $T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$ wird auf D angewandt. Dieser Prozess wird iterativ solange wiederholt, bis die Abstände von D zu M unter einer bestimmten Grenze δ_{max} liegen oder die Anzahl der maximalen Iterationen erreicht ist.
Zur Ermittlung der Transformationsmatrix in der jeweiligen Iteration werden die Punktwolkenschwerpunkte $\bar{M}$ und $\bar{D}$ ermittelt. Dafür wird die Summe aller Punkte durch die Anzahl N geteilt.

$$\bar{M} = \frac{1}{N_M} \sum_{i=1}^{N_M} M_i \qquad 4\text{-}1$$

$$\bar{D} = \frac{1}{N_D} \sum_{i=1}^{N_D} D_i \qquad 4\text{-}2$$

Aus diesen Schwerpunkten kann zunächst die Translation t berechnet werden.

$$t = \bar{D} - \bar{M} \qquad 4\text{-}3$$

Zur Berechnung der Rotation wird die Korrelationsmatrix W gebildet.

$$W = \sum_i (M_i - \bar{M})(D_i - \bar{D})^T \qquad 4\text{-}4$$

Über die Singularitätswertzerlegung (Singular Value Decomposition), auf die in [Corke 2013] näher eingegangen wird, kann die Rotationsmatrix R berechnet werden.
Es ist wichtig, dass für den Scan D eine ungefähre Position geschätzt wird, da umso näher der Scan D an der wahren Position liegt, desto höher ist die Chance, dass das Scanmatching ein ausreichend gutes Ergebnis liefert. Liegen geschätzte Position und Ist-Position zu weit auseinander, konvergiert der ICP-Algorithmus nicht gegen das globale Minimum, sondern gegen ein lokales Minimum. Die Schätzung kann aus den Odometriedaten erfolgen oder aus Geschwindigkeitsinformationen in einem Robotermodell durch Integration berechnet werden.

4.2 Filterung der Sensordaten

Um ein möglichst gutes Ergebnis zu erzielen, müssen die Sensordaten vor dem Matchingprozess gefiltert werden. Außerdem kann durch die Verringerung der Datenmenge die Berechnungszeit des Scamatching-Algorithmus runtergesetzt werden.

Medianfilter

Mit dem Medianfilter werden Ausreißer, die aufgrund von Sensorrauschen oder fehlerhaften Messungen entstehen, durch Glättung der Punktwolke D kompensiert. Der Algorithmus funktioniert so, dass für jeden Messpunkt k ein Mittelwert aus dem eigentlichen Messwert k und den n umliegenden Punkten gebildet wird. Je höher der Wert n gewählt wird, desto stärker wird die Punktwolke geglättet. Die Anzahl der Messpunkte ändert sich durch den Medianfilter nicht [Matlab medfilt1]. Die Signal Processing Toolbox in Matlab stellt einen Medianfilter als Funktion zur Verfügung.

```
y = medfilt1(x,n)
```

Dabei entspricht x der Punktwolke D, n der Anzahl der umliegenden Punkte und y dem gefilterten Datensatz. Die folgende Abbildung 4-2 zeigt Rohdaten und die mit dem Medianfilter erzeugten Daten im Vergleich.

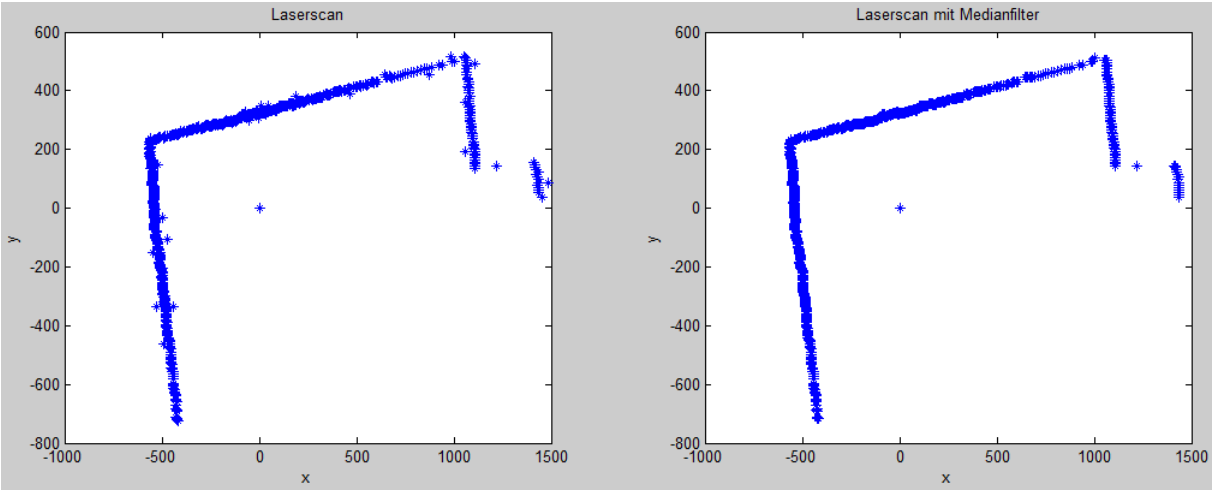

Abbildung 4-2: Medianfilter am Beispielscan

Reduktionsfilter

Der Reduktionsfilter verringert die Größe der Datensätze wie in Abbildung 4-3 zu sehen ist. Dabei werden mehrere Punkte, die sich in einem bestimmten Umkreis befinden, zu einem Punkt zusammengefasst. Der Algorithmus startet mit dem ersten Datenpunkt des Laserscans und summiert die darauffolgenden Datenpunkte solange auf, bis der Abstand zwischen dem ersten Datenpunkt und dem aktuellen Datenpunkt den Umkreisdurchmesser von $2r$ überschreitet. Es wird ein Mittelwert aus den gesammelten Punkten gebildet und diese werden durch den berechneten Mittelwert ersetzt. Der letzte Datenpunkt ist der neue Startpunkt, zu-

dem wieder alle darauffolgenden Punkte, die sich in dem definierten Umkreis befinden, addiert werden. Die Reduktion der Datensätze hat zur Folge, dass der ICP-Algorithmus schneller konvergiert und somit die Rechenzeit verkürzt wird.
Der Reduktionsfilter wurde als Funktion in Matlab implementiert.

```
function pointCloudRed = reducFilt(pointCloud, r)
```

Dabei stellt pointCloud die Punktwolke D, r den Radius des Umkreises und pointCloudRed den gefilterten Datensatz dar.

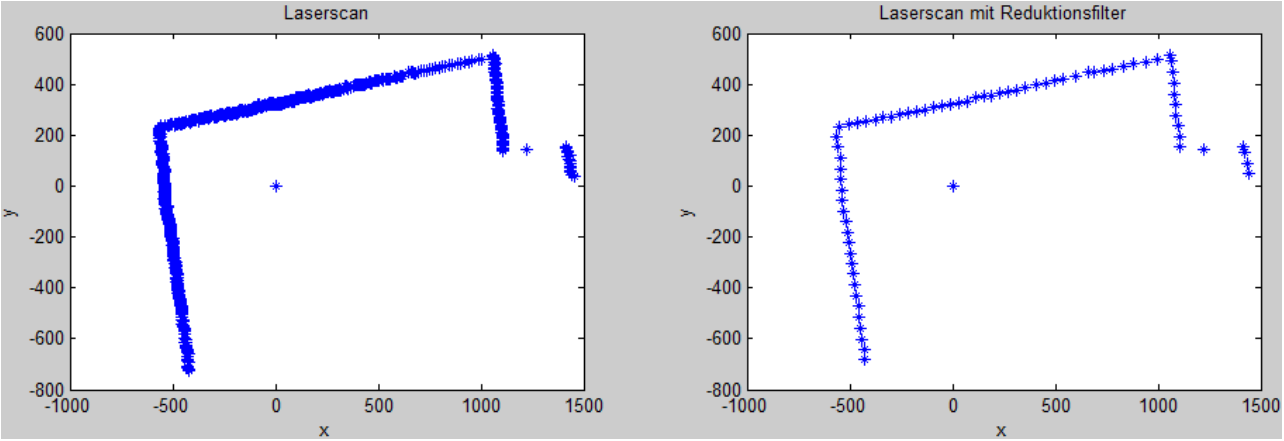

Abbildung 4-3: Reduktionsfilter am Beispielscan

4.3 Der kD-Baum

Der kD-Baum findet Anwendung beim Matching von großen Datenmengen. Durch ein Ausschlussverfahren ist es möglich die Suche nach dem nächsten Nachbarn im icp-Algorithmus deutlich zu beschleunigen. Er findet hauptsächlich Anwendung beim Scanmatching von 3D-Punktwolken.
Es handelt sich hierbei um einen binären Suchbaum in k-ter Dimension. Für die Punktwolken von Distanzkameras wäre k = 3, der Algorithmus wird zur Einfachheit halber allerdings für den 2-dimensionalen Raum erläutert. Es wird zwischen homogenen und heterogenen kD-Bäumen unterschieden. Der Unterschied ist, dass der homogene kD-Baum seine Punkte in den Knoten speichert und der heterogene kD-Baum seine Punkte in den Blättern speichert. Im Folgenden wird nur der homogene kD-Baum beschrieben.
Der kD-Baum wird so erstellt, dass die vorliegende Punktwolke in zwei annähernd gleich große Punktwolken aufgeteilt wird, wobei ein Punkt auf der aufteilenden Linie (auch als „Knoten“ bezeichnet) liegt. Die beiden Unterpunktwolken werden anschließend jeweils wieder auf die gleiche Weise gesplittet. Diese Teilung wird so oft wiederholt, bis sich jeder Punkt auf einem Knoten befindet[Bentley 1975]. Die entstehenden Freiräume werden in der Literatur oft als Blätter bezeichnet. In der folgenden Abbildung 4-4 ist die Vorgehensweise dargestellt.

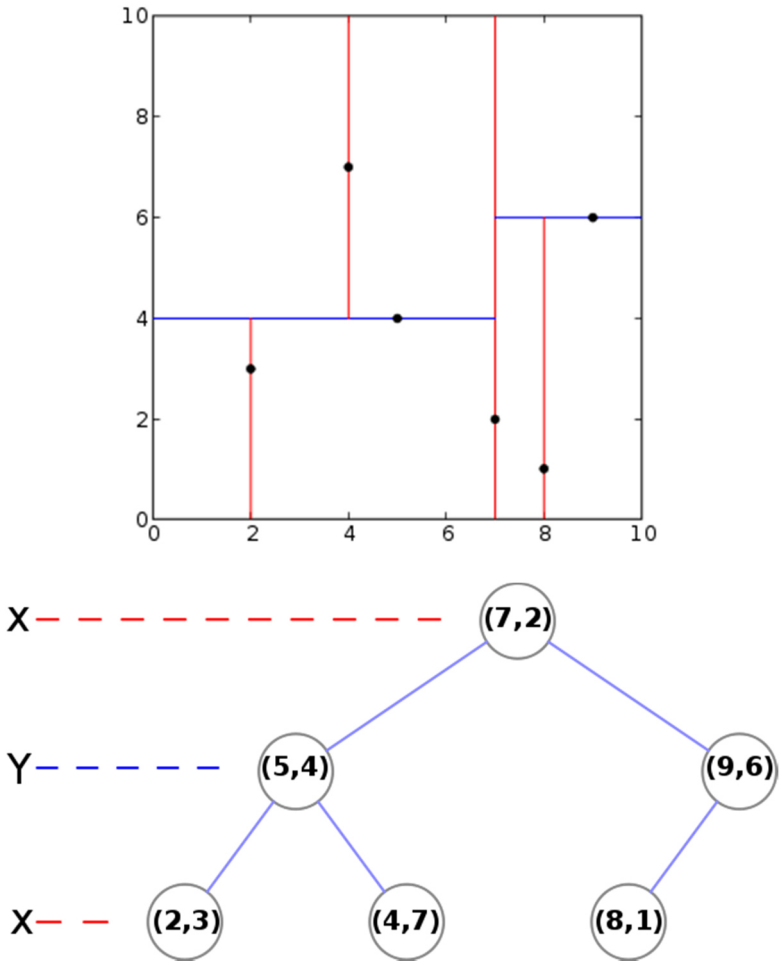

Abbildung 4-4: Erstellung eines kD-Baums aus einem 2D-Datensatz von [Wiki kd-tree]

Soll nun für einen Punkt d_i derjenige Punkt aus der Punktwolke M mit dem geringsten Abstand gefunden werden, wird dieser Prozess durch die Anwendung des kD-Baums beschleunigt. Der Algorithmus beginnt mit dem Punkt an der Wurzel des kD-Baums. Es wird die Distanz von diesem Punkt zu d_i berechnet und als kürzeste Distanz d_{min} gespeichert. Anschließend wird einer der beiden Tochterknoten – zum Beispiel der Tochterknoten des linken Strangs – überprüft. Ist die Distanz von dem Tochterknoten zu p_i geringer als d_{min} wird dieser als Nachbarpunkt mit kleinstem Abstand gespeichert und d_{min} durch die kleinere Distanz ersetzt. Der andere Strang des kD-Baums kann nun komplett ausgeschlossen werden. Diese Iteration wird mit der nächst tieferen Ebene wiederholt. Der Punkt mit dem geringsten Abstand ist gefunden, wenn entweder beide Tochterknoten keinen geringeren Abstand aufweisen oder wenn die unterste Ebene erreicht wurde. Dort entscheidet es sich dann zwischen den zwei übrig gebliebenen Knoten [Wiki kd-tree].

5 Regelung und Steuerung des Roboters

5.1 Regelung

Bei der Regelung liegt im Vergleich zur Steuerung immer ein geschlossener Regelkreis vor. Dieser umfasst normalerweise eine Regelstrecke, einen Regler und einen Abgleich zwischen Regel- und Führungsgröße. Durch die ständige Rückmeldung des Ist-Zustandes ist eine Korrektur hinsichtlich der Führungsgröße und eine Kompensation von Störgrößen gewährleistet [Philippsen 2015]. Die folgende Abbildung 5-1 zeigt den typischen Aufbau eines Regelkreises.

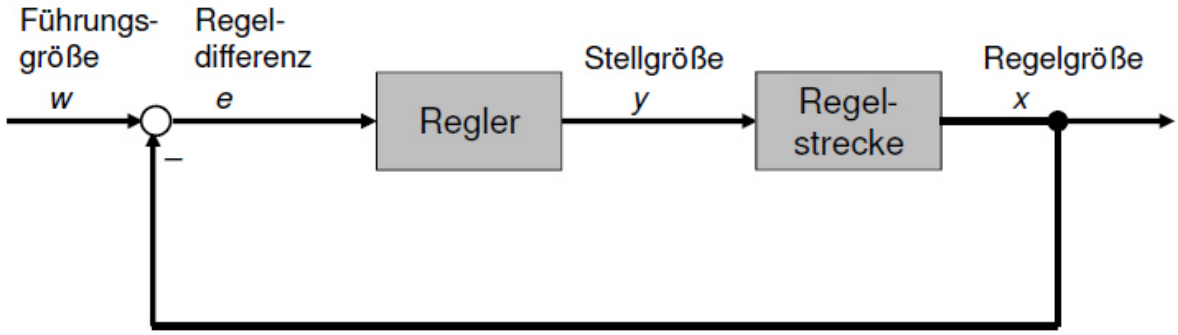

Abbildung 5-1: typischer Aufbau eines Regelkreises aus [Koeppen 2014]

Bei dem Roboter war die erste Idee, einen einfachen Regelkreis im Simulinkmodell aufzubauen. Der Regelkreis führt fortlaufend einen Abgleich aus Ist- und Sollposition durch und ermittelt daraus die zugehörigen Translations- und Rotationsgeschwindigkeiten. Die folgende Abbildung 5-2 und zugehörigen Gleichungen sollen den Soll-und Ist-Abgleich des Roboters verdeutlichen.

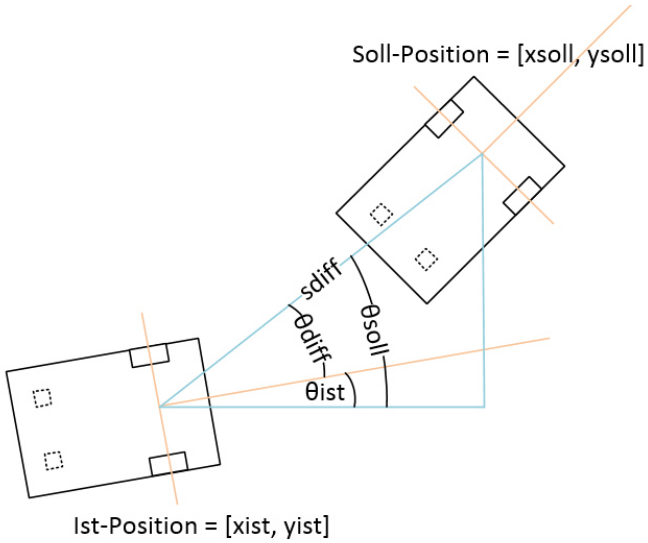

Abbildung 5-2: Soll- und Ist-Abgleich des Roboters

$$s_{diff} = \sqrt{(x_{soll} - x_{ist})^2 + (y_{soll} - y_{ist})^2} \quad 5\text{-}1$$

$$\rightarrow v_{diff} = \frac{s_{diff}}{\Delta t} \quad 5\text{-}2$$

$$\theta_{diff} = \theta_{soll} - \theta_{ist} \quad 5\text{-}3$$

$$\rightarrow \omega_{diff} = \frac{\theta_{diff}}{\Delta t} \quad 5\text{-}4$$

Dabei stellt s_{diff} die Strecke zwischen Ist- und Soll-Position, θ_{diff} die Winkeldifferenz von Ist- und Soll-Position und v_{diff} und ω_{diff} die daraus resultierenden Translations- und Winkelgeschwindigkeiten dar.

Die Größen v_{diff} und ω_{diff} werden jeweils mit einem PD-Regler geregelt und an die Motoren des Roboters – die Regelstrecke – übergeben. Der Differentialanteil kompensiert hierbei Stabilitäts- und Überschwingprobleme, die durch den Proportionalanteil des Reglers entstehen [Hertzberg, Lingemann und Nüchter 2012]. Die Werte für den P-und D-Anteil wurden zunächst in der Simulation am Computer soweit angepasst, dass sich eine perfekte Bahnfahrt ergibt. Im Anschluss wurden die Regelparameter am realen Modell noch geprüft und verfeinert. Die neue Ist-Position wird entweder im Simulinkmodell durch Integration der Geschwindigkeiten (Ist-Position Modell) generiert oder durch Einspeisung der Ist-Position, die mittels Lasersensor ermittelt wurde (Ist-Position Sensor). Die folgende Abbildung 5-3 zeigt den im diesem Absatz beschriebenen Regelkreis.

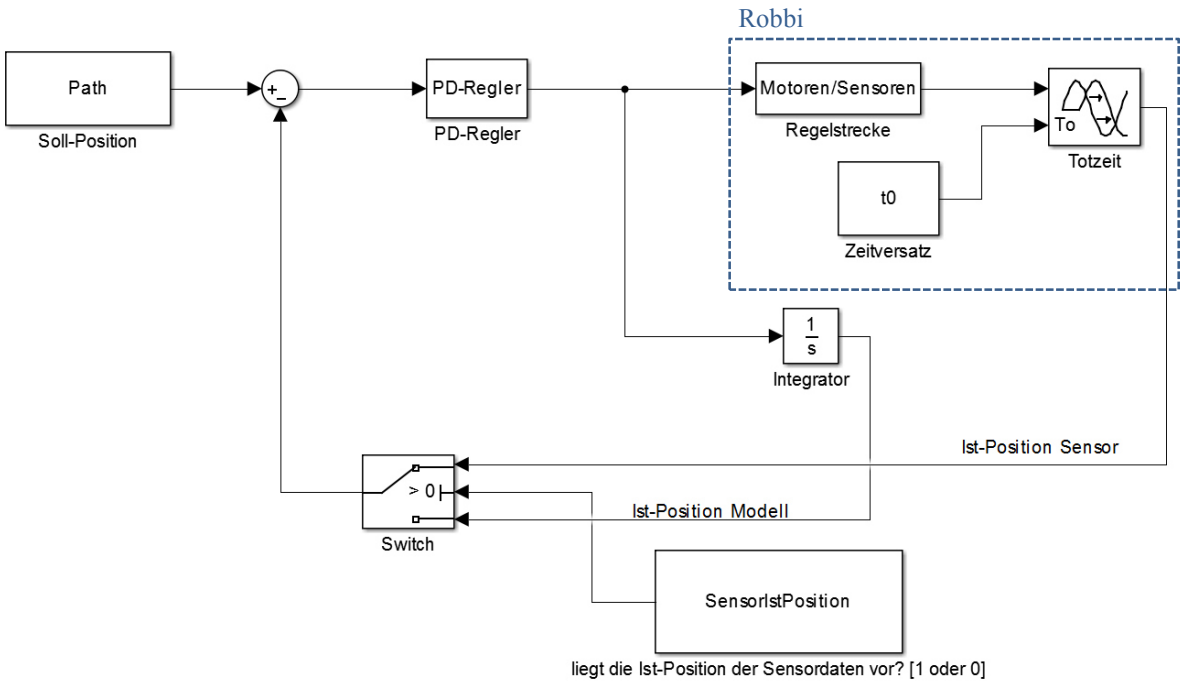

Abbildung 5-3: einfacher Regelkreis des Roboters

Der Abbildung 5-3 ist zu entnehmen, dass die durch die Sensordaten gewonnene Ist-Position mit einem zeitlichen Versatz im Regelkreis ankommt. Der Grund für die Totzeit ist, dass die

Rohdaten des Sensors erst umgewandelt und danach ausgewertet werden müssen. Darunter fallen das 3-Zeichen-Entschlüsseln des ASCII-Codes, die Anwendung des ICP-Algorithmus und eine eventuelle erneute Pfadermittlung beim Auftreten unbekannter Objekte. Die Durchlaufzeiten variieren hierbei stark. Werden die Daten nur entschlüsselt und über das Scanmatching an die Karte angepasst, liegen die Zeiten zwischen 0.3 und 1.1 Sekunden. Die Zeit hängt hierbei davon ab, wie nah die geschätzte Position an der tatsächlichen Position liegt. Je genauer die geschätzte Position ist, desto schneller konvergiert der Scanmatching-Algorithmus. Muss zusätzlich ein neuer Pfad berechnet werden, steigen die Durchlaufzeiten an und liegen zwischen 15 und 22 Sekunden, weshalb der Roboter bei einer Pfadneuberechnung anhält. Der Zeitversatz von der Aufnahme des Laserscans bis zur Übergabe der ausgewerteten Daten an das Simulinkmodell führt bei einem einfachen Regelkreis zu einer oszillierenden Roboterfahrt. Das heißt, während die Ist-Position aus den Sensordaten ermittelt wird, fährt der Roboter weiter in die falsche Richtung. Sobald die Ist-Position des Sensors im Modell ankommt, ändert der Roboter schlagartig seine Richtung. Diese extremen Richtungsänderungen führen nicht nur zu einer unsauberen Bahnführung, sondern bringen auch schlechte Ergebnisse beim Scanmatching mit sich. Ausfälle beim Scanmatching führen wiederum dazu, dass Ist-Positionen durch die Sensorauswertung im Regelkreis seltener zur Verfügung gestellt werden und somit eine genaue Positionsregelung nicht möglich ist.

Den Hinweis für die Lösung dieses regelungstechnischen Problems gab Herr Prof. Dr.-Ing Jochen Maaß, ein Professor des Departments Informations- und Elektrotechnik an der HAW Hamburg. Er gab eine Einführung in die modellbasierte Regelung. Bei der modellbasierten Regelung werden zwei Regelkreise ineinander geschachtelt. Der erste Regelkreis regelt die Roboterfahrt bezüglich des vorgegebenen Pfades und der zweite Regelkreis regelt die Positionsabweichung von Ist-Position Modell zu Ist-Position Laser.
Der Soll-Ist-Abgleich mit den vorgegebenen Pfadpunkten wird hierbei nur mit den vom Modell integrierten Werten durchgeführt, sodass keine sprungartigen Positionsänderungen entstehen. Die zu integrierenden Geschwindigkeiten werden vorher allerdings so manipuliert, dass die Roboterposition im Modell sich in die Richtung der wahren Ist-Position Sensor bewegt. Für die Manipulation ist der zweite Regelkreis zuständig. Dieser führt einen Soll-Ist-Abgleich zwischen Ist-Position Sensor und Ist-Position Modell, die zum selben Zeitpunkt berechnet wurde, als der Sensorscan aufgenommen wurde. Die dabei entstehende Differenz wird mit einem P-Regler angepasst und zu den zu integrierenden Translations- und Rotationsgeschwindigkeiten addiert. Die folgende Abbildung 5-4 zeigt den Aufbau des modellbasierten Regelkreises.

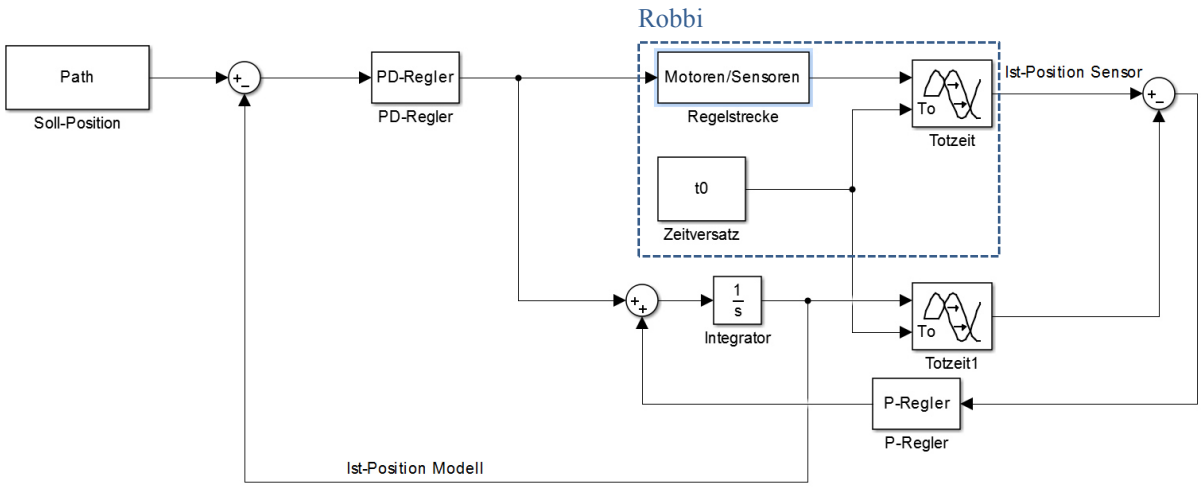

Abbildung 5-4: modellbasierte Regelung des Roboters

Das Modell stellen in der vorhergehenden Abbildung also der Integrator und das Totzeitglied 1 dar. Dieser Regelkreis führt zu deutlich besseren Ergebnissen. Der Roboter macht keine ruckartigen Bewegungen mehr und fährt den Pfad ausreichend genau ab. Leichte Schwankungen sind allerdings immer noch zu beobachten. Um die Schwankungen komplett zu kompensieren, müssten die Geschwindigkeitswerte, die aus dem PD-Regler resultieren, noch eine simulierte Regelstrecke, die dem Verhalten der Regelstrecke des Roboters entspricht, durchlaufen. Nach Herr Prof. Dr.-Ing Jochen Maaß handelt es sich bei dem Roboter näherungsweise um ein IT1-System. Es war allerdings nicht möglich, die Kennwerte des Systems mittels Sprungantwort zu ermitteln, da die Encodersensoren der Motoren nicht ausreichend gut funktionieren. Zu der Funktionsfähigkeit der Encodersensoren wurde eine Messreihe durchgeführt, die in Kapitel 0 thematisiert wird.

5.2 Steuerung

Bei der Steuerung handelt es sich um eine offene Wirkungskette, das heißt, dass der Ist-Zustand nicht überwacht wird. Die Steuerung reagiert auf plötzliche Ereignisse, die während der Roboterfahrt passieren. Folgende Ereignisse werden bei dem Roboter berücksichtigt.

- Wird das Programm gestartet, lokalisiert sich der Roboter und es wird ein entsprechender Pfad zum angegebenen Ziel berechnet und an das Modell übergeben.
- Sobald der Benutzer das Startkommando gibt, fährt der Roboter den geplanten Pfad ab.
- Sobald ein Objekt den Grenzabstand zum Roboter unterschreitet, stoppt der Roboter.
- Sobald das zu nahe gekommene Objekt aus dem Sichtfeld des Roboters entfernt wird, führt der Roboter den geplanten Pfad weiter durch.
- Erscheint ein unbekanntes Objekt in der Umgebung, das sich auf dem geplanten Pfad des Roboters befindet, stoppt der Roboter und es wird ein neuer Pfad berechnet.
- Wurde der Pfad an das Simulinkmodell übergeben, führt der Roboter seine Fahrt zum Ziel mit dem neuen Pfad durch.

- Hat der Roboter die gewünschte Zielposition erreicht, endet das Programm automatisch.

Die Steuerung wird in der folgenden Abbildung 5-5 als Moore-Automat visualisiert. Die zugehörigen Tabelle 5-1, Tabelle 5-2 und Tabelle 5-3 beschreiben die Zustände des Roboters sowie die Ein- und Ausgänge der Steuereinheit.

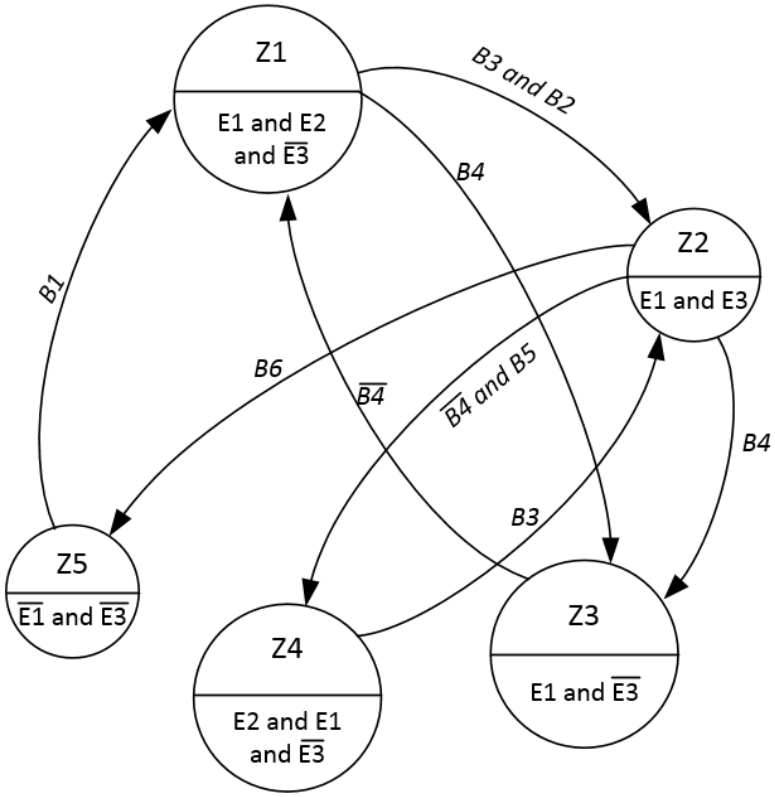

Abbildung 5-5: Steuerung des Roboters im Moore-Automat dargestellt

Tabelle 5-1: Zustände des Roboters

Z1	Programm gestartet
Z2	Der Pfad wird abgefahren
Z3	Es befindet sich ein Objekt im Grenzbereich
Z4	Es befindet sich ein Objekt auf der geplanten Bahn, aber nicht im Grenzbereich
Z5	Roboterfahrt Ende/Programm Ende

Tabelle 5-2: Eingänge der Steuerung

B1	B1 = 1, wenn Programm gestartet
B2	B2 = 1, wenn der Bediener das Startkommando für die Fahrt gegeben hat

B3	B3 = 1, wenn die Berechnung des Pfades abgeschlossen wurde
B4	B4 = 1, wenn ein Objekt den Grenzabstand unterschritten hat
B5	B5 = 1, wenn sich ein Objekt auf der geplanten Bahn findet
B6	B6 = 1, wenn das Ziel erreicht wurde

Tabelle 5-3: Ausgaben der Steuerung

E1	Laser-Scan
E2	Berechnung eines Pfades von der Ist-Position zum Ziel
E3	Motoren laufen variable, entsprechend der Regelung

6 Implementierung

Die Erstellung des Programms teilt sich in zwei Hauptphasen:

1. die Simulation und
2. die Realisierung an den Hardwarekomponenten.

In der ersten Phase wurde der Roboter soweit wie möglich in Simulink/Matlab simuliert, sodass diverse Problemstellungen im Vorhinein gelöst werden konnten. Die verschiedenen theoretischen Konzepte und Algorithmen konnten einfach und schnell ausgetestet werden. Dabei entwickelte sich ein Grundkonzept, das einen einfachen Regelkreis beinhaltet. Außerdem kristallisierte sich während der Simulationsphase eine Navigation über die Gitternetzkarte und Potentialfeldmethode heraus. Der Lasersensor wurde über einen *event listener* simuliert. Dafür wurde eine abgeänderte Ist-Position an den *event listener* übergeben. Dieser hat die zu der Position gehörigen Lasermessdaten erstellt, ausgewertet und als neue Ist-Position an das Simulinkmodell übergeben.
Wird das in der Simulation erstellte Programm auf die Hardwarekomponenten übertragen, fällt schnell auf, dass in der Simulation einige Aspekte nicht berücksichtigt wurden. Es entstehen Probleme aufgrund von Laufzeit oder Datenübertragung. Außerdem wurden einige logische Verhaltensweisen nicht berücksichtigt, die erst durch das reale Testen des Roboters klar wurden. Diese Probleme müssen in der zweiten Phase Schritt für Schritt beseitigt werden.
In den folgenden Unterkapiteln wird das finale Konzept, das auf den Hardwarekomponenten läuft, vorgestellt.

6.1 finales Konzept

Das Programm besteht aus den folgenden vier Einheiten.

- das Initialskript
- das Simulinkmodell
- das Skript zur Laserdatenauswertung
- Bedieneroberfläche im Control Desk

Das Initialskript definiert alle notwendigen Anfangsparameter, auf die sowohl das Simulinkmodell als auch das Skript zur Laserdatenauswertung zugreifen. Es werden die Eigenschaften des Roboters und seiner Umgebung definiert. Darunter fallen die Abmaße, Geschwindigkeits- und Beschleunigungsbegrenzungen, Zielposition, die ungefähre Startposition im Raum, die Karte der Umgebung sowie ihr zugehöriges Potentialfeld, eine initiale Pfadplanung und die Parameter der PD-Regler.
Das Simulinkmodell ist für die Ansteuerung der Motoren zuständig. Es beinhaltet den modellbasierten Regelkreis, der entsprechend Modell und Sensorinformation die Motoren regelt.

Außerdem beinhaltet das Model einen „startstop“-Parameter, mit dem die Roboterfahrt unterbrochen werden kann. Das Modell wird dabei in einen wartenden Zustand versetzt. Es ist zu erwähnen, dass sich das Abfahren des Pfades in drei Abschnitte teilt. Am Anfang dreht sich der Roboter in die Richtung des abzufahrenden Pfades. Wurde die geforderte Orientierung erreicht, fährt der Roboter entlang des Pfades, bis er am Zielpunkt angekommen ist. Durch den gewählten Pfadplanungsalgorithmus erreicht der Roboter den Zielpunkt nicht mit der gewünschten Endorientierung. Deshalb wird im dritten Schritt die Drehung in die Endorientierung durchgeführt.

Das Skript zur Laserdatenauswertung scannt permanent die Umgebung und wertet die Daten aus. Es hat Lese- und Schreibzugriff auf das Simulinkmodell. Nachdem überprüft wurde, ob sich kein Objekt im Grenzbereich befindet, wird dem Modell auf der MicroAutoBox die aktuelle ungefähre Position entnommen und aus dem Laserscan eine Punktwolke um diese Position herum erstellt. Zunächst müssen die unbekannten Objekte der Umgebung herausgefiltert werden, sodass ein gutes Ergebnis beim Scanmatching erzielt werden kann. Anschließend ist eine Ermittlung der neuen Position, die aus den Laserdaten hervorgeht, über den ICP-Algorithmus möglich. Nachdem die neue Position ermittelt wurde, muss noch geprüft werden, ob ein neuer Pfad ermittelt werden muss. Die Ermittlung eines neuen Pfades kann mehrere Auslöser haben, die im Kapitel 6.1.3 genauer erläutert werden.

Die folgende Abbildung 6-1 zeigt die Bedieneroberfläche im Control Desk. Mit dem Eingabefeld startstop/Value kann die Roboterfahrt durch die Eingabe einer Eins gestartet werden. In der zweiten Zeile kann die ungefähre Startposition [x, y, θ_{start}] eingegeben werden sowie die Ist-Position während der Roboterfahrt eingesehen werden. In der dritten Zeile können die Zielkoordinaten [x, y, θ_{Ziel}] eingegeben werden. Diese sind bis zum Erreichen der Zielposition nicht veränderbar.

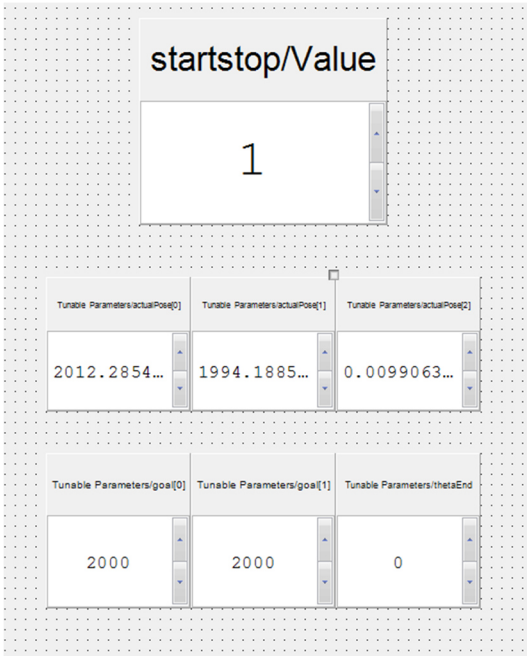

Abbildung 6-1: Bedieneroberfläche im Control Desk

In den folgenden Unterkapiteln 6.1.1, 6.1.2, 6.1.3 und 0 werden für die Programmstruktur besonders prägende Problemstellungen und ihre Lösung vorgestellt.

6.1.1 Kartenerstellung

In dieser Arbeit kennt der Roboter die Karte seiner Umgebung. Damit das Scanmatching bei der Lokalisierung des Roboters sauber funktioniert, ist es notwendig, dass die Karte der Umgebung möglichst genau aufgenommen wird. Für die Erstellung der Karte, wurde ebenfalls der ICP Algorithmus verwendet. Dafür wurde zunächst ein Referenzscan an einer markanten Stelle wie einer Ecke aufgenommen. Anschließend wurde der Roboter mit kleinen Schritten durch den Raum geführt. Dabei wurden die darauffolgenden Scans mittels ICP-Algorithmus an den Referenzscan angepasst. Hat der Algorithmus mit einer ausreichend hohen Genauigkeit funktioniert, wird der aufgenommen Scan zu dem Referenzscan hinzugefügt. Dadurch erweitert sich die Karte Stück für Stück. Die folgende Abbildung 6-2 soll verdeutlichen, wie sich der Referenzscan in kleinen Schritten erweitert. Der Referenzscan im linken Bild wurde im rechten Bild durch einen weiteren Scan erweitert, wodurch eine weitere Ecke des Raumes ergänzt wurde.

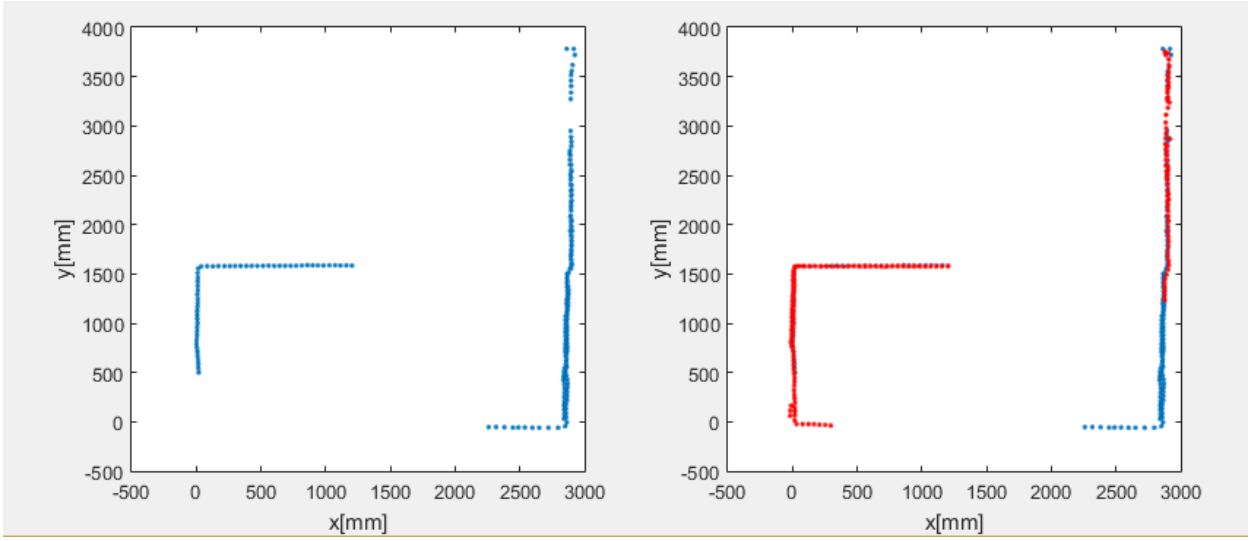

Abbildung 6-2: Referenzscan (links) Referenzscanerweiterung (rechts)

Die folgende Abbildung 6-3 zeigt die fertig erstellte Karte der Testumgebung.

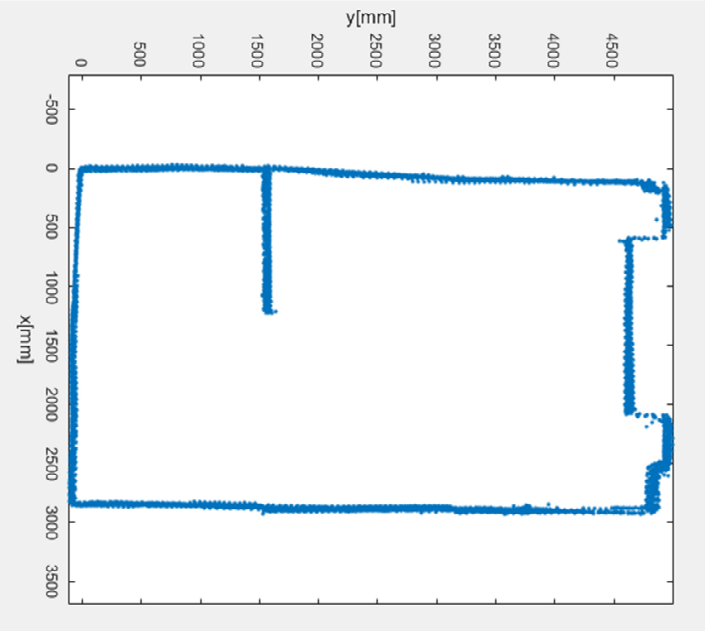

Abbildung 6-3: Karte der Testumgebung

6.1.2 Filterung von unbekannten Objekten

Bevor das Scanmatching über den ICP-Algorithmus durchgeführt wird, ist das Filtern von unbekannten Objekten ein wichtiger Bestandteil. Nimmt der Sensor Daten von Objekten auf, die nicht in dem Scan der Karte vorkommen, führt dies zu schlechten Ergebnissen beim Scanmatching. Aufgrund der zusätzlichen Objekte im Sensorscan, werden die Punktwolken nicht mehr optimal übereinandergelegt – es entsteht meistens ein leichter Versatz zur Ist-Position. Die folgende Abbildung 6-4 zeigt ein beispielhaftes Scanmatching ohne Objektfilterung.

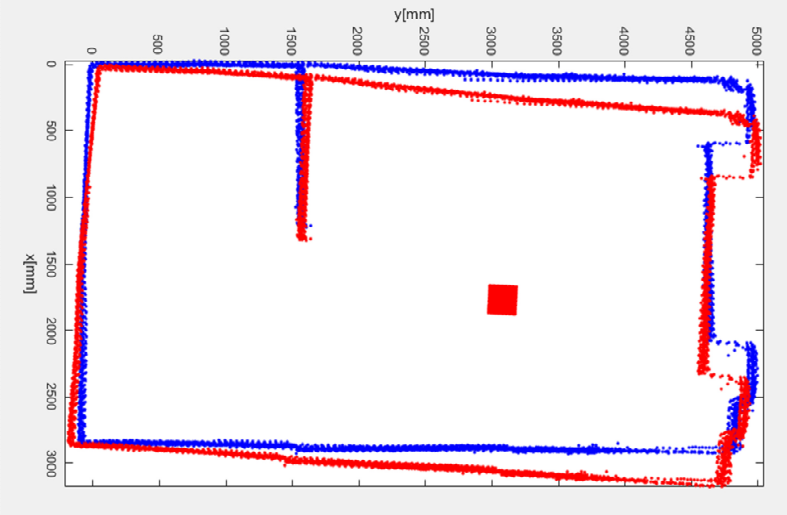

Abbildung 6-4: Scanmatching mit unbekannten Objekten

Für die Filterung von aufgenommenen unbekannten Objekten im Laserscan wurde die Funktion *filterObstacle()* erstellt. Die Funktion vergleicht den aufgenommenen Scan *pointCloudNew* sowohl mit dem vorangegangenen gefilterten Scan *pointCloudOld* als auch mit der Karte *map*. Dabei werden für jeden Punkt aus dem aktuellen Scan der nächste Nachbar aus den anderen beiden Punktwolken bestimmt. Überschreitet ein Punkt eine definierte Grenze (Grenzabstand zum vorigen Scan $d_{1,Grenz} = 100mm$; Grenzabstand zur Karte $d_{2,Grenz} = 300mm$), wird dieser vorerst als unbekanntes Objekt definiert. Die Funktion hat als Rückgabewert die von unbekannten Objekten gefilterte Punktwolke *pointCloudWithoutObstacle* sowie die Punktwolke *pointCloudObstacle*, die alle Messwerte der unbekannten Objekte beinhaltet.

```
function [pointCloudWithoutObstacle, pointCloudObstacle] = filterObsta-
cle(pointCloudNew, pointCloudOld, map)
    pointCloudWithoutObstacle = pointCloudNew;
    if (isempty(pointCloudOld) == false )
        [~,d1] = dsearchn(pointCloudOld', pointCloudNew');
        [~,d2] = dsearchn(map', pointCloudNew');
        delete = find(d1 > 100 & d2 > 300);
        pointCloudObstacle = pointCloudWithoutObstacle(:,delete);
        pointCloudWithoutObstacle(:,delete) = [];
    end
```

Mit der gefilterten Punktwolke ist eine genaue Positionsbestimmung möglich. Nach dem erfolgreichen Scanmatching von gefiltertem Scan zu Karte sollte die Objektfilterung jedoch erneut durchgeführt werden, da es sein kann, dass man Punkte gefiltert hat, die eigentlich zu einem bekannten Objekt gehören. Da der Scan nun optimal auf die Karte passt, werden in diesem Schritt nur die tatsächlichen unbekannten Objekte gefiltert. Diese können dann ausgewertet und weiterverarbeitet werden. Es ist hinzuzufügen, dass der erste aufgenommene Scan im Skript zur Laserdatenauswertung von einer ungestörten Umgebung sein muss, da dieser die erste Vergleichspunktwolke *pointCloudOld* darstellt. Es dürfen sich also anfangs keine unbekannten Objekte in dem Raum befinden.

6.1.3 Pfadplanung

In diesem Abschnitt wird erläutert, wann und wie die Pfadplanung im Skript zur Laserauswertung durchgeführt wird. Die Pfadplanung wird an diesem Roboter nach der Potentialfeldmethode realisiert. Dabei ergeben drei, nun folgende wesentliche Fälle.

- Es wird eine Fahrt von A nach B in der bekannten Karte bestimmt.
- Es wird eine Fahrt von A nach B in einer neuen Umgebung mit zusätzlichen Objekten, die umfahren werden müssen, bestimmt.
- Es kann kein Pfad ermittelt werden.

Die drei Fälle unterscheiden sich enorm in ihrer Rechenzeit. Bei dem ersten Fall wird die Ermittlung des Potentialfelds, das die meiste Rechenzeit in Anspruch nimmt, bereits am Anfang bestimmt. Das Potentialfeld wird nur übergeben und weiterverwendet. Ein Pfad kann ausreichend schnell berechnet werden. Sobald sich die Umgebung verändert und sich ein unbekann-

tes Objekt auf dem geplanten Pfad oder im angrenzenden Sicherheitsabstand befindet, muss ein neuer Pfad, der die Objekte umfährt, an das Simulinkmodell übergeben werden. Dafür muss auch ein neues Potentialfeld für die neue Umgebung ermittelt werden. Dies kann bei der Genauigkeit und Größe der Karte auf dem verwendeten Laptop bis zu 15 Sekunden dauern. Aus diesem Grund stoppt der Roboter zur Berechnung des neuen Pfades. Nachdem die Berechnung abgeschlossen ist, führt der Roboter seine Fahrt mit dem neuen Pfad fort. Der dritte Fall tritt ein, wenn sich aufgrund von zu vielen oder zu breiten Objekten die Umgebung in zwei nicht zusammenhängende Räume teilt. Der Roboter kann nicht an den Objekten vorbeifahren und es lässt sich somit auch kein Potentialfeld ermitteln. Folglich entsteht im Programm eine Endlosschleife. Diese Situation wurde so gehandhabt, dass die Berechnung des Potentialfeldes nach der Überschreitung einer bestimmten Zeit t_{grenz} abgebrochen wird und der Roboter auf seinem alten Pfad weiterfährt bis sich das Objekt schlussfolgernd irgendwann im Grenzbereich befindet. Der Roboter erwartet dann, dass das Objekt von einem Menschen entfernt wird.

6.1.4 Problematik beim Auslesen der Encoder

Die anfängliche Idee des Roboterkonzeptes war es, die Encoder der Motoren auszulesen, sodass man auch ohne Laserdaten eine möglichst genaue Position hat. Dies kann von Vorteil sein, wenn der Laser für ein paar Sekunden keine Positionsdaten liefern kann. Der Grund für eine gescheiterte Lokalisierung kann zum Beispiel ein ungenügendes Ergebnis beim Scanmatching sein. Nach einiger Zeit fiel allerdings auf, dass die Encoder sehr ungenau sind. Um diese Vermutung zu validieren, wurde eine Messreihe durchgeführt. In dieser Messreihe fährt der Roboter 100, 200 und 300 cm mit verschiedenen, aber konstanten Geschwindigkeiten ab. Dabei wird die angegebene gefahrene Strecke der Encoder mit der tatsächlich gefahrenen Strecke, die mit der Hilfe eines Maßbandes ermittelt wurde, verglichen. Die folgende Abbildung 6-5 visualisiert den Versuchsaufbau.

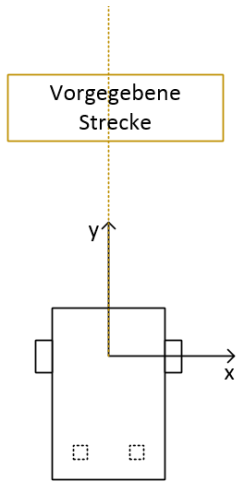

Abbildung 6-5: Versuchsaufbau Encoderversuch

Die Testgeschwindigkeiten liegen bei einem PWM von 45% - 10%. Der Geschwindigkeitsbereich ergibt sich aus den Geschwindigkeitsbegrenzungen des Roboters. Es wurde eine Geschwindigkeitsbegrenzung von $v_{max} = 22.5 \frac{mm}{s}$ und von $\omega_{max} = \frac{v_{max}}{B1} * 2$ im Modell definiert. Das heißt, die maximal Geschwindigkeit, die ein Rad erreichen kann, ist $v_{L/R,max} = v_{max} + \omega_{max} * B1 = 67.5 \frac{mm}{s}$. Diese maximal erreichbare Geschwindigkeit entspricht nach der Formel 2–9 einem PWM-Signal von 12.08%. Die folgende
Tabelle 6-1 beinhaltet die Werte der Messreihe.

Tabelle 6-1: Messreihe Encoderversuch

vorgegebene Strecke [cm]	PWM	*gemessen durch Encoder* x [cm]	y [cm]	*gemessen mit Maßband* x [cm]	y [cm]
100	0.45	-2.80	95.00	0.00	62.00
200	0.45	4.60	195.00	-2.00	127.00
300	0.45	1.30	295.00	2.50	190.00
100	0.40	-1.60	95.00	0.00	75.00
200	0.40	-8.50	195.00	2.50	154.00
300	0.40	-8.50	295.00	4.00	234.00
100	0.35	-1.90	95.00	0.00	80.00
200	0.35	-7.00	195.00	-0.50	166.00
300	0.35	-18.90	295.00	3.50	252.00
100	0.30	-1.50	95.00	1.50	82.00
200	0.30	-7.30	195.00	-1.50	173.50
300	0.30	-5.40	295.00	-8.50	263.50
100	0.20	-1.40	95.00	-4.50	89.00
200	0.20	-5.30	195.00	-5.50	180.00
300	0.20	-11.70	295.00	-9.50	275.00
100	0.10	0.10	95.00	1.00	90.00
200	0.10	2.30	195.00	2.50	184.00
300	0.10	1.10	295.00	-3.00	279.00

Aus den Messwerten lassen sich keine Abhängigkeiten in die x-Richtung erschließen. Die gefahrene Strecke in x-Richtung steigt nicht immer mit der Länge der vorgegebenen Strecke. Außerdem ist keine Proportionalität zwischen den gemessenen Werten der Encoder und den gemessenen Werten mit dem Maßband zu erkennen. Die Auswertung der Distanz in x-Richtung wird somit ausgeschlossen, da einerseits keine Abhängigkeiten zu erkennen sind, andererseits die Distanzen in x-Richtung zu gering sind, um mit den gegebenen Messeinheiten aussagekräftige Ergebnisse zu erzielen. Bei den aussagekräftigeren Messwerten in die y-Richtung sind Abhängigkeiten zu erkennen. Abbildung 6-6, Abbildung 6-7 und Abbildung 6-8 zeigen die gemessenen Strecken in Abhängigkeit von dem PWM-Signal über die drei verschiedenen Distanzen.

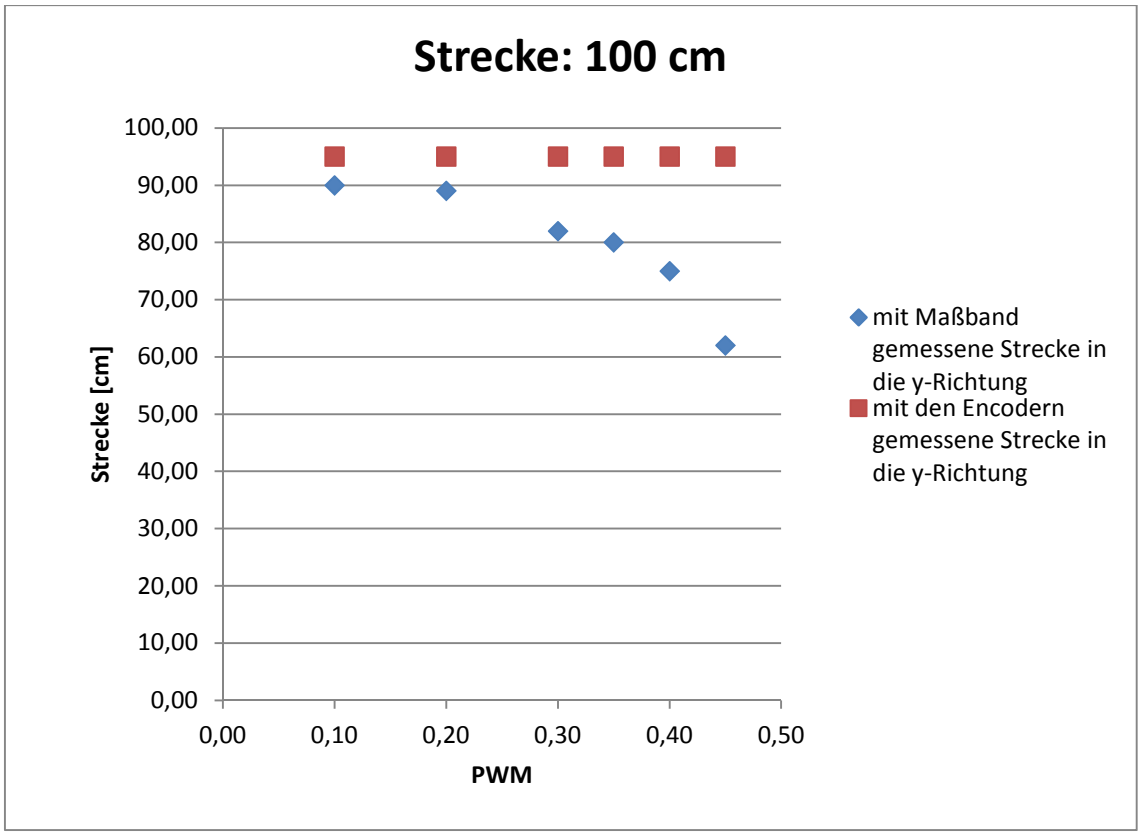

Abbildung 6-6: Strecke in Abhängigkeit des PWM-Signals über 100 cm

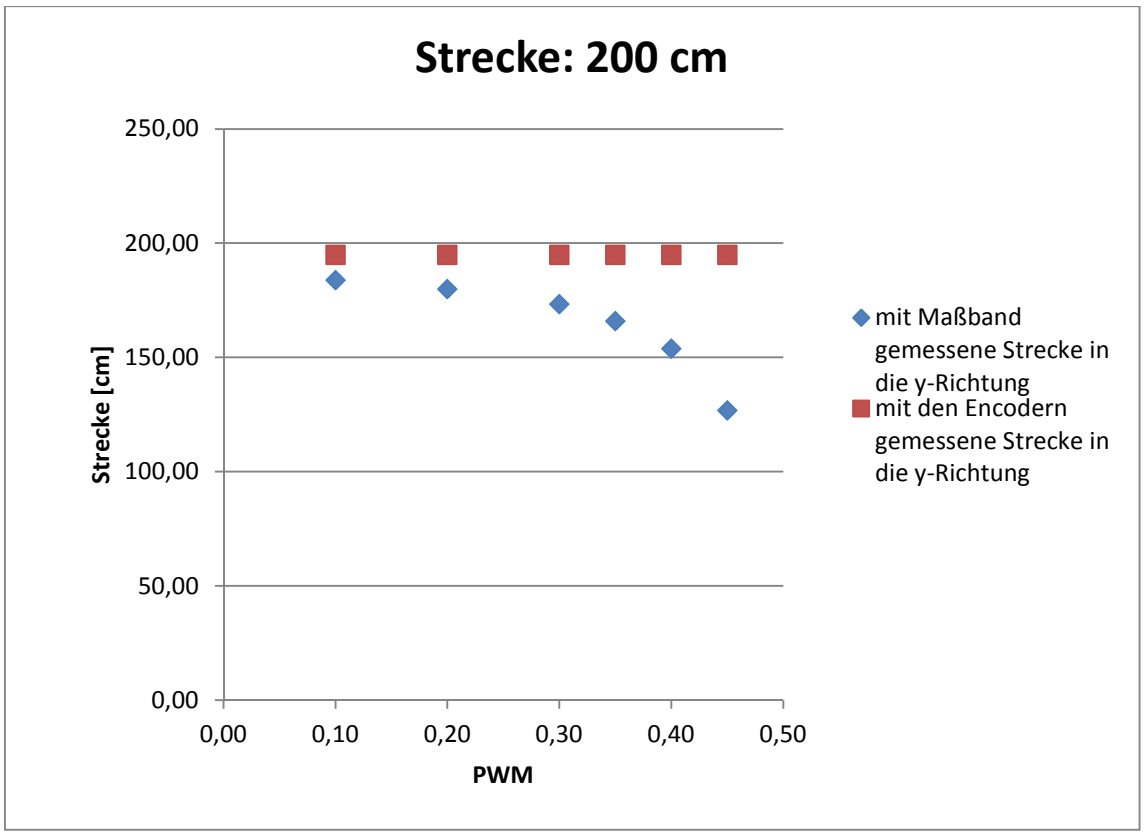

Abbildung 6-7: Strecke in Abhängigkeit des PWM-Signals über 200 cm

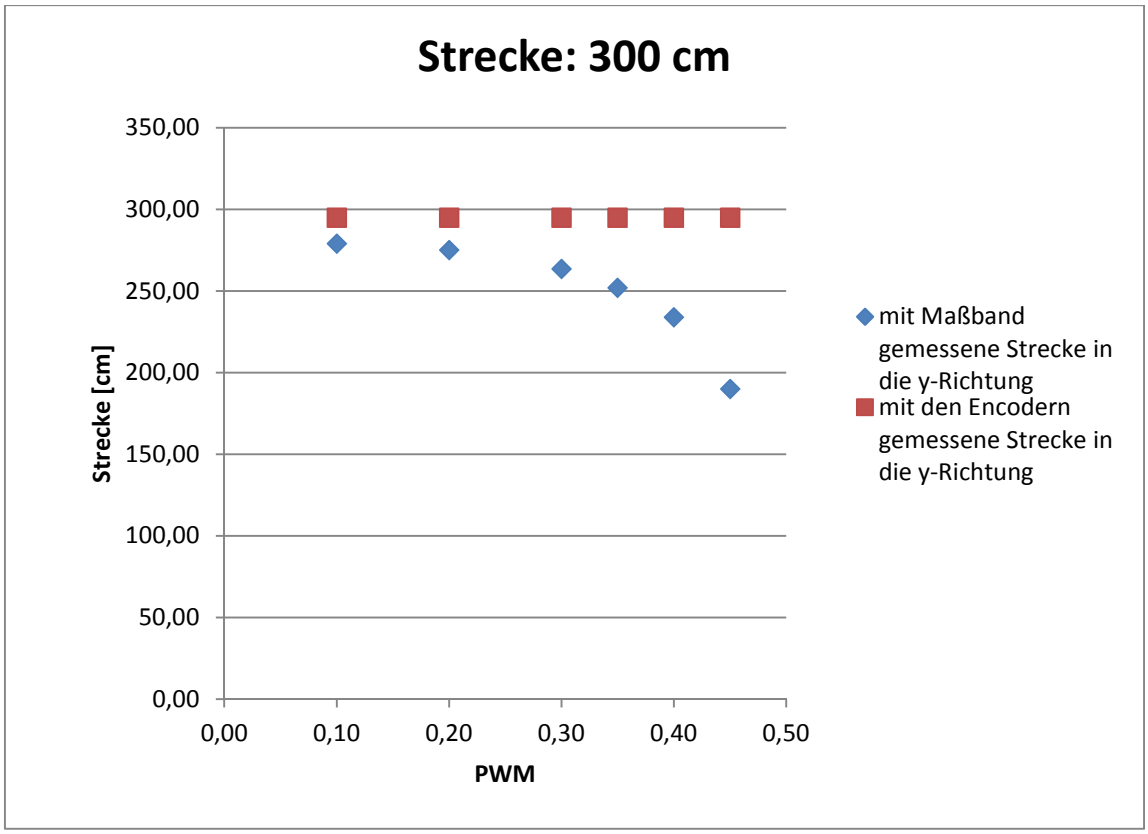

Abbildung 6-8: Strecke in Abhängigkeit des PWM-Signals über 300 cm

Abbildung 6-6, Abbildung 6-8 und Abbildung 6-8 ist zu entnehmen, dass die Genauigkeit der Encoder mit steigender Geschwindigkeit zunimmt. Außerdem kann man vereinfacht sagen, dass die gemessene Strecke mit dem Maßband proportional zur gemessenen Strecke mit den Encodern bei gleicher Geschwindigkeit steigt. Eine längere Strecke erhöht demnach die Abweichung um den Anteil der Verlängerung. Der Roboter befindet sich während seiner Roboterfahrt überwiegend in den unteren Geschwindigkeitsbereichen, da die maximale Geschwindigkeit der Räder von $v_{L/R,max} = 67.5 \frac{mm}{s}$ nur in Kurven erreicht werden kann. Aus diesem Grund sind die Odometriedaten meistens sehr ungenau und unbrauchbar. Als Fazit der Messreihe wurden die Encoder in dem Robotermodell nicht mehr berücksichtigt. Als alternative Lösungsvariante wurde die Roboterregelung – wie bereits erwähnt – in einem modelbasiertem Regelkreis realisiert.

6.2 verwendete Funktionen

In diesem Abschnitt wird eine Beschreibung über die in dem Initialisierungsskript „Initialize“ und dem Skript zur Laserdatenauswertung „Localisation“ verwendeten Funktionen, die nicht aus der Matlab-Library oder der HIL API stammen, gegeben. In der folgenden Abbildung 6-9 ist dargestellt, welche Funktionen es gibt und wie diese ineinander geschachtelt sind.

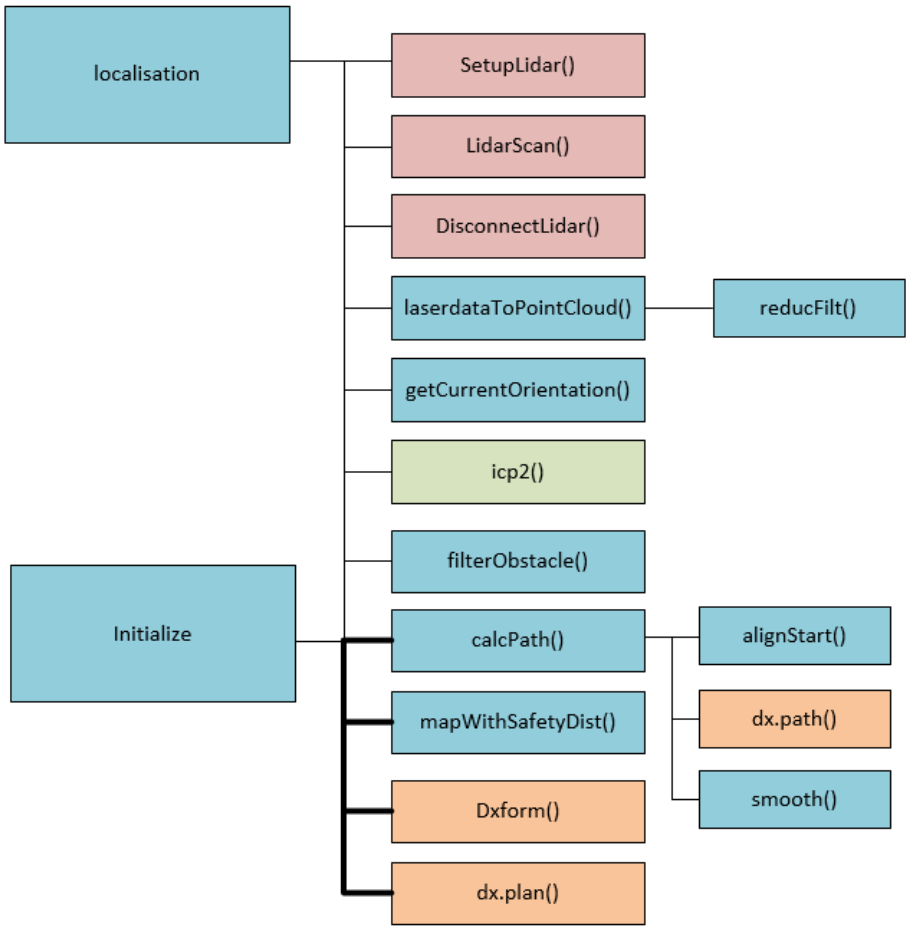

Abbildung 6-9: Übersicht der verwendeten Funktionen

lidar = SetupLidar(Port)

- Erstellt und initialisiert eine serielle Verbindung *lidar* zum Lasersensor, der an dem Com Port *Port* angeschlossen. Dabei werden allgemeine Informationen des Lasers ermittelt. Außerdem wird das Laserscannen gestartet.
- Herkunft: Matlab File Exchange von S. Shrestha [Shrestha 2012]; wurde modifiziert

rangescan = LidarScan(lidar)

- Der letzte aufgenommene Laserscan im Bereich von -120° bis 120° wird über die serielle Datenverbindung *lidar* übertragen. Diese Dateninformation wird so dekodiert, dass im Rückgabewert *rangescan* Abstandswerte in mm zur Verfügung stehen.
- Herkunft: Matlab File Exchange von S. Shrestha [Shrestha 2012]

DisconnectLidar(lidar)

- Die serielle Verbindung *lidar* wird geschlossen.
- Herkunft: Matlab File Exchange von S. Shrestha [Shrestha 2012], wurde modifiziert

pointCloud = laserdataToPointCloud(scan, position, Medfilt, Redfilt)

- Der Laserscan *scan,* der aus der Funktion LidarScan() hervorgeht, wird von Polarkoordinaten in kartesische Koordinaten um einen Bezugspunkt *position* umgerechnet. Dabei kann zusätzlich der Reduktions- und Medianfilter angewandt werden, sofern die Parameter *Medfilt* und *Redfilt* größer als null sind. *Medfilt* gibt an, dass für jeden Punkt $x(i)$ ein Mittelwert über den Bereich $x(i\ -\ (Medfilt-1)/2 : i\ +\ (Medfilt-1)/2)$ gebildet wird. *Redfilt* repräsentiert einen Radius und definiert den Umkreis, in dem alle Punkte durch Mittelwertbildung zu einem Punkt zusammengefasst werden. Der Rückgabewert ist die gefilterte Punktwolke *pointCloud*.
- Herkunft: R. Güldenring

currentOrientation = getCurrentOrientation(pointCloud, RT, TT, currentPosition, positionOfModel)

- Die Funktion berechnet aus den Eingangsparametern die Orientierung *currentOrientation* in rad, die aus den Laserdaten hervorgeht. Dabei sind *RT* und *TT* die Rotations- und Translationsmatrizen, die aus der Funktion icp2() hervorgehen. *pointCloud* ist die unveränderte Punktwolke, die aus der Funktion laserdataToPointCloud() hervorgeht. *currentPosition* ist die die Ist-Position, die nach dem Scanmatching berechnet wird und *positionOfModel* ist die Position, an der sich der simulierte Roboter aus dem Simulinkmodell befindet.
- Herkunft: R. Güldenring

[RT, TT, ER] = icp2(q, p, k)

- Gibt die Rotationsmatrix *RT* und die Translationsmatrix *TT* zurück, um die Punktwolke *p* auf die Punktwolke *q* zu verschieben (RT*q + TT). Mit *k* kann man die Anzahl der Iterationen festlegen, wobei der Standartwert bei 10 Iterationen liegt. *ER* liefert einen (k+1)-langen Vektor mit dem quadratischen Mittelwert des Fehlers für jede Iteration, wobei ER(0) der initiale Error ist.
- Herkunft: Matlab File Exchange von J. Wilm [Wilm 2013]

[pointCloudWithoutObstacle, pointCloudObstacle] = filterObstacle(pointCloudNew, pointCloudOld, map)

- Die aktuelle Punktwolke *pointCloudNew* wird auf unbekannte Objekte untersucht und gefiltert. Dabei wandern alle Punkte, die zu einem unbekannten Objekt gehören, in das Array *pointCloudObstacle*. Diese werden zusätzlich aus der aktuellen Punktwolke gelöscht, die als *pointCloudWithoutObstacle* zurückgegeben wird. Zum Filtern der Objekte wird die aktuelle Punktwolke mit der in der davor aufgenommenen Punktwolke *pointCloudOld* und der Umgebungskarte *map* (als Punktwolke) verglichen.
- Herkunft: R. Güldenring

[tableDataX, tableDataY, tableTime, thetaBegin, path] = calcPath(position, res, map2d, dx, pathlength, vlim)

- Diese Funktion ermittelt aus der Ist-Position *position* und dem Potentialfeld *dx* einen neuen Pfad. Der Pfad wird geglättet und als x- und y-Vektor in *tableDataX* und *tableDataY* zurückgegeben. *tableTime* ist der zugehörige Zeitvektor, der die Geschwindigkeit des Roboters bestimmt. Dabei muss die maximale Geschwindigkeit *vlim* einberechnet werden. *pathlength* definiert die Länge der Pfadvektoren, da diese im Initialskript festgelegt wurden und im Simulinkmodell nicht mehr verändert werden dürfen. Die Umgebungskarte mit Sicherheitsabstand *map2d* wird als Gitternetzkarte an die Unterfunktion alignStart() übergeben.
- Herkunft: R. Güldenring

map2d = mapWithSafetyDist(safetyDist, map)

- Die Funktion erstellt aus der Umgebungskarte *map* im Rasterformat eine Rasterkarte *map2d* mit dem gegebenen Sicherheitsabstand *safetyDist*. Der Sicherheitsabstand sollte mindestens dem Radius des Roboterumfangs entsprechen.
- Herkunft: R. Güldenring

dx = DXform(map)

- Es wird ein Objekt zur Navigation mit der Potentialfeldmethode erstellt. *map* ist die Karte, in der der Pfad geplant wird. Die Karte wird als Rasterkarte übergeben.
- Herkunft: Robotics Toolbox by Peter Corke [Corke 2013]

ER2 = dx.plan(goal)

- *dx* ist das Objekt, das aus dem Konstruktor DXform() hervorgeht. Mit den in dem Objekt enthaltenen Informationen wird das Potentialfeld für die Umgebung bezüglich des übergebenen Ziels *goal* ermittelt. Überschreitet die Potentialfeldermittlung eine Zeit von 20 sec, wird diese abgebrochen und der Rückgabewert *ER2* ist eine eins, sonst eine null.
- Herkunft: Robotics Toolbox by Peter Corke [Corke 2013], modifiziert

path = dx.path(start)

- *dx* ist das Objekt, das aus dem Konstruktor DXform() hervorgeht. Bevor diese Methode angewandt wird, sollte das Potentialfeld über dx.plan(goal) bestimmt worden sein. Mit der Methode path() wird nun der kürzeste Pfad von der Startposition *start* zum Ziel ermittelt.
- Herkunft: Robotics Toolbox by Peter Corke [Corke 2013]

pointCloudRed = reducFilt(pointCloud, r)

- Auf die Punktwolke *pointCloud* wird der Reduktionsfilter aus Kapitel 4.2 angewandt. *r* entspricht hierbei dem Radius für den Umkreis, in dem die Punkte gemittelt werden sollen. *pointCloudRed* ist die gefilterte Punktwolke mit reduzierter Arraylänge.
- Herkunft: R. Güldenring

start = alignStart(start_in, mapMatrix)

- Diese Funktion ist notwendig, wenn sich der Roboter im Sicherheitsabstand der Karte befindet. Ist dies der Fall, kann über dx.path() kein Pfad bestimmt werden, da sich der Roboter nach der Definition im Objekt befindet. Es gibt allerdings Fälle, wo sich der Roboter zwar im Sicherheitsabstand befindet, dem Objekt allerdings noch nicht zu nah ist. Um diesen Konflikt zu umgehen, wird ein Startpunkt *start* ermittelt, der der eigentlichen aktuellen Position *start_in* am nächsten ist und sich zusätzlich in einer freien Zelle der Rasterkarte *mapMatrix* befindet.
- Herkunft: R. Güldenring

smoothedPath = smooth(path, grade)

- Der aus dx.path() ermittelte Pfad *path* wird um den Grad *grade,* wie in Kapitel 3.2.5 beschrieben, geglättet. Der geglättete Pfad wird in *smoothedPath* als Array zurückgegeben.
- Herkunft: R. Güldenring

6.3 Simulinkmodell

Bei dem Simulinkmodell handelt es sich um eine sehr komplexe Verknüpfung von Blöcken und Funktionen. Um die Übersicht zu bewahren, wird das Modell in Abbildung 6-10 in relevante Teilbereiche aufgeteilt und im Anschluss mit einer zugehöriger Abbildung genauer erläutert. Das komplette Blockschaltbild ist im Anhang A.4 zu finden.

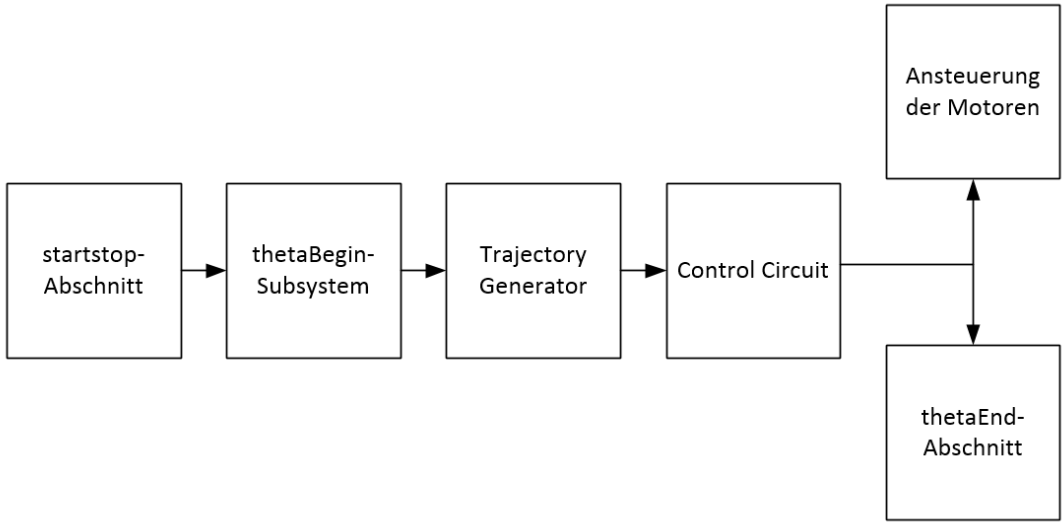

Abbildung 6-10: relevante Bereiche des Simulinkmodells

startstop-Abschnitt

Die Konstante *startstop* ist ein wichtiger Bestandteil des Simulinkmodells. Wird diese auf Null gesetzt, wird das Modell in den Wartezustand versetzt. Das heißt, im Regelkreis bleibt der Roboter auf der aktuellen Position stehen, die PWM-Eingänge werden auf 0.5 gesetzt und es werden keine weiteren Sollpositionen aus dem Trajektoriengenerator übergeben. Bevor der *startstop*-Wert an die einzelnen Parteien weitergeleitet wird, muss er allerdings noch die *FilterFunction1* durchlaufen. Diese Funktion deckt den Fall ab, wenn der Roboter die gewünschten x-und y-Koordinaten des Ziels erreicht hat *(reqEndPosition* = 1). Der Roboter soll also nicht mehr weiterfahren, sich allerdings noch zur Endorientierung drehen. Der Ausgang *filteredStart* wird also beim Erreichen des Ziels bereits auf null gesetzt, damit nur noch eine Drehung im Modell gewährleistet wird.

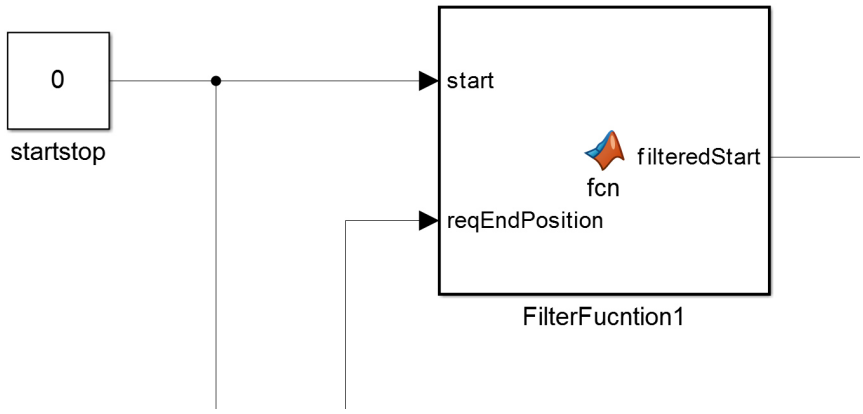

Abbildung 6-11: startstop-Abschnitt

thetaBegin-Subsystem

Sobald die *start*-Variable auf Eins gesetzt wurde, wird überprüft, ob der Roboter in die Richtung des Pfades *thetaBegin* ausgerichtet ist. Ist dies nicht der Fall, dreht sich der Roboter mit einer konstanten Winkelgeschwindigkeit von *wBegin* in die Richtung der Anfangsorientierung. *wBegin* wird dabei durch den Regelkreis geleitet. *currentModelTheta* ist dabei die Richtung, in der sich der Roboter momentan befindet. Wurde die Anfangsorientierung erreicht, wird die Variable *startDrivingPath* auf Eins gesetzt. Sie gibt den Startschuss für den Trajektoriengenerator, der im nachfolgenden Abschnitt erläutert wird.

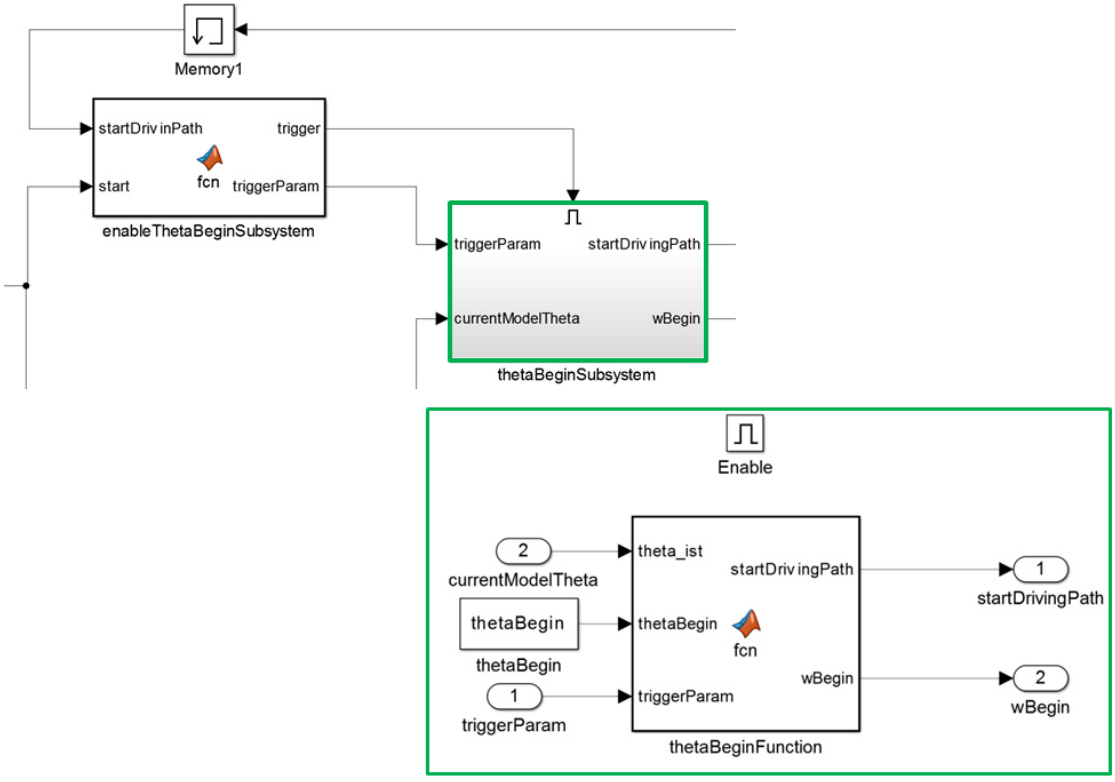

Abbildung 6-12: thetaBeginSubsystem

trajectoryGenerator

In dem Trajektorientgenerator ist der abzufahrende Pfad mit zugehörigem Zeitvektor in einer Look-Up-Table gespeichert. Sobald der Roboter seine Anfangsausrichtung erreicht hat (*startDrivingPath* = 1) und *start* Eins ist, übergibt der Trajektoriengenerator die Soll-Position *setPosition* nach und nach an den Regelkreis. Wie schnell die Werte des Pfades übergeben werden, ist in dem Zeitvektor der Look-Up-Table definiert. Ist die Differenz von Ist- und Soll-Position im Regelkreis-Subsystem zu groß, wird der *hold*-Parameter auf eins gesetzt. Der Trajektoriengenerator wartet dann so lange, bis der Roboter die Distanz aufgeholt hat und führt die Soll-Position-Übergabe fort.

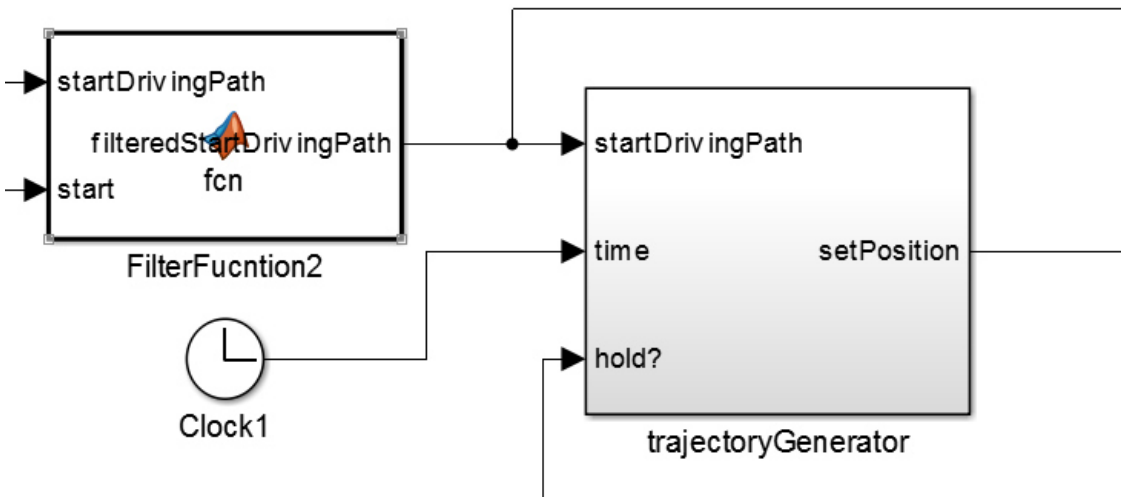

Abbildung 6-13: trajectoryGenerator

Control Circuit

Bei diesem Subsystem handelt es sich um das komplexeste Subsystem, weshalb das Blockschaltbild aus Platzgründen nur im Anhang A.4 zu finden ist. In dem Subsystem ist der in Kapitel 5.1 beschriebene modellbasierte Regelkreis realisiert. Des Weiteren werden die Rotationsgeschwindigkeiten aus der Anfangs- und Enddrehung (*wBegin, wEnd*) durch den Regelkreis geschleust. Am Ausgang des Subsystems liegen die zu fahrenden Translations- und Winkelgeschwindigkeiten (*v,omega*), die Ist-Position des Modells *[currentModelX, currentModelY, currentModeltTheta]* und der Parameter *hold*, der an den Trajektoriengenerator übergeben wird, an.

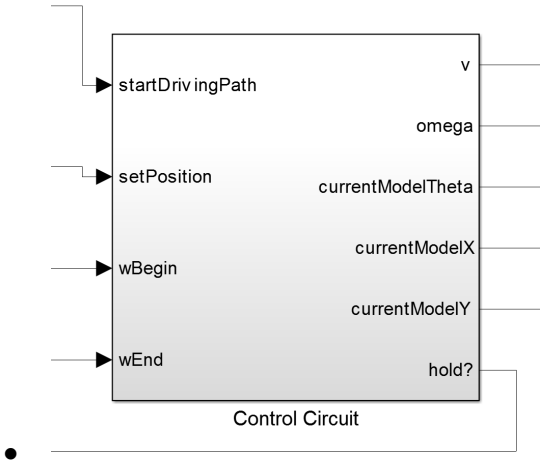

Abbildung 6-14: Control Circuit

Ansteuerung der Motoren

Die aus dem Regelkreis resultierende Translations- und Rotationsgeschwindigkeiten (*v, omega*) werden in dem Subsystem Differentialantrieb nach Kapitel 2.1 in die entsprechenden Geschwindigkeiten am linken und am rechten Rad des Roboters umgerechnet. In dem Subsystem PWM-Signal werden aus den Geschwindigkeitsinformationen nach Kapitel 2.2 die notwendige Pulsweitenmodulation *PWMR* und *PWML* ermittelt. Diese PWM-Signale werden mit der konstanten Periode *PWM-Period* über die von dSpace erstellten Blöcke an die Motoren übergeben.

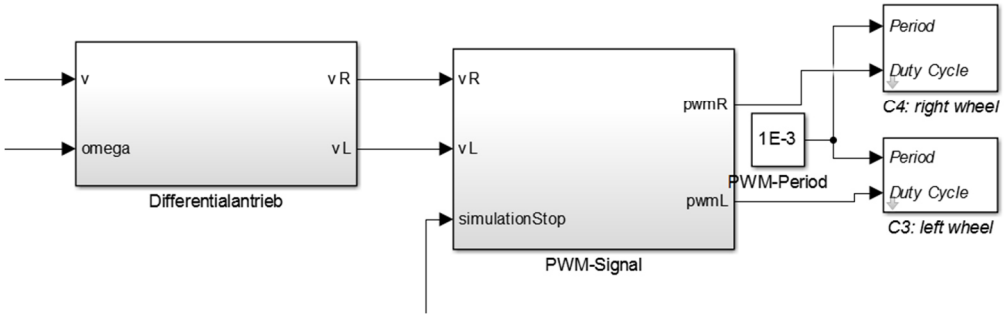

Abbildung 6-15: Anteuerung der Motoren

thetaEnd-Abschnitt

Im thetaEnd-Abschnitt wird in der Matlabfunktion *GoalReached* durchgehend überprüft, ob der Roboter die x- und y-Koordinaten des Ziels *goal* erreicht hat. Dafür werden die Koordinaten des Ziels mit denen der Ist-Position *[currentModelX, currentModelY]* aus dem Regelkreis verglichen. Wird die Zielposition erreicht, übergibt die Funktion mit der Variable *reqEndPosition* eine Eins an die Matlabfunktion *thetaEndReached*. Diese Funktion stellt sicher, dass der Roboter die gewünschte Zielorientierung *thetaEnd* erreicht. Dafür dreht sich der Roboter so lange mit der Winkelgeschwindigkeit *wEnd* bis die Endorientierung erreicht ist. Sobald dieser Fall eintritt, wird mit *simulationStop* das PWM-Signal auf 0.5 gesetzt und die Simulation gestoppt.

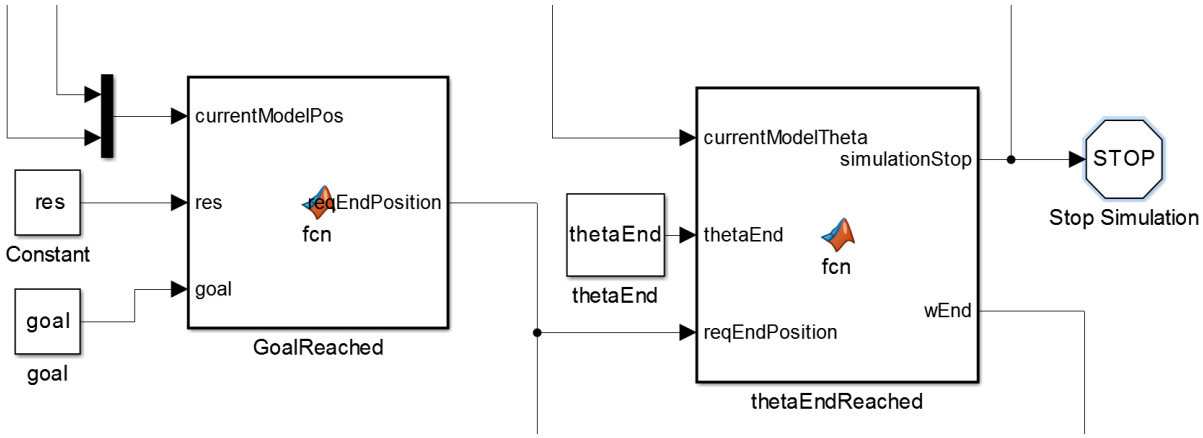

Abbildung 6-16: thetaEnd-Abschnitt

7 Zusammenfassung und Ausblick

Realisiert wurde ein Roboter, der mit einer durchschnittlichen Geschwindigkeit von ungefähr $25\,\frac{mm}{s}$ einen vorgegebenen Pfad, der den Roboter von A nach B bringt, abfährt. Werden dem Roboter unbekannte Gegenstände in den Weg gestellt, weicht dieser den Objekten aus und kommt – vorausgesetzt das Objekt befindet sich nicht auf dem Zielpunkt – am angegebenen Ziel an. Kommt der Roboter einem Objekt oder einer Wand zu nahe, stoppt dieser und wartet bis der Gegenstand entfernt wird. Zusammenfassend wurden die Grundfunktionen eines mobilen Roboters in dieser Bachelorarbeit realisiert. Damit der Roboter mit dem heutigen Fortschritt der Robotertechnik annähernd mithalten kann, sind allerdings noch weitere Punkte zu realisieren. Die Optimierungsmöglichkeiten werden im Folgenden besprochen.

- Wie in Kapitel 0 erwähnt, bewegt sich der Roboter mit einer Geschwindigkeitsbegrenzung von $v_{max} = 22.5\,\frac{mm}{s}$ und $\omega_{max} = \frac{v_{max}}{B} * 2$. Um die Effizienz zu erhöhen, sollten höhere Geschwindigkeiten erreicht werden. Es wurden bereits Testfahrten mit der doppelten Geschwindigkeit durchgeführt. Dabei fiel auf, dass die Roboterfahrt auf geraden Strecken und leichten Kurven bei doppelter Geschwindigkeit gut funktioniert. Zwei kritische Stellen sind allerdings Kurven mit einer starken Krümmung und Objekte, die sich im Grenzbereich des Roboters befinden. In den starken Kurven erreicht der Roboter seine Höchstgeschwindigkeit. Je höher die Geschwindigkeit, desto seltener wird – bezogen auf die gefahrene Strecke – ein Abgleich von Ist-Position Laser zu Ist-Position Modell vollzogen. Es wird seltener eine Korrektur der Ist-Position Modell durchgeführt, die zugleich als Schätzposition für das Scanmatching dient. Bei einer schlechten Schätzung führt der Scanmatching-Algorithmus zu einem ungenügenden Ergebnis, sodass keine Ist-Position Laser ermittelt werden kann. Die zweite kritische Stelle sind Objekte, die dem Roboter zu nahe kommen. Der Roboter fährt auf das Objekt zu und sobald es in den Grenzbereich eintritt, kann es sein, dass der Laser aufgrund der Prozesszeiten erst wieder eine Aufnahme macht, wenn der Roboter bereits gegen den Gegenstand stößt. Der Roboter stoppt also zu spät. Als Lösungsansatz würde die Verringerung der Prozesszeiten des Laserskriptes oder eine Trajektorienplanung, die die Geschwindigkeit des Roboters bestimmt, dienen. Bei der Trajektorienplanung sollte der Roboter auf freien, langen Strecken schneller fahren und in Kurven oder in der Nähe eines Objektes seine Geschwindigkeit verringern.

- Die Genauigkeit des Roboters könnte verbessert werden. Im Moment erreicht der Roboter die Zielposition mit einer ungefähren Abweichung von bis zu 10 cm. Die Genauigkeit hängt von den Prozesszeiten des Skriptes zur Auswertung der Laserinformationen ab. Würden diese Zeiten verringert werden, könnte eine kontinuierlichere Korrektur und somit eine höhere Genauigkeit erreicht werden. Der Einsatz von funktionierenden Encodersensoren würde die Genauigkeit ebenso erhöhen.

- Es sollte eine Kamera in dem Hardwaresystem angeschlossen werden, damit Objekte, die sich nicht in der Arbeitsebene des Lasers befinden, erkannt werden. In dieser Arbeit wurde bereits mit der PMD Nano und der Kinect v1 gearbeitet. Aus Zeitgründen konnte dieser Gesichtspunkt allerdings nicht abschlossen werden. Hierbei sei die Einbindung der Kinect empfohlen, da sie sowohl Bild- als auch Tiefeninformationen liefert. Außerdem gibt es zum Kinectsensor sehr viel Open Source Quellcode, an dem man sich gut orientieren kann.

- Die Umgebungskarte wurde in dieser Arbeit über das Scanmatching erstellt. Allerdings wurde der Roboter manuell durch den Raum geführt und dabei Scans aufgenommen. Dies ist sehr zeitaufwändig und körperlich belastend für den Bediener. Der Prozess der Kartenerstellung sollte automatisiert werden. Dabei könnte der Roboter mit einer Fernbedienung durch den Raum geführt werden und die Kartenerstellung in einem Matlab-Skript automatisiert werden.

- Eine weitere Steigerung des vorangehenden Stichpunktes wäre es, das SLAM-Verfahren (Simultaneous Localization and Mapping) an dem Roboter zu realisieren. Dabei lokalisiert der Roboter sich, während er gleichzeitig seine Umgebungskarte erstellt.

- Die Bedienerfreundlichkeit des Roboters ist im Moment noch schlecht zu beurteilen. Der Anwender muss mehrere Programme ausführen, um die Roboterfahrt zu starten. Außerdem ist die Arbeit mit dem Roboter aus ergonomischer Sicht sehr schlecht zu bewerten. Die Programme laufen auf einem Laptop, der sich auf dem Roboter befindet. Der Anwender muss sich zum Start des Roboters über den Roboter beugen, was zu Belastung des Rückens führt und aus gesundheitlichen Aspekten problematisch für den Anwender ist. Ein Fortschritt wäre eine zusätzliche Bedienereinheit, die aus der Distanz auf den Laptop zugreift. Es könnte sich zum Beispiel um ein Tablet handeln, das eine selbsterklärende bedienerfreundliche App enthält, die alle Programmeinheiten verknüpft und ausführt.

Literaturverzeichnis

[Corke 2013] P. Corke: Robotics, Vision and Control – Fundamental Algorithms in MATLAB, 2. Auflage: 2013, Springer-Verlag

[Bentley 1975] J.L. Bentley: Multidimensional Binary Search Trees Used for Associative Searching, Paper, Stanford University,1975

[Bernshausen 2011] J. Bernshausen: Integration von 3D-„Time-of-Flight“ – Kameras in Applikation zur sicheren Steuerung autonomer Roboter, Doktorarbeit, Universität Siegen, Department Elektrotechnik und Informatik, 2011

[Bräunl 2008] T. Bräunl: Embeded Robotics – Mobile Robot Design and Application with Embedded Systems, 3. Auflage: 2008, Springer –Verlag

[dSPACEHelpDesk] dSpace GmbH: dSPACEHelpDesk – Basisinformationen, Anleitungen, Beispiele und Referenzinformationen zu den Softwareprodukten der Firma dSpace, Revision: Mai 2015

[Faulhaber 2009] Faulhaber: Bediensungsanleitung, 4. Auflage: 2009, Faulhaber

[Hertzberg, Lingemann und Nüchter 2012] J. Hertzberg, K. Lingemann, A. Nüchter: Mobile Roboter – Eine Einführung aus der Sicht der Informatik, 1. Auflage: 2012, Springer-Verlag

[Hoyuko, Kawata 2008] Kawata: Communication Protocol Specification For SCIP2.0 Standart, 4. Auflage: 2008, Hokuyo Automatic Co.,LTD

[HVV Schnellbahnplan] HVV: Schnellbahnplan – URL: http://www.hvv.de/pdf/produktplaene/hvv_produktplan_schnellbahnplan_usar.pdf – Abruf: 2015-06-20

[Koeppen 2014] Prof. Dr.-Ing. Birgit Koeppen: Vorlesungsunterlagen des Moduls Mess-, Steuer- und Regelungstechnik im Department Maschinenbau der HAW Hamburg

[Matlab medfilt1] Matlab: function medfilt(x, n), URL: http://de.mathworks.com/help/signal/ref/medfilt1.html – Abruf: 2015-07-05

[MicroAutoBox 2013] dSpace: Hardware Installation and Configuration, 2013, dSpace GmbH

[MicroAutoBoxOverviewofBoardRevisions_Release2015A] dSpace: MicroAutoBoxOverviewofBoardRevisions_Release2015A.pdf, Mai 2015, dSpace GmbH

[Migration Guide 2013] dSpace: MLIB/MTRACE Migration Guide – dSpace MLIB/MTRACE – dSpace HIL API. NET, März 2013 – URL: http://www.dspace.com/support/patches/TASC/MLIB/discontinuation/MLIB_HIL_API_MigrationGuide/MLIB_HIL_API_MigrationGuide.pdf – Abruf: 2015-08-09

[NYTimes 2015] Emma Cott: Sex Dolls That Talk Back, June 2015, New York Times – URL: http://www.nytimes.com/2015/06/12/technology/robotica-sex-robot-realdoll.html?_r=1 – Abruf: 2015-08-26

[Philippsen 2015] H. Philippsen: Einstig in die Regelungstechnik - Vorgehensmodell für den praktischen Reglerentwurf, 2.Auflage: 2015, Carl Hanser Verlag, München

[RobertvonGoess 2009] RobertvonGoess: Fahrtenregler, März 2009 – URL: http://www.rc-forum.de/member.php?33769-RobertvonGoess&s=85321c5b1b491931ff13160283d8f448 – Abruf: 2015-08-09

[Robot Path Smoothing] Robot Path Smoothing – URL: http://www.comp.dit.ie/jkelleher/rtf/classmaterial/week10/PathSmoothing.pdf – Abruf: 2015-08-07

[Shrestha 2012] S.Shrestha: Hokuyo URG-04LK Lidar Driver for Matlab, Mai 2012 – URL: http://www.mathworks.com/matlabcentral/fileexchange/36700-hokuyo-urg-04lx-lidar-driver-for-matlab – Abruf: 2015-08-27

[Thrun, Burgard und Fox 2006] S. Thrun, W. Burgard, D. Fox: Probalistics Roboti cs, 1. Auflage: 2006, The MIT Press Cambridge, Massachusetts, London, England

[Wiki kd-tree] Wikipedia: k-d tree, Juni 2015 – URL: https://en.wikipedia.org/wiki/K-d_tree – Abruf: 2015-06-15

[Wilm 2013] J.Wilm: Iterative Closest Point, Januar 2013, – URL: http://www.mathworks.com/matlabcentral/fileexchange/27804-iterative-closest-point – Abruf: 2015-08-27

Anhang A Programm

A.1 selbsterstellte Funktionen

alignStart.m

```
function start = alignStart(start_in, mapMatrix)
abfrage = 0;
i = 1;
[z,s] = size(mapMatrix);
x = [ 0 0 1 -1 1 -1 1 -1];
y = [1 -1 0 0 1 1 -1 -1];
while (abfrage == 0)
    if (mapMatrix(start_in(2,1),start_in(1,1)) == 0)
        start = start_in;
        abfrage = 1;
        continue;
    end
    for j = 1:length(x)
        startTemp = start_in + [x(1,j)*(i-1); y(1,j)*(i-1)];
        if ( startTemp(2,1) > 0 && startTemp(2,1) <= z && start-
Temp(1,1) > 0 && startTemp(1,1) <= s)
            if(mapMatrix(startTemp(2,1),startTemp(1,1)) == 0)
                start = startTemp;
                abfrage = 1;
                break;
            end
        end
    end
    i = i+1;
end
```

calcPath.m

```
function [tableDataX, tableDataY, tableTime, thetaBegin, path] =
calcPath(position, res, map2d, dx, pathlength, vlim)
    start = round(position(1:2,1)./res);
    start = alignStart(start, map2d);
    path = dx.path(start);
    smoothedPath = smooth(path,100);
    smoothedPath = smoothedPath.*res;
    [a,~] = size(smoothedPath);
    for i = length(smoothedPath)+1:pathlength
        smoothedPath(i,:) = smoothedPath(a,:);
    end

    tableDataX = smoothedPath(3:end,1)';
    tableDataY = smoothedPath(3:end,2)';
    tableTime = zeros(1,length(tableDataX));

    for i = 2:length(tableDataX)
        dist = sqrt((tableDataX(i)-tableDataX(i-
1))^2+(tableDataY(i)-tableDataY(i-1))^2);
        if (dist == 0)
            f = 0.2;
        else
        f = dist/vlim;
```

```
        end
        tableTime(1,i) = tableTime(1,i-1)+f;
    end
    thetaBegin = atan2((tableDataY(1,2)-
tableDataY(1,1)),(tableDataX(1,2)-tableDataX(1,1)));
            tableTime(1,i) = tableTime(1,i-1)+f;
          end
          thetaBegin = atan2((tableDataY(1,2)-
      tableDataY(1,1)),(tableDataX(1,2)-tableDataX(1,1)));
```

filterObstacle.m

```
function [pointCloudWithoutObstacle, pointCloudObstacle] = fil-
terObstacle(pointCloudNew, pointCloudOld, map)
    pointCloudWithoutObstacle = pointCloudNew;
    if (isempty(pointCloudOld) == false )
        [~,d1] = dsearchn(pointCloudOld', pointCloudNew');
        [~,d2] = dsearchn(map', pointCloudNew');
        delete = find(d1 > 100 & d2 > 300);
        pointCloudObstacle = pointCloudWithoutObstacle(:,delete);
        pointCloudWithoutObstacle(:,delete) = [];
    end
```

getCurrentOrientation.m

```
function currentOrientation = getCurrentOrientation(pointCloud, RT,
TT, currentPosition, positionOfModel)
        p1 = pointCloud(:,1);
        p2 = RT*pointCloud(:,1)+TT;
        p1_s = p1-[positionOfModel(1:2,1);0];
        p2_s = p2-currentPosition;
        phi1 = atan2(p1_s(2,1),p1_s(1,1));
        phi3 = -positionOfModel(3,1)+phi1;
        phi4 = atan2(p2_s(2,1),p2_s(1,1));
        currentOrientation = phi4-phi3;
```

laserdataToPointCloud.m

```
function pointCloud = laserdataToPoint-
Cloud(scan,position,MedFilt,RedFilt)
deg = ones(1,682)*position(3)+[-682/2:1:682/2-1].*240/682*pi/180;
x0 = ones(1,682)*position(1);
y0 = ones(1,682)*position(2);
x = x0 + scan.*cos(deg);
y = y0 + scan.*sin(deg);

pointCloud = [x;y];

del = find((pointCloud(1,:) >= position(1)-50 & pointCloud(1,:) <=
position(1)+50) & (pointCloud(2,:) >= position(2)-50 & point-
Cloud(2,:) <= position(2)+50));
pointCloud(:,del) = [];

if (MedFilt ~= 0)
    pointCloud = medfilt1(pointCloud',MedFilt)';
end

if (RedFilt ~= 0)
    pointCloud = reducFilt(pointCloud,20)';
end
```

mapWithSafetyDist.m

```
function map2 = mapWithSafetyDist(safetyDist, map)
map2 = map;
[z,s] = size(map2);
for i=1:z
    for j=1:s
        if (map(i,j) == 1)
            i1 = i;
            i2 = i;
            j1 = j;
            j2 = j;
            count = 0;
            while ( i1 > 1 && count ~= safetyDist)
                i1 = i1-1;
                count = count + 1;
            end
            count = 0;
            while ( i2 < z && count ~= safetyDist)
                i2 = i2+1;
                count = count + 1;
            end
            count = 0;
            while ( j1 > 1 && count ~= safetyDist)
                j1 = j1-1;
                count = count + 1;
            end
            count = 0;
             while ( j2 < s && count ~= safetyDist)
                j2 = j2+1;
                count = count + 1;
            end
            map2(i1:i2,j1:j2) = 1;
        end
    end
end
```

reducFilt.m

```
function pointCloudRed = reducFilt(pointCloud, r)
    pStart = [pointCloud(1,1);pointCloud(2,1)];
    pointCloudRed = [];
    scanTemp= [pointCloud(1,1),pointCloud(2,1)];
    count = 2;
    count2 = 1;
    for i=2:length(pointCloud(1,:))
        if(pointCloud(1,i) == 0 && pointCloud(2,i) == 0)
            continue
        end
        if (sqrt((pStart(1,1)-pointCloud(1,i))^2+(pStart(2,1)-
pointCloud(2,i))^2) < 2*r)
            scanTemp(count,:) = [pointCloud(1,i),pointCloud(2,i)];
            count = count+1;
        else
            pStart = [pointCloud(1,i);pointCloud(2,i)];
            count = 2;
            if(isempty(scanTemp) == false)
                pointCloudRed(count2,:) =
[sum(scanTemp(:,1))/length(scanTemp(:,1));sum(scanTemp(:,2))/length
(scanTemp(:,1))];
                scanTemp = [];
                count2 = count2 + 1;
            end
```

```
                scanTemp= [pointCloud(1,i),pointCloud(2,i)];
            end
        end
```

smooth.m

```
function smoothedPath = smooth(path, grade)
[z,s] = size(path);
for (i = 1 : grade)
    weightSmooth = 0.5;
    smoothedPath = path;
    for (i = 2 : z-1)
        for (j = 1 : s)
            smoothedPath(i,j) = smoothedPath(i,j) + weightSmooth *
(path(i-1,j) + path(i+1,j) - (2*path(i,j)));
        end
    end
    path = smoothedPath;
end
```

A.2 Initialisierungsskript

```
global res dx map2d map2dWithSafetyDist pointCloudMap2d safetyDist
pathlength vlim grenzScan tableDataX tableDataY Port
%% Benutzereingaben
%1.1: Properties of the robbi in mm: L - length; H - hight; B = width;
B1 = distance
%between the wheels
L = 430;
H = 530;
B = 450;
B1 = 370;

%1.2 speed limit and accleration limit: vlim - speed lim-
it(translation);
%accel - acceleration limit(translation); alim - speed lim-
it(rotation);
%accelw - acceleration limit(rotation); DrehMax - maximum revolution
speed
%of the wheels in [1/min]
vlim = 1*(45/2+0/8*45/2);
accel = vlim/2;
alim =vlim/B1*2;
accelw = alim/2;
DrehMax = 280/14;

%1.3 resolution of the map (example: the size of a cell of the map is
10 cm, so
%the resolution is res = 10)
res = 10;

%1.3 name of the map
nameOfMap = 'karteMatrix';

%1.4 start = [x [mm], y [mm], thetaBegin [rad]], goal=[x [mm], y [mm]]
%thetaEnd = thetaEnd [rad]
start = [500;1000;pi/4];
goal = [2000,4000];
```

```
thetaEnd = 0;

%1.5 Parameters for the PID-Controller
pv = 3;
iv = 0;
dv = 7;
pw = 1;
iw = 0;
dw = 7;

%1.6 Com Port of the Lasersensor
Port = 'COM9';

%%
integrateToolboxes();

%%%%%%%%%%%%%
%%%Mapping%%%
%%%%%%%%%%%%%

%2D-Map
load(nameOfMap);
map2d = eval(genvarname(nameOfMap));

%generate map with safety distance
safetyDist = round(B*1/res/2+20);
map2dWithSafetyDist = mapWithSafetyDist(safetyDist,map2d);

%load Pointcloud for the Scanmatching-Algorithm
load('Karte.mat');
pointCloudMap2d = Karte;

%load size of Robot as Scan
load('grenzScan.mat');

%%%%%%%%%%%%%%%%%%%%
%%% Pathplanning %%%
%%%%%%%%%%%%%%%%%%%%
dx = DXform(map2dWithSafetyDist);
tic
ER = dx.plan(goal./res);
toc
if (ER == 1)
    fprintf('Die Karte sollte nur einen Raum repräsentieren');
else
    pathlength = 5002;
    [tableDataX, tableDataY, tableTime,thetaBegin, path] =
calcPath(start, res, map2dWithSafetyDist, dx, pathlength, vlim);
    actualPose = [start(1:2,1); start(3,1)];
end


%%%%%%%%%%%%%%
%%%Plotting%%%
%%%%%%%%%%%%%%
% figure(1)
% dx.plot(path)
% hold on
% plot(tableDataX,tableDataY,'.b')
% [z,s] = size(map2d);
```

```
% for i=1:z
%     for j=1:s
%         if(map2d(i,j)==1)
%             plot(j,i,'k*')
%         end
%     end
% end
% axis equal
% hold off
```

A.3 Skript zur Laserdatenauswertung

```
global res dx map2d map2dWithSafetyDist pointCloudMap2d safetyDist
pathlength vlim grenzScan tableDataX tableDataY goal Port
integrateToolboxes();
%Initalparameter
firstRound = 1;
simulationHold = 0;
check = 0;
check2 = 0;
[z,s] = size(map2d);
ER2 = 1;
dxDynamic = [];

%required paths in the simulink model
startstopPath = 'Model Root/startstop/Value';
tooClosePath = 'Model Root/PWM-Signal/tooClose/Value';
tableDataXPath = 'Model Root/trajectoryGenerator/x-
Coordinates/LookUpTableData';
tableDataYPath = 'Model Root/trajectoryGenerator/y-
Coordinates/LookUpTableData';
bpXPath = 'Model Root/trajectoryGenerator/x-Coordinates/bp01Data';
bpYPath = 'Model Root/trajectoryGenerator/y-Coordinates/bp01Data';
xIstPath = 'Model Root/Control Circuit/currentModelX';
yIstPath = 'Model Root/Control Circuit/currentModelY';
thetaIstPath = 'Model Root/Control Circuit/currentModelTheta';
actualPosePath = 'Model Root/Control Circuit/actualPose/Value';
resetPath = 'Model Root/Control Cir-
cuit/Integration/setActualPose/Value';
tElapsedPath = 'Model Root/Control Circuit/timeDelay/Value';
simulationPath = 'Model Root/Stop Simulation/In1';
thetaBeginPath = 'Model Root/thetaBeginSubsystem/thetaBegin/Value';
goalPath = 'Model Root/goal/Value';


try
NET.addAssembly('dSPACE.HILAPI.MAPort');
NET.addAssembly('dSPACE.HILAPI.Common');
NET.addAssembly('ASAM.HILAPI.Implementation');
NET.addAssembly('ASAM.HILAPI.Interfaces');
import ASAM.HILAPI.dSPACE.MAPort.*;
import ASAM.HILAPI.Implementation.Common.ValueContainer.*;
catch e
    error(e.message)
end

try
lidar = SetupLidar(Port);
currentFolder = pwd;
```

```
simulationmodelPath =
sprintf('%s\\simulationmodel.sdf',currentFolder);
config-
Dict=NET.createGeneric('System.Collections.Generic.Dictionary',{'Syste
m.String', 'System.String'});
configDict.Add('ApplicationPath',simulationmodelPath);
configDict.Add('PlatformIdentifier','DS1401');
simulationModelMAPort = MAPort(configDict);

goal1 = [simula-
tionModelMAPort.Read(goalPath).Value.Item(0),simulationModelMAPort.Rea
d(goalPath).Value.Item(1)];
simulationModelMAPort.Write(resetPath,FloatValue(1.0));
simulationModelMAPort.Write(resetPath,FloatValue(0.0));

%Calculates a new potential field, if the User defined a new goal in
the Control Desk Platform
if (sum(goal ~= goal1)>0)
    goal = goal1;
    dx = DXform(map2dWithSafetyDist);
    ER = dx.plan(goal1./res);
end

while (simulationModelMAPort.Read(simulationPath).Value == 0)
    laserScan = LidarScan(lidar);
    tStart = tic;

    %filtering unsucceded Values (they are zero or less than 10 mm)
    measurementSucceed = find(laserScan > 10);

    %is the Robot too close to an Object?
    diff = laserScan(1,measurementSucceed)-grenzScan(1, measurement-
Succeed);
    tooClose = find (diff < 50);

    if (length(tooClose) > 5)
        if (simulationHold == 0 && simula-
tionModelMAPort.Read(startstopPath).Value == 1)
            fprintf('tooClose\n')
            simulationHold = 1;
            simula-
tionModelMAPort.Write(startstopPath,FloatValue(0.0));
            simulationModelMAPort.Write(tooClosePath,FloatValue(1.0));
            continue
        else
            continue
        end
    else
        if (simulationHold == 1)
            positionOfModel = [simula-
tionModelMAPort.Read(xIstPath).Value;simulationModelMAPort.Read(yIstPa
th).Value;simulationModelMAPort.Read(thetaIstPath).Value];
            if isempty(dxDynamic)
                [tableDataX, tableDataY, tableTime, thetaBegin, path]
= calcPath(positionOfModel, res, map2dWithSafetyDist, dx, pathlength,
vlim);
                fprintf('It has been calculated a new Path after Stop-
ping because of an Obstacle\n');
                %%%%%%%%%%%%%
                %%%Plotting%%%
                %%%%%%%%%%%%%
```

```
                h = figure(2);
                dx.plot(path)
            else
                [tableDataX, tableDataY, tableTime, thetaBegin, path]
= calcPath(positionOfModel, res, map2dDynamicWithSafetyDist, dxDynam-
ic, pathlength, vlim);
                %%%%%%%%%%%%%%
                %%%Plotting%%%
                %%%%%%%%%%%%%%
                fprintf('It has been calculated a new Path after Stop-
ping because of an Obstacle\n');
                figure(2)
                dxDynamic.plot(path)
            end
            hold on
            plot(tableDataX./res,tableDataY./res,'.b')
            hold off
            simulationModelMAPort.Write(thetaBeginPath, Float-
Value(thetaBegin));
            simulationModelMAPort.Write(tableDataXPath, FloatVector-
Value(tableDataX));
            simulationModelMAPort.Write(tableDataYPath, FloatVector-
Value(tableDataY));
            simulationModelMAPort.Write(bpXPath, FloatVectorVal-
ue(tableTime));
            simulationModelMAPort.Write(bpYPath, FloatVectorVal-
ue(tableTime));
            simula-
tionModelMAPort.Write(startstopPath,FloatValue(1.0));
            simulationModelMAPort.Write(tooClosePath,FloatValue(0.0));
            simulationHold = 0;
        end
    end

    %ScanMatching
    positionOfModel = [simula-
tionModelMAPort.Read(xIstPath).Value;simulationModelMAPort.Read(yIstPa
th).Value;simulationModelMAPort.Read(thetaIstPath).Value];
    laserPointCloud = laserdataToPointCloud(laserScan, positionOfMod-
el, 5, 20);

    if (firstRound ==1)
        laserPointCloudOld = laserPointCloud;
        filteredLaserPointCloud = laserPointCloud;
    else
        [filteredLaserPointCloud, pointCloudObstacle] = filterObsta-
cle(laserPointCloud, laserPointCloudOld, pointCloudMap2d(1:2,:));
        laserPointCloudOld = filteredLaserPointCloud;
    end

    filteredLaserPointCloud(3,:) = ze-
ros(1,length(filteredLaserPointCloud));
    [RT,TT, ER] = icp2(pointCloudMap2d, filteredLaserPointCloud,30);
    fprintf('Scanmatching done with Error of: %g\n',ER(end));

    %%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%Plotting Scanmatching%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%
    currentPosition = RT*[positionOfModel(1:2,1);0] + TT;
    TR =
[TT(1)*ones(1,length(filteredLaserPointCloud));TT(2)*ones(1,length(fil
teredLaserPointCloud));TT(3)*ones(1,length(filteredLaserPointCloud))];
```

```
    filteredLaserPointCloudTrans = RT*filteredLaserPointCloud+TR;
    g = figure(1);
    hold on
    plot(pointCloudMap2d(1,:),pointCloudMap2d(2,:),'*r');

plot(filteredLaserPointCloud(1,:),filteredLaserPointCloud(2,:),'*b');

plot(filteredLaserPointCloudTrans(1,:),filteredLaserPointCloudTrans(2,
:),'*k');
    plot(currentPosition(1,1),currentPosition(2,1),'xk')
    axis equal
    hold off

    %if scanmatching was ok, the current position of the Robot will be
    %calculated. If there is an Object on the planned Path of the ro-
bot a
    %new Path will be calculated

    if(ER(end) > 35)
        continue;
    else
        currentPosition = RT*[positionOfModel(1:2,1);0] + TT;
        currentOrientation = getCurrentOrienta-
tion(filteredLaserPointCloud, RT, TT, currentPosition, positionOfMod-
el);
        actualPose = [currentPosition(1:2,1); currentOrientation];
        % In the Beginning of the drinving Robot a path from the actu-
al
        % Pose has to be determined
        if (firstRound == 1)
            firstRound = 0;
            [tableDataX, tableDataY, tableTime, thetaBegin, path] =
calcPath(currentPosition(1:2,1), res, map2dWithSafetyDist, dx, path-
length, vlim);
            fprintf('FIRST path has been calculated\n')
            %%%%%%%%%%%%%%
            %%%Plotting%%%
            %%%%%%%%%%%%%%
            h = figure(2);
            dx.plot(path)
            hold on
            plot(tableDataX./res,tableDataY./res,'.b')
            hold off

            %transfer parameter to the simulink model
            simulationModelMAPort.Write(thetaBeginPath, Float-
Value(thetaBegin));
            simulationModelMAPort.Write(actualPosePath, FloatVector-
Value(actualPose));
            simulationModelMAPort.Write(resetPath,FloatValue(1.0));
            simulationModelMAPort.Write(resetPath,FloatValue(0.0));
            simulationModelMAPort.Write(tableDataXPath, FloatVector-
Value(tableDataX));
            simulationModelMAPort.Write(tableDataYPath, FloatVector-
Value(tableDataY));
            simulationModelMAPort.Write(bpXPath, FloatVectorVal-
ue(tableTime));
            simulationModelMAPort.Write(bpYPath, FloatVectorVal-
ue(tableTime));
            if (simulationModelMAPort.Read(startstopPath).Value == 1)
                simula-
tionModelMAPort.Write(startstopPath,FloatValue(0.0));
```

```
                simula-
tionModelMAPort.Write(startstopPath,FloatValue(1.0));
            end
        else
            %in Case of Obstacle
            if (length(pointCloudObstacle) > 2)
                %It has to be checked if the Obstacle is on the
planned
                %Path
                lengthLaserPointCloud = length(laserPointCloud);
                laserPointCloud(3,:) = zeros(1,lengthLaserPointCloud);
                TR =
[TT(1)*ones(1,lengthLaserPointCloud);TT(2)*ones(1,lengthLaserPointClou
d);TT(3)*ones(1,lengthLaserPointCloud)];
                laserPointCloudTrans = RT*laserPointCloud(1:3,:)+TR;
                [k0, d0] =
dsearchn(pointCloudMap2d(1:2,:)',laserPointCloudTrans(1:2,:)');
                pointCloudObstacleTrans = laserPointCloud-
Trans(:,find(d0>100));
                [k3,d3] = dsearchn([tableDataX ; ta-
bleDataY]',positionOfModel(1:2,1)');
                startCalc = round(safetyDist*2)+k3;
                [k1,d1] =
dsearchn([tableDataX(1,startCalc:end);tableDataY(1,startCalc:end)]',po
intCloudObstacleTrans(1:2,:)');
                dist = [ta-
bleDataX(1,startCalc+k1);tableDataY(1,startCalc+k1)]-
pointCloudObstacleTrans(1:2,:);
                pointOnPath = find(abs(dist(1,:)/res) < (safetyDist-
10) & abs(dist(2,:)./res) < (safetyDist-10));
                pointCloudObstacleTrans = pointCloudObsta-
cleTrans(:,pointOnPath);
                ER2 = 1;
                %if the pointCloudObstacleTrans is not empty, there is
an
                %obstacle on the path
                if (isempty(pointCloudObstacleTrans) == false)
                    %It has to be checked if the Obstacle is on the
goal,
                    %if yes the robot should just go on like before
                    [k2,d2] =
dsearchn(pointCloudObstacleTrans(1:2,:)',goal);
                    if(floor(d2/res) > safetyDist*sqrt(2))
                        check2 = 1;
                        if (simula-
tionModelMAPort.Read(startstopPath).Value == 1)
                            simula-
tionModelMAPort.Write(startstopPath,FloatValue(0.0));
                            check = 1;
                        end
                        %a new map map2dDynamic will be created
                        [a, b] = size(pointCloudObstacleTrans);
                        map2dDynamic = map2d;
                        for i = 1:b
                            temp1 =
round(pointCloudObstacleTrans(1,i)/res);
                            temp2 =
round(pointCloudObstacleTrans(2,i)/res);
                            if (temp1 > 0 && temp1 <= s && temp2 > 0
&&temp2 <=z)
                                if (map2dDynamic(temp2, temp1) == 0)
                                    map2dDynamic(temp2,temp1) = 1;
                                end
```

```
                            end
                        end
                        %A new potential will be created. In case
there is
                        %no way to fullfill the map with Potential (in
Case
                        %the Room is divided in two), ER2 will be 1.
                        map2dDynamicWithSafetyDist = mapWithSafe-
tyDist(safetyDist, map2dDynamic);
                        dxDynamic =
DXform(map2dDynamicWithSafetyDist);
                        ER2 = dxDynamic.plan(goal./res);
                        fprintf('ER2:%g',ER2);

                        %Another Scan will be made because the Calcu-
lation
                        %of the new Potential takes up to 15 sec. The
                        %Position need to be updated
                        laserScan = LidarScan(lidar);
                        positionOfModel = [simula-
tionModelMAPort.Read(xIstPath).Value;simulationModelMAPort.Read(yIstPa
th).Value;simulationModelMAPort.Read(thetaIstPath).Value];
                        laserPointCloud = laserdataToPoint-
Cloud(laserScan, positionOfModel, 5, 20);
                        [filteredLaserPointCloud, pointCloudObstacle]
= filterObstacle(laserPointCloud, laserPointCloudOld, pointCloud-
Map2d(1:2,:));
                        laserPointCloudOld = filteredLaserPointCloud;
                        filteredLaserPointCloud(3,:) = ze-
ros(1,length(filteredLaserPointCloud));
                        [RT,TT, ER] = icp2(pointCloudMap2d, fil-
teredLaserPointCloud,30);
                        currentPosition =
RT*[positionOfModel(1:2,1);0] + TT;
                        currenOrientation = getCurrentOrienta-
tion(filteredLaserPointCloud, RT, TT, currentPosition, positionOfMod-
el);
                        actualPose = [currentPosition(1:2,1); curren-
tOrientation];
                        simulationModelMAPort.Write(actualPosePath,
FloatVectorValue(actualPose));
                        simula-
tionModelMAPort.Write(resetPath,FloatValue(1.0));
                        simula-
tionModelMAPort.Write(resetPath,FloatValue(0.0));
                        %If there was no Error the Path around the Ob-
stacle
                        %can be calculated, else the robot just will
go on
                        %as before
                        if (ER2 ==0)
                            [tableDataX, tableDataY, tableTime, the-
taBegin, path] = calcPath(currentPosition(1:2,1), res,
map2dDynamicWithSafetyDist, dxDynamic, pathlength, vlim);
                            fprintf('Path around OBSTACLE has been
calculated\n')
                        else
                            [tableDataX, tableDataY, tableTime, path]
= calcPath(currentPosition(1:2,1), res, map2dWithSafetyDist, dx, path-
length, vlim);
                            fprintf('PATH has been calculated\n')
                        end
```

```
                        %%%%%%%%%%%%%
                        %%%Plotting%%%
                        %%%%%%%%%%%%%
                        h = figure(2);
                        dxDynamic.plot(path)
                        hold on
                        plot(tableDataX./res,tableDataY./res,'.b')
                        hold off

                        simulationModelMAPort.Write(thetaBeginPath,
FloatValue(thetaBegin));
                        simulationModelMAPort.Write(tableDataXPath,
FloatVectorValue(tableDataX));
                        simulationModelMAPort.Write(tableDataYPath,
FloatVectorValue(tableDataY));
                        simulationModelMAPort.Write(bpXPath, FloatVec-
torValue(tableTime));
                        simulationModelMAPort.Write(bpYPath, FloatVec-
torValue(tableTime));
                        simula-
tionModelMAPort.Write(tElapsedPath,FloatValue(0));
                        if (check == 1)
                            simula-
tionModelMAPort.Write(startstopPath,FloatValue(1.0));
                            check = 0;
                        end
                    end
                end
            end
            %if no new Path has been calculated just the new Position
will
            %be updated
            if (check2 ==0)
                simulationModelMAPort.Write(actualPosePath, FloatVec-
torValue(actualPose));
                fprintf('actualPose\n');
                tElapsed = toc(tStart);
                simula-
tionModelMAPort.Write(tElapsedPath,FloatValue(tElapsed));
            else
                check2 = 0;
            end
        end
    end
    fprintf('Zeit:%g',toc(tStart))
end
simulationModelMAPort.Dispose();
DisconnectLidar(lidar);

catch e
    if(exist('lidar','var'))
        DisconnectLidar(lidar);
    end
    if (exist('simulationModelMAPort', 'var'))
        simulationModelMAPort.Dispose();
    end

    if (isa(e, 'NET.NetException'))
        if (isa(e.ExceptionObject,
'ASAM.HILAPI.Interfaces.Common.Error.IHILAPIException'))
            disp('HIL API exception in');
            disp(e.stack);
```

```
            fprintf('Code: %d\n', e.ExceptionObject.Code);
            fprintf('CodeDescription: %s\n',
char(e.ExceptionObject.CodeDescription));
            fprintf('VendorCode: %d\n', e.ExceptionObject.VendorCode);
            fprintf('VendorCodeDescription: %s\n',
char(e.ExceptionObject.VendorCodeDescription));
        else
            disp('.NET exception in');
            disp(e.stack);
            fprintf('Message: %s', char(e.message));
        end
    else
        rethrow(e);
    end
end

clear platformHandler;
```

A.4 Simulinkprogramm

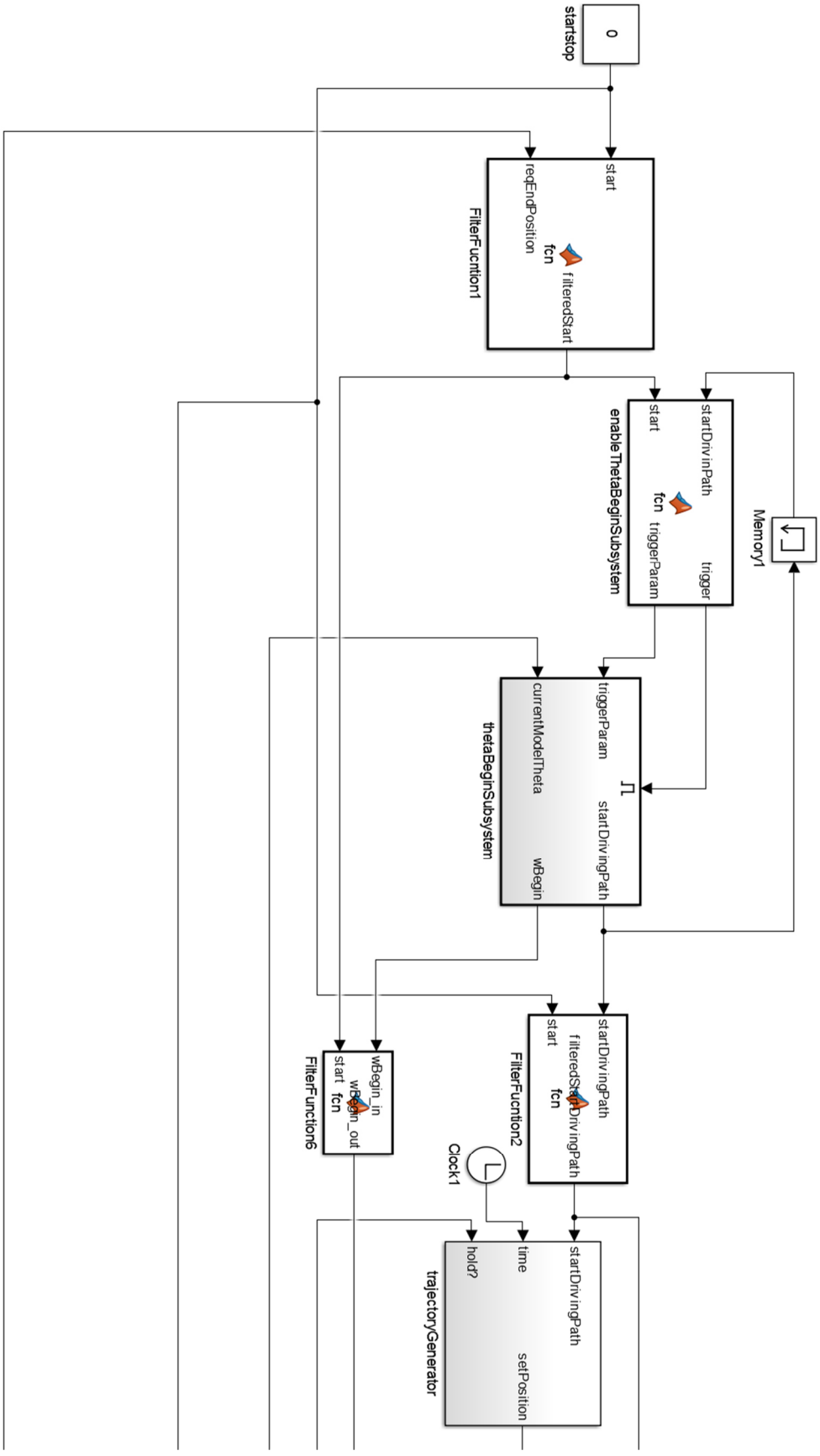

Abbildung 0-1: oberste Ebene des Simulinkmodells: erste Hälfte

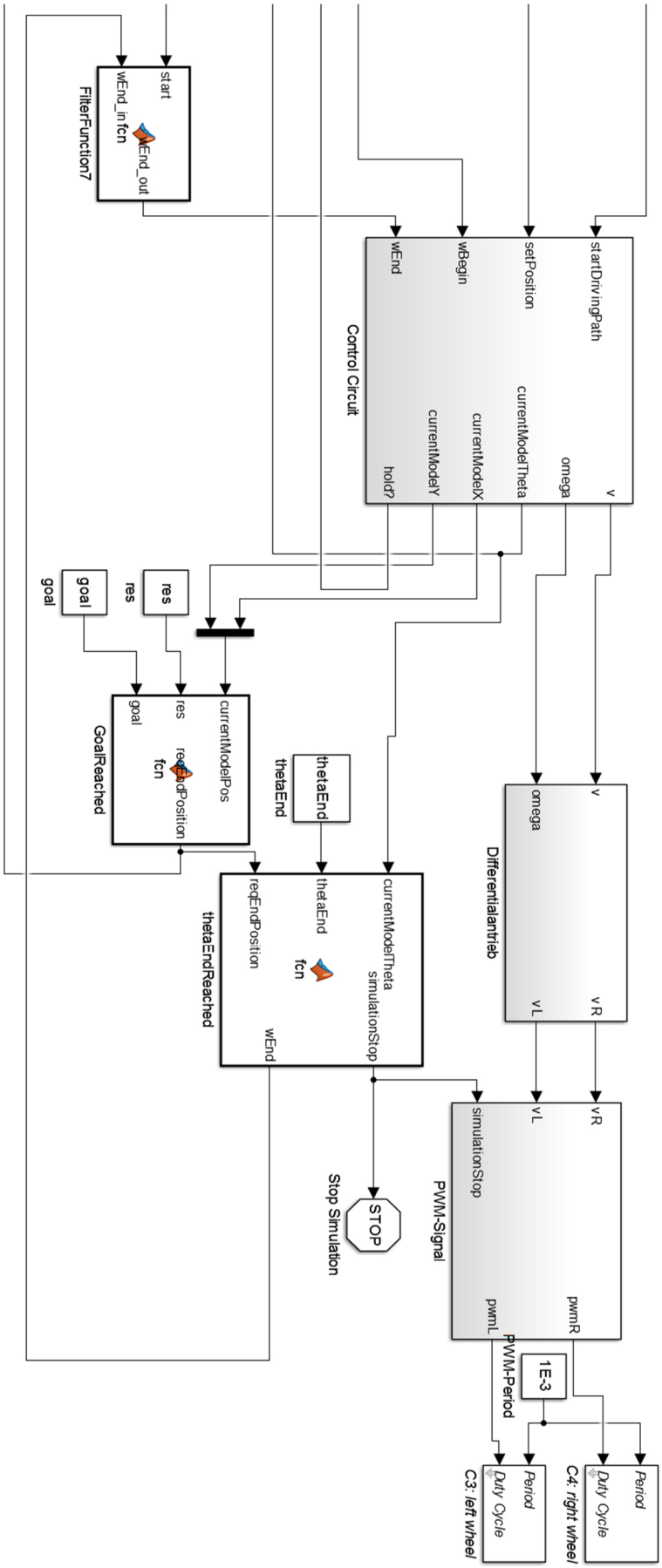

Abbildung 0-2: oberste Ebene des Simulinkmodells: zweite Hälfte

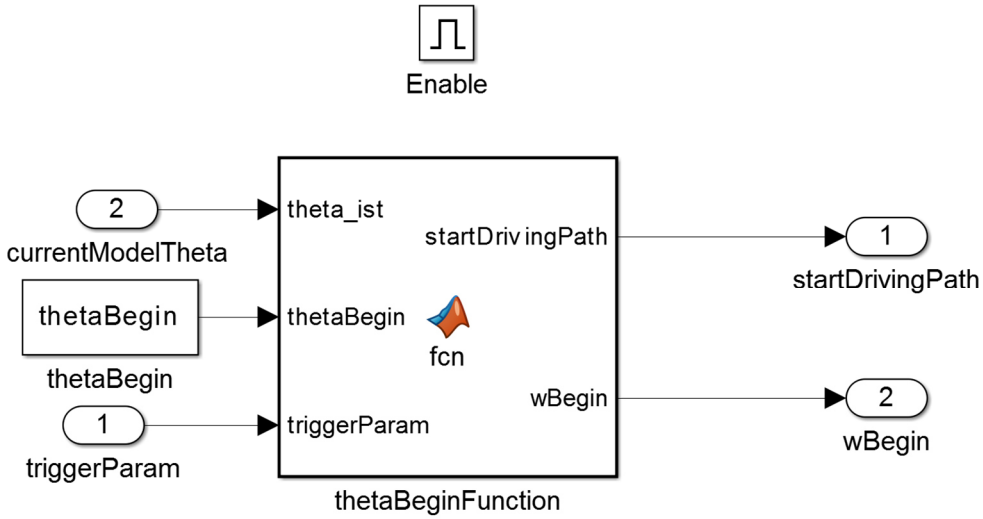

Abbildung 0-3: thetaBeginSubsystem

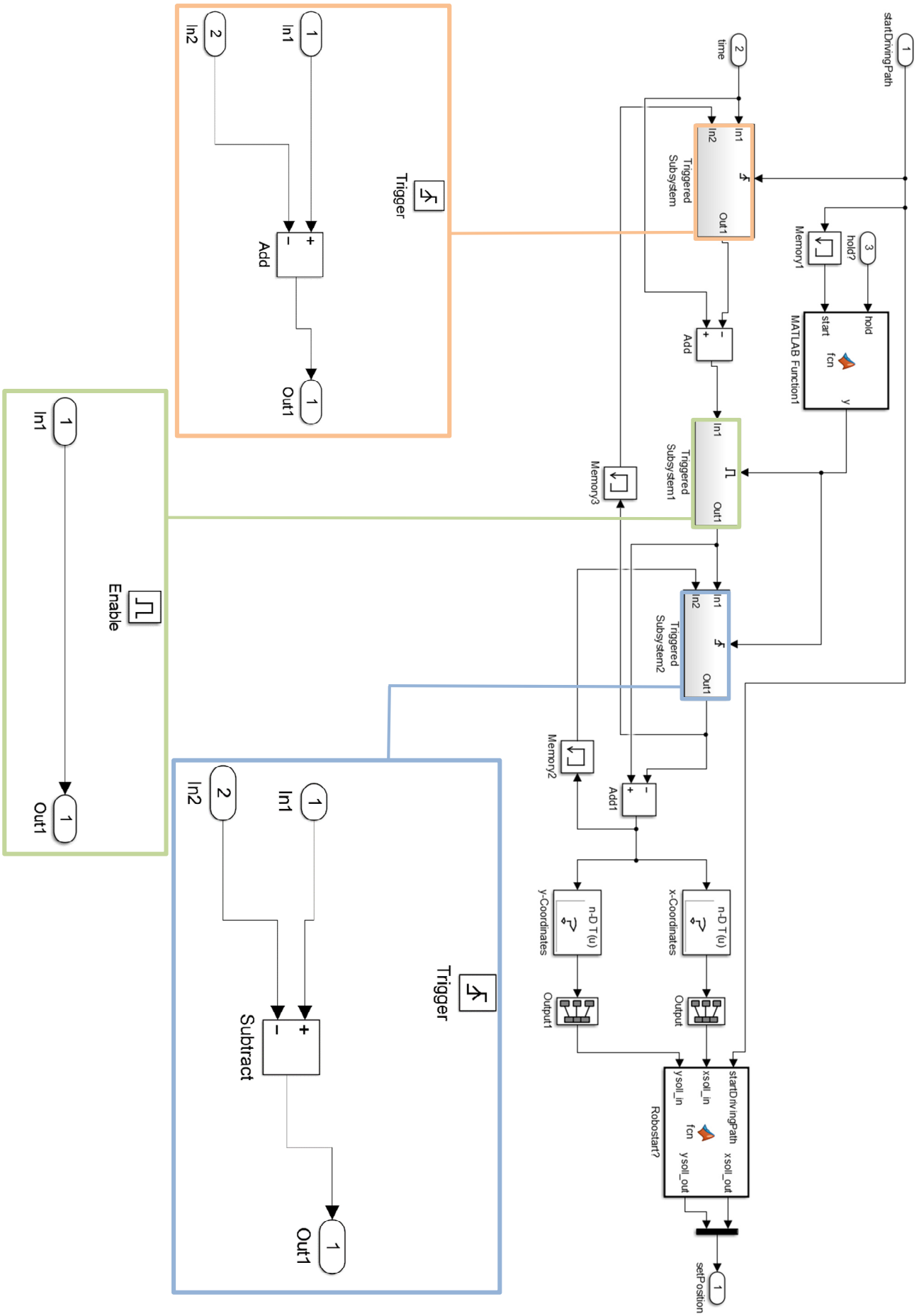

Abbildung 0-4: trajectoryGenerator

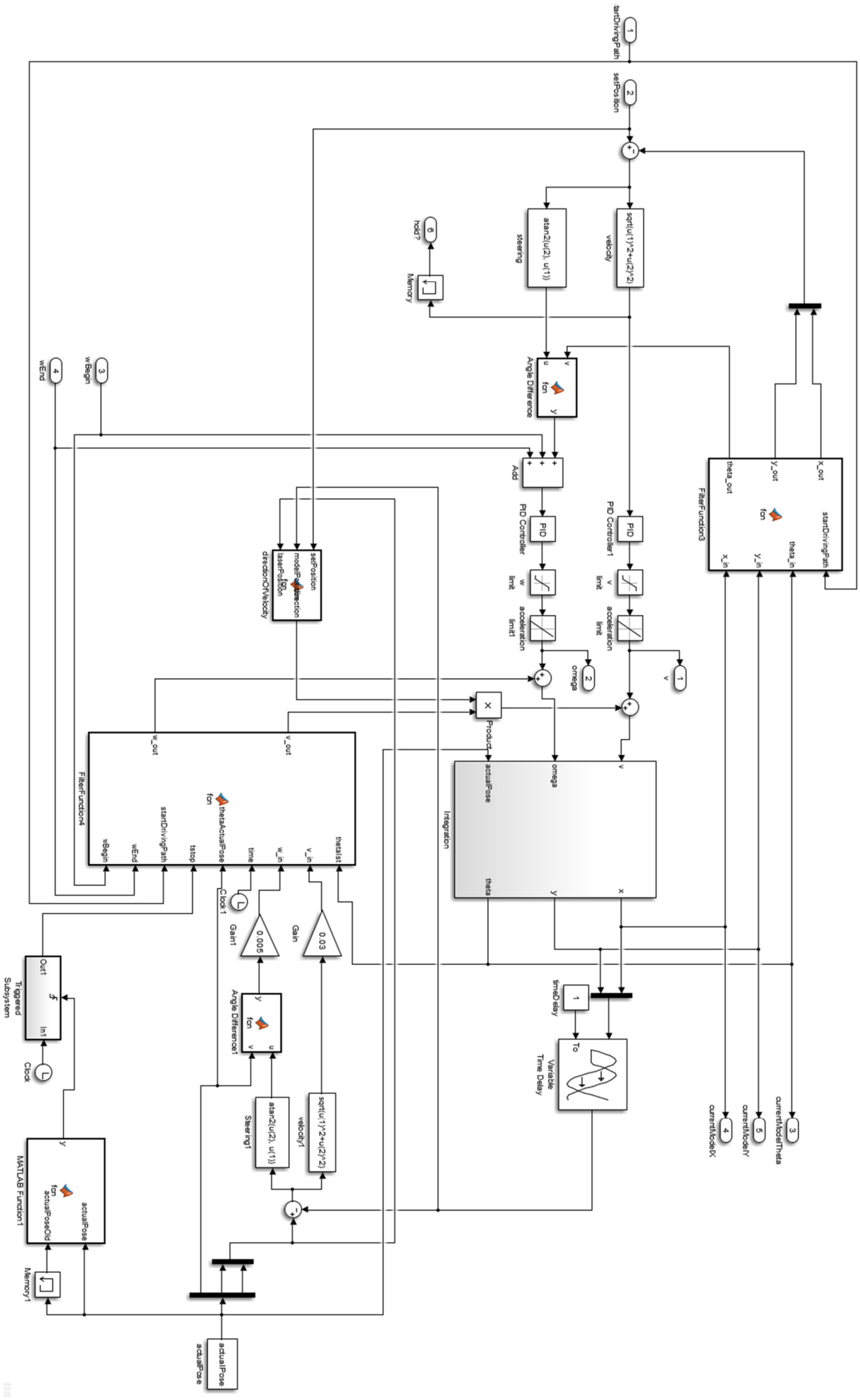

Abbildung 0-5: Control Circuit

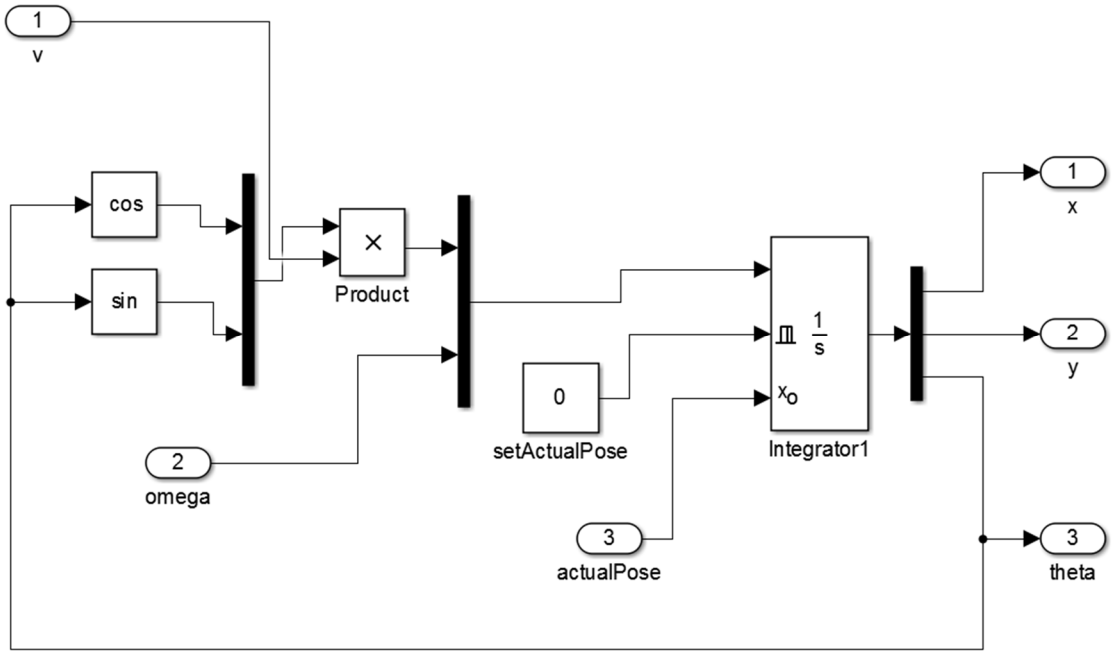

Abbildung 0-6: Control Circuit/Integrator

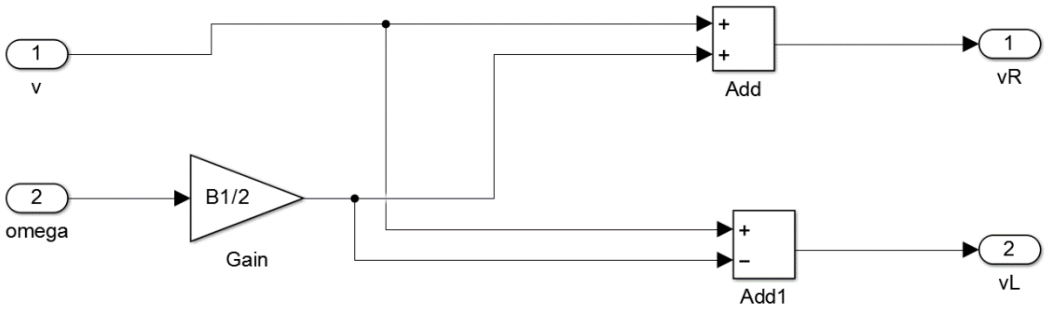

Abbildung 0-7: Differentialantrieb

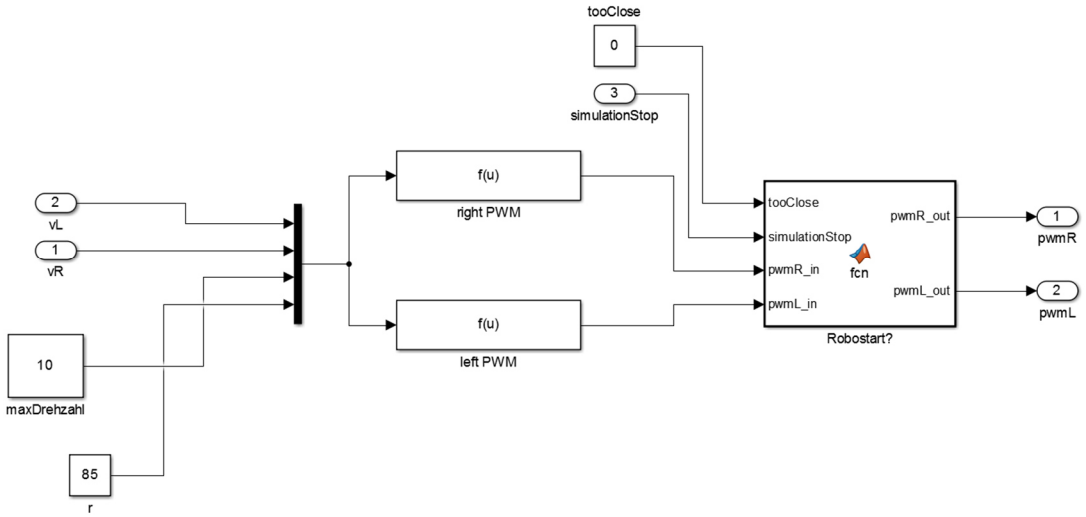

Abbildung 0-8: PWM-Signal

A.4.a Inhalte der Matlab-Functions

Control Circuit/Angle Difference

```
function y = fcn(v,u)
y = angdiff(u,v);
```

Control Circuit/Angle Difference1

```
function y = fcn(v,u)
y = angdiff(u,v);
```

Control Circuit/ directionOfVelocity

```
function direction = fcn(setPosition,modelPosition,laserPosition)
    if (sqrt(sum((setPosition-
    modelPosition).^2))>sqrt(sum((setPosition-laserPosition)).^2))
        direction = 1;
    else
        direction = -1;
    end
```

Control Circuit/FilterFunction3

```
function [x_out, y_out, theta_out] =
fcn(startDrivingPath,theta_in,y_in,x_in)
    if(startDrivingPath == 0)
        x_out = 0;
        y_out = 0;
        theta_out = 0;
    else
        x_out = x_in;
        y_out = y_in;
        theta_out = theta_in;
    end
```

Control Circuit/FilterFunction4

```
function [v_out, w_out] = fcn(thetaIst, v_in, w_in, time, thetaAc-
tualPose, tstop, startDrivingPath, wEnd, wBegin)
    m = 0.5;
    tspann = 5;
    if (startDrivingPath == 0 && wBegin == 0 && wEnd == 0)
        v_out = 0;
        w_out = 0;
    elseif (time-tstop)>tspann && startDrivingPath == 1
        w_out = 0;
        v_out = v_in;
    elseif ((wBegin ~= 0 || wEnd ~=0) && startDrivingPath == 0 &&
(time-tstop)<=tspann)
        w_out = angdiff(thetaActualPose, thetaIst)*m;
        v_out = 0;
    elseif ((wBegin ~= 0 || wEnd ~=0) && startDrivingPath == 0 &&
(time-tstop)>tspann)
        w_out = 0;
        v_out = 0;
    else
        v_out = v_in;
        w_out = w_in;
    end
```

Control Circuit/MATLAB Function1

```
function y = fcn(actualPose, actualPoseOld)
if (actualPose(1,1) == actualPoseOld(1,1))
```

```
    y = 0;
else
    y = 1;
end
```

enableThetaBeginSubsystem

```
function [trigger, triggerParam] = fcn(startDrivinPath, start)
if (start == 1 && startDrivinPath == 0)
    trigger = start;
    triggerParam = 2;
elseif (start == 0 && startDrivinPath == 1)
    trigger = 1;
    triggerParam = 3;
else
    trigger = 0;
    triggerParam = 0;
end
```

FilterFunction1

```
function filteredStart = fcn(start, reqEndPosition)
if (reqEndPosition == 1)
    filteredStart = 0;
else
    filteredStart = start;
end
```

FilterFunction2

```
function filteredStartDrivingPath = fcn(startDrivingPath, start)
if(startDrivingPath ==1 && start ==1)
    filteredStartDrivingPath = 1;
else
    filteredStartDrivingPath = 0;
end
```

FilterFunction6

```
function wBegin_out = fcn(wBegin_in, start)
if (start == 0)
    wBegin_out = 0;
else
    wBegin_out = wBegin_in;
end
```

FilterFunction7

```
function wEnd_out = fcn(start, wEnd_in)
if(start == 0)
    wEnd_out = 0;
else
    wEnd_out = wEnd_in;
end
```

GoalReached

```
function reqEndPosition  = fcn(currentModelPos,res, goal)
if(sqrt((res*goal(1,1)-currentModelPos(1,1))^2+(res*goal(2,1)-
currentModelPos(2,1))^2)<2*res)
    reqEndPosition = 1;
else
    reqEndPosition = 0;
end
```

PWM-Signal/Robostart?

```
function [pwmR_out, pwmL_out] = fcn(tooClose, simulationStop,
pwmR_in, pwmL_in)
if (simulationStop == 1 || tooClose == 1)
    pwmR_out = 0.5;
    pwmL_out = 0.5;
else
    pwmR_out = pwmR_in;
    pwmL_out = pwmL_in;
end
```

thetaBeginSubsystem/thetaBeginFunction

```
function [startDrivingPath, wBegin] = fcn(theta_ist, thetaBegin,
triggerParam)
if (triggerParam == 2)
    w = angdiff(thetaBegin, theta_ist);
        if w > 0 && abs(w) > 5*pi/180
            wBegin = 0.04;
            startDrivingPath = 0;
        elseif w < 0 && abs(w) > 5*pi/180
            wBegin = -0.04;
            startDrivingPath = 0;
        else
            wBegin = 0;
            startDrivingPath = 1;
        end
else
    wBegin = 0;
    startDrivingPath =0;
end
```

thetEndReached

```
function [simulationStop, wEnd]  = fcn(currentModelTheta, thetaEnd,
reqEndPosition)
    w = angdiff(thetaEnd, currentModelTheta);
    if (reqEndPosition == 1)
        if abs(w) > pi/180 && w > 0
            wEnd = 0.05;
            simulationStop = 0;
        elseif abs(w) > pi/180 && w < 0
            wEnd = -0.05;
            simulationStop = 0 ;
        else
            wEnd = 0;
            simulationStop = 1;
        end
    else
        simulationStop = 0;
        wEnd = 0;
    end
```

trajectoryGenerator/Matlab Function1

```
function y = fcn(hold, start)
if (hold == 0 || hold < 50 || start == 0)
    y = 1;
else
    y = 0;
end
```

trajectoryGenerator/Robostart?

```
function [xsoll_out, ysoll_out] = fcn(startDrivingPath, xsoll_in,
ysoll_in)
    if(startDrivingPath == 0)
        xsoll_out = 0;
        ysoll_out = 0;
    else
         xsoll_out =  xsoll_in;
         ysoll_out = ysoll_in;
    end
```

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Dieses Blatt, mit der folgenden Erklärung, ist nach Fertigstellung der Abschlussarbeit durch den Studierenden auszufüllen und jeweils mit Originalunterschrift als letztes Blatt in das Prüfungsexemplar der Abschlussarbeit einzubinden.
Eine unrichtig abgegebene Erklärung kann -auch nachträglich- zur Ungültigkeit des Studienabschlusses führen.

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: Güldenring

Vorname: Ronja

dass ich die vorliegende Bachelorarbeit bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Bahnplanung, Selbstlokalisierung und Hindernisumfahrung eines mobilen Roboters

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

- die folgende Aussage ist bei Gruppenarbeiten auszufüllen und entfällt bei Einzelarbeiten -

Die Kennzeichnung der von mir erstellten und verantworteten Teile der -bitte auswählen- ist erfolgt durch:

Hamburg	25.09.2015	
Ort	Datum	Unterschrift im Original