



# Masterarbeit

## **Entwicklung einer Bewegungssteuerung für ein fahrerloses Transportsystem**

Vorgelegt von:

Annika Gholum

Studiengang: Prozessmanagement Usability Engineering Industrie 4.0

Matrikelnummer: 1678060

Abgabetermin:

06.09.2023

Erstprüfer:

Prof. Dr. Jens Hofschulte

Zweitprüfer

Prof. Dr. Franz Kallage

## **Aufgabenstellung**

Hochschule Hannover

Masterarbeit

### **Entwicklung einer Bewegungssteuerung für ein fahrerloses Transportsystem**

Annika Gholam  
Matr. Nr 1678060

In der Produktion kommen zunehmend fahrerlose Transportsysteme zum Einsatz, um die produzierten Güter zwischen den einzelnen Fertigungsstellen zu transportieren. Diese Transportsysteme müssen dazu sehr manövrierfähig sein, um flexibel auch bei engen Platzverhältnissen agieren zu können. Fahrzeuge mit Mecanum-Räder sind hierfür besonders gut geeignete, da sie omnidirektionale Bewegungen ermöglichen. Sie erfordern jedoch einen höheren Steuerungsaufwand.

Ziel dieser Masterarbeit ist die Entwicklung einer Bewegungssteuerung für ein fahrerloses Transportsystem auf der Basis einer Mecanum-Plattform. Hierzu ist zunächst eine Einzelachsregelung der Mecanum-Räder und Ansteuerung der Kinematik auf dem Microcontroller des Fahrzeugs zu implementieren. Im Weiteren soll dann eine Trajektorienplanung für eine Punkt-zu-Punkt Bewegung auf einem externen Rechner implementiert werden, die den Microcontroller geeignet ansteuert.

Erstprüfer: Prof. Dr.-Ing. Jens Hofschulte

Zweitprüfer: Prof. Dr.-Ing. Franz Kallage

**Eidesstattliche Erklärung**

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit mit dem Thema „Entwicklung einer Bewegungssteuerung für ein fahrerloses Transportsystem“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht wurden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Hameln, den 04.09.2023

---

ANNIKA GHOLAM

## Inhaltsverzeichnis

Aufgabenstellung .....	I
Eidesstattliche Erklärung .....	II
Abbildungsverzeichnis .....	V
Tabellenverzeichnis .....	VII
Programmcodeverzeichnis .....	VIII
Abkürzungen .....	IX
Formeln .....	X
1 Einleitung .....	1
2 Vorgehensweise zur Entwicklung der Bewegungssteuerung .....	2
3 Ausgangssituation .....	4
3.1 Arduino .....	4
3.2 Fahrerloses Transport Fahrzeug mit Mecanum-Wheels .....	6
3.2.1 Aufbau und Kinematik der Mecanum Wheels .....	7
3.2.2 Servomotoren, PWM-Signal und Timer zum Einstellen des PWM-Signals .....	9
3.2.3 Ultraschallsensoren <i>Dual UltraSonic</i> und Bus-System RS485 .....	11
3.2.4 Arduino-Board .....	12
3.2.5 Bibliotheken und Demoskript .....	15
3.3 Trackingsystem OptiTrack .....	23
4 Product Backlog für die Bewegungssteuerung .....	24
5 Schnittstellen zur Datenübertragung .....	26
5.1 Sprint Backlog #1 .....	26
5.2 Schnittstellen der vorhandenen Systeme .....	27
5.2.1 Rest API .....	27
5.2.2 Datenaustausch über Bluetooth .....	29
5.3 Implementierung der Hardware und der Softwareschnittstellen .....	30
5.3.1 Installation und Kalibrierung der Trackingkamera .....	30
5.3.2 Implementieren des Bluetooth-Moduls HC05 .....	34
5.3.3 Programmcode zum Datenaustausch .....	37
6 Bewegungssteuerung entlang einer Punkt-zu-Punkt-Verbindung .....	40
6.1 Sprint Backlog #2 .....	40
6.2 Grundlagen zu Kaskadenregelung, Trajektorienplanung und Transformation von Koordinatensystemen .....	41
6.2.1 Kaskadenregelung .....	41
6.2.2 Bahn- und Trajektorienplanung einer Punkt-zu-Punkt-Bewegung .....	42
6.2.3 Transformation zwischen den Koordinatensystemen .....	46
6.3 Programmcodes Sprint #2 .....	49
6.3.1 Programmcode für Fahrbewegung in <i>Arduino</i> .....	49

6.3.2	Programmcode für Lageregelung.....	51
6.3.3	Programmcode für Trajektorienplanung in Python .....	55
7	Autonome Pfadplanung .....	61
7.1	Sprint Backlog #3.....	61
7.2	Pfadplanungsalgorithmen .....	62
7.3	Implementieren des Pfadplanungsalgorithmus .....	68
7.4	Generierung einer Fahrbewegung entlang eines Pfades.....	76
8	Schlussbetrachtung .....	80
8.1	Zusammenfassung .....	80
8.2	Kritische Betrachtung .....	82
8.3	Ausblick .....	84
9	Literaturverzeichnis .....	85
10	Anhang .....	A
	Anhang A: Datenblatt FTF .....	B
	Anhang B: Demoskript.....	D
	Anhang C: Arduino-Skript für Fahrbewegung Sprint #1 .....	H
	Anhang D: Arduino-Skript für Fahrbewegung Sprint#2 .....	K
	Anhang E: Skript Lageregelung.....	M
	Anhang F: Skript Bahnplanung für Punkt-zu-Punkt-Bewegung .....	O
	Anhang G: Skript Pfadplanung und Fahrbewegung entlang Pfad .....	U

**Abbildungsverzeichnis**

Abbildung 2-1: Prozessablauf <i>Scrum</i> .....	2
Abbildung 3-1: Leeres Arduino-Sketch.....	5
Abbildung 3-2: Der Compiler als Dolmetscher .....	5
Abbildung 3-3: Das <i>Arduino-Board Uno</i> .....	6
Abbildung 3-4: Bilder des Fahrzeuges aus zwei Perspektiven .....	6
Abbildung 3-5: Aufbau Mecanum-Rad .....	7
Abbildung 3-6: Geschwindigkeitsvektoren der Mecanum-Räder eines Fahrzeuges .....	7
Abbildung 3-7: Bewegungsrichtung Fahrzeug mit Mecanum-Rädern nach Baumgarten, 1990:S.7f .....	8
Abbildung 3-8: PWM-Signal mit verschiedenen Tastverhältnissen.....	9
Abbildung 3-9: Timer-Register für Timer 0 .....	10
Abbildung 3-10: Ultraschallsensor Dual Ultra Sonic.....	11
Abbildung 3-11: Datenübertragung vom Master zu den Slaves über RS485 .....	12
Abbildung 3-12: Arduino-Board V1.1.....	13
Abbildung 3-13: Arduino IO Expansion Board.....	14
Abbildung 3-14: Anschluss der Komponenten an Arduino-Board.....	14
Abbildung 3-15: Interrupt Service Routine .....	16
Abbildung 3-16: Vereinfachter Regelkreis.....	17
Abbildung 3-17: Fahrrichtung abhängig von den Winkeln .....	19
Abbildung 3-18: OptiTrack S250e .....	23
Abbildung 5-1: Schnittstellen zwischen den Systemen .....	27
Abbildung 5-2: Funktionsweise zum Abrufen vom Webdokumenten im WWW.....	28
Abbildung 5-3: REST API .....	28
Abbildung 5-4: Bestandteile einer URI .....	29
Abbildung 5-5: Aufbau des Trackingsystems .....	30
Abbildung 5-6: Installation der Trackingkamera .....	31
Abbildung 5-7: HTML-Seite des Trackers .....	31
Abbildung 5-8: Kalibriervorlage (links) und Kalibrierturm (rechts).....	32
Abbildung 5-9: Koordinatensystem des Trackers nach Kalibrierung .....	33
Abbildung 5-10: Trackingsystem "Bodies" .....	33
Abbildung 5-11: <i>HC05 Bluetooth Wireless RF-Transceiver-Modul RS232</i> .....	34
Abbildung 5-12: Verbinden der HC05-Moduls mit dem <i>Arduino Board "Arduino Mega"</i> .....	35
Abbildung 5-13: Befehlsfolge zum Einstellen des HC05-Moduls .....	35
Abbildung 5-14: Verbinden des HC05-Moduls mit <i>Arduino Board</i> des Fahrzeuges .....	36
Abbildung 6-1: Kaskadenregelung .....	41
Abbildung 6-2: Trajektorie nach Hofschulte, 2022: S.191 .....	43
Abbildung 6-3: Geschwindigkeitsverlauf mit Rampenprofil nach Mareczek, 2020: S. 27 .....	43
Abbildung 6-4: Koordinatensysteme .....	46
Abbildung 6-5: Markerposition in Trackingsystem.....	52
Abbildung 6-6: Markerposition auf Fahrzeug .....	52
Abbildung 7-1: Zellaufteilung bei einem A-Stern Algorithmus.....	63

Abbildung 7-2: Gesamtpotential .....	63
Abbildung 7-3: Erstellen einer <i>Probabilistic Roadmap</i> .....	64
Abbildung 7-4: Rapid Exploring Random Trees .....	65
Abbildung 7-5: Lage dreier Punkte im Raum nach Borke, 1988 .....	69
Abbildung 7-6: Anpassen des belegten Konfigurationraums .....	71
Abbildung 7-7: Ausgangssituation des betrachteten RRT-Algorithmus .....	73
Abbildung 7-8: Flussdiagramm zur Pfadsuche.....	74
Abbildung 7-9: Geschwindigkeitsverlauf an Via-Punkten.....	76
Abbildung 7-10: Geschwindigkeitsverläufe der Streckenabschnitte .....	77
Abbildung 7-11: Geschwindigkeitsdifferenz an Via-Punkt.....	79
Abbildung 8-1: Optimierungsmaßnahmen an Via-Punkten.....	82

**Tabellenverzeichnis**

Tabelle 3-1: Pin-Belegung .....	15
Tabelle 4-1: <i>Product Backlog</i> Teil 1 .....	24
Tabelle 4-2: <i>Product Backlog</i> Teil 2 .....	25
Tabelle 5-1: <i>Sprint Backlog</i> #1 .....	26
Tabelle 6-1: Sprint Backlog #2 .....	40
Tabelle 7-1: Sprint Backlog #3 .....	61
Tabelle 7-2: Vergleich der Pfadplanungsverfahren .....	66
Tabelle 8-1: Offene Anforderungen .....	84

**Programmcodeverzeichnis**

Programmcode 3-1: Definieren von Fahrzeug Parametern .....	18
Programmcode 3-2: Definition der Funktion "SetCarMove" .....	19
Programmcode 3-3: Definition der Funktion "setCarAdvance" .....	19
Programmcode 3-4: Deklarierung der Motoren und Sensoren sowie Funktion <i>sonarsUpdate()</i> .....	20
Programmcode 3-5: Funktion „goAhead“ .....	21
Programmcode 3-6: Array <i>motion</i> und Funktion <i>demoWithSensors</i> .....	21
Programmcode 3-7: Funktion <i>void setup()</i> und <i>void loop()</i> .....	22
Programmcode 5-1: Auslesen der Trackingdaten.....	37
Programmcode 5-2: Senden von Daten über serielle Schnittstelle von Python .....	38
Programmcode 5-3: Arduino Sketch zur Annahme und Verarbeitung von Befehlen .....	38
Programmcode 6-1: Auslesen und Verarbeiten von seriellen Daten zur Bewegungsausführung .....	49
Programmcode 6-2: Funktion <i>void setup()</i> für Bewegungssteuerung.....	50
Programmcode 6-3: Interrupt Service Routine für Timer 0 .....	51
Programmcode 6-4: Lageregelung „Auslesen von Trackingdaten und Korrektur des Winkels“ .....	51
Programmcode 6-5: Lageregelung „Umrechnung der Soll-Position in Bezug auf das Fahrzeug“ .....	52
Programmcode 6-6: Lageregelung „Berechnung der Fahrgeschwindigkeit und der Fahrrichtung“ .....	53
Programmcode 6-7: Lageregelung „Berechnung der Winkelgeschwindigkeit“ .....	53
Programmcode 6-8: Lageregelung „Übergabe der Fahrbefehle an <i>Arduino</i> “ .....	54
Programmcode 6-9: Lageregelung „Aufrufen der Lageregelung“ .....	54
Programmcode 6-10: Bahnplanung „Streckenberechnung“.....	55
Programmcode 6-11: Bahnplanung „Berechnung der maximalen Zeit“ .....	56
Programmcode 6-12: Bahnplanung „Berechnung von Geschwindigkeiten und Zeiten“ .....	56
Programmcode 6-13: Bahnplanung „Listen für Interpolation“ .....	57
Programmcode 6-14: Bahnplanung "Funktion für Interpolation" .....	57
Programmcode 6-15: Bahnplanung "Bestimmen der interpolierten Punkte und Geschwindigkeiten" ..	58
Programmcode 6-16: Bahnplanung "Durchlaufen der Lageregelung" .....	59
Programmcode 6-17: Bahnplanung "Korrektur der Geschwindigkeitswerte" .....	59
Programmcode 6-18: Bahnplanung "Berechnung der Fahrrichtung" .....	59
Programmcode 6-19: Bahnplanung "Halten der Zielposition und Zielorientierung" .....	60
Programmcode 7-1: Pfadplanung "Funktionen für AR-Kontrolle und Hinderniskontrolle" .....	68
Programmcode 7-2: Pfadplanung "Funktion für Kollisionskontrolle" .....	70
Programmcode 7-3: Pfadplanung "Definition des Konfigurationraums" .....	71
Programmcode 7-4: Pfadplanung "Definieren des Zielpunktes" .....	72
Programmcode 7-5: Pfadplanung "Auswahl der Punkte für Pfad" .....	75
Programmcode 7-6: Pfadplanung "Geschwindigkeitsverlauf eines Streckenabschnitts" .....	78
Programmcode 7-7: Pfadplanung "Abfahren des Pfades" .....	79

**Abkürzungen**

<b>Abkürzung</b>	<b>Bezeichnung</b>
API	Application Programming Interface
COM-Port	Communication port
DOF	Degrees of freedom, Freiheitsgrade
FTF	Fahrerloses Transportfahrzeug
FTS	Fahrerloses Transportsystem
HTML	Hypertext Markup Language
HTTP	Hypertext Transferprotokoll
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation
LAN	Local Area Network
LED	Light-emitting diode, Leuchtdiode
PRM	Probabilistic Roadmaps
PoE	Power over Ethernet
PWM	Pulsweitenmodulation
REST	Representational State Transfer
RRT	Rapid Exploring Random Tree
USB	Universal Serial Bus
URI	Uniform Ressource Identifier
WWW	World Wide Web

**Formeln**

<b>Formel</b>	<b>Einheit</b>	<b>Bezeichnung</b>
$A$	$\text{mm}^2$	Fläche
$a_m$	$\text{mm/s}^2$	Maximale Beschleunigung
$a_{m,i}$	$\text{mm/s}^2$	Maximale Beschleunigung einer Komponente
${}^{FTF}H_0$	-	Homogene Transformationsmatrix vom Koordinatensystem des Fahrzeugs zum Bezugskoordinatensystem
${}^0H_{FTF}$	-	Homogene Transformationsmatrix vom Bezugskoordinatensystem zum Koordinatensystem des Fahrzeugs
$k_p$	$1/\text{s}$	Proportionalitätsfaktor translatorisch
$k_\varphi$	$1/\text{s}$	Proportionalitätsfaktor rotatorisch
$R_{ges}$	-	Rotationsmatrix
$R_y$	-	Elementardrehung um y-Achse
$R_z$	-	Elementardrehung um z-Achse
$s$	$\text{mm}$	Strecke
$s_e$	$\text{mm}$	Gesamtstrecke
$s_{e,i}$	$\text{mm}$	Gesamtstrecke einer Komponenten
$s_{FTF}$	$\text{mm}$	Position Fahrzeug
$s_{soll}$	$\text{mm}$	Soll-Position Fahrzeug
$\dot{s}$	$\text{mm/s}$	Geschwindigkeit
$\ddot{s}$	$\text{mm/s}^2$	Beschleunigung
$t$	$\text{s}$	Zeit
$t_a$	$\text{s}$	Beschleunigungszeit
$t_b$	$\text{s}$	Zeit, ab der Bremsvorgang beginnt
$t_e$	$\text{s}$	Gesamtzeit
$v$	$\text{mm/s}$	Geschwindigkeit

---

$v_m$	mm/s	Maximale Geschwindigkeit
$v_{m,i}$	mm/s	Maximale Geschwindigkeit einer Komponente
$v_{trans}$	mm	Translatorische Geschwindigkeit
$v_x$	mm/s	Geschwindigkeit in x-Richtung
$v_y$	mm/s	Geschwindigkeit in y-Richtung
$x_{FTS}$	mm	x-Koordinate des Fahrzeugs
$x_{soll}$	mm	x-soll-Koordinate des Fahrzeugs
$y_{FTS}$	mm	y-Koordinate des Fahrzeugs
$y_{soll}$	mm	y-soll-Koordinate des Fahrzeugs
$\Delta s$	mm	Streckendifferenz zwischen Fahrzeug und Soll-Position
$\Delta s_x$	mm	x-Anteil der Streckendifferenz zwischen Fahrzeug und Soll-Position
$\Delta s_y$	mm	y-Anteil der Streckendifferenz zwischen Fahrzeug und Soll-Position
$\Delta x$	mm	Positionsabweichung in x-Richtung
$\Delta y$	mm	Positionsabweichung in y-Richtung
$\Delta\varphi$	rad	Winkelabweichung
$\varphi_{soll}$	rad	Soll-Orientierung
$\varphi_{FTS}$	rad	Orientierung des Fahrzeugs
$\omega$	rad/s	Winkelgeschwindigkeit

## 1 Einleitung

Im Zuge der Industrialisierung ändert sich kontinuierlich die Art der Produktherstellung. Von der Mechanisierung im 18. Jahrhundert, über die Elektrifizierung im 19.Jahrhundert bis zur Automatisierung und Digitalisierung ab den 1970er Jahren hat sich die Produktionsart fortgehend weiterentwickelt. Die letzte Industriealisierungsstufe "Industrie 4.0" legt den Fokus auf die Vernetzung der Fabriken entlang der Wertschöpfungskette. Sie beschreibt eine Produktionskette, in der der Mensch fast gänzlich durch Maschinen und Roboter ersetzt wird. Der Mensch hat nur noch Kontroll- und Eingreiffunktionen. Während der Industrialisierung gewann die Bewegungssteuerung bereits mit der Elektrifizierung und vor allem mit der Automatisierung an Bedeutung. Der Begriff Bewegungssteuerung bezeichnet die Steuerung, Koordinierung und Überwachung einer Bewegung. Sie ist daher eine grundlegende Voraussetzung einer jeden Maschine oder eines jeden Roboters. Die Hauptaufgabe der Bewegungssteuerung bei Robotern ist die Entgegennahme und Abarbeitung von Roboterprogrammen und Bewegungsbefehlen. Mit Hilfe von Interpolationen werden passende zeitliche Zwischenstellungen anhand einer gewünschten Zielpose erstellt. Diese dienen dazu, den Zustand des System zu verschiedenen Zeitpunkten zu beschreiben (vgl. Wenz, 2008: S.1).

Mit der Automatisierung, insbesondere mit der Vernetzung der Fabriken, werden Transportaufgaben weitestgehend von fahrerlosen Transportsystemen (FTS) übernommen. Kennzeichnend für FTS ist eine flurgebundene Förderung durch die Verwendung von fahrerlosen Transportfahrzeugen (FTF). Diese befördern auf Grundlage eines definierten Wegenetzes Güter unterschiedlicher Größe, Masse und Formen. Sie werden automatisch gesteuert und ohne menschliche Berührung betrieben. Sie sind daher in einer automatisierten und vernetzten Fabrik nicht wegzudenken (vgl. Linde, 2023).

Aus diesem Grund beschäftigen sich Universitäten und Hochschulen mit diesen Themenangeboten. Auch die Hochschule Hannover hat den Anspruch sich mit diesen Themen auseinanderzusetzen. Deshalb werden in verschiedenen Laboren wie z.B. dem Robotik-Labor die Studenten durch diverse Workshops und Vorlesungen mit Robotern und den Themen "Vernetzung" und "Automatisierung" vertraut gemacht. Die Labore verfügen beispielsweise über programmierbare virtuelle Fabriken, Minitaturreproduktionssystemen und verschiedene Roboter. Die Studenten haben hier die Gelegenheit ein Verständnis für die Programmierung und die Funktionsweise solcher Systeme aufzubauen.

Das Ergebnis dieser Arbeit soll Bestandteil des Robotik-Labors werden und den Studenten aufzeigen, welche Möglichkeiten der Bewegungssteuerung eines FTS bestehen. Das Ziel der Aufgabenstellung ist die Entwicklung einer Bewegungssteuerung eines FTS für eine Punkt-zu-Punkt-Bewegung. Darüber hinaus soll in dieser Arbeit eine autonome Pfadplanung generiert werden. Mit Hilfe einer Bewegungssteuerung für eine Mehrpunktbewegung sollen die ermittelten Punkte des Pfades abgefahren werden. Dafür stellt die Hochschule FTF sowie ein Trackingsystem zur Verfügung. Zur Umsetzung dieser Aufgabe wird zunächst die Vorgehensweise und die Ausgangssituation betrachtet. Bei der Betrachtung der Ausgangssituation wird analysiert, wie das Fahrzeug aufgebaut ist, aus welchen Komponenten es besteht, wie das Trackingsystem funktioniert etc. Anschließend werden Anforderungen an die Bewegungssteuerung definiert. Im nächsten Schritt werden die Schnittstellen begutachtet und gegebenenfalls die Systeme um passende Schnittstellen für eine Kommunikation erweitert. Im Anschluss erfolgt die Programmierung einer Trajektorien- und Pfadplanung. Die Trajektorienplanung erfolgt für eine Punkt-zu-Punkt-Verbindung und wird durch die Interpolation von Zwischenposen generiert. Für die Pfadplanung werden verschiedene Algorithmen der Pfadfindung untersucht. Ziel dabei ist es, eine autonome Pfadplanung zu ermöglichen.

Eine anfangs theoretische Betrachtung der notwendigen Inhalte erfolgt nicht in dieser Arbeit. Die benötigten theoretischen Informationen fließen in die entsprechenden Kapitel ein und sollen so den praktischen Inhalt an passenden Stellen ergänzen. Als Ergebnis dieser Arbeit sollen die Programmcodes für die Bewegungssteuerung vorliegen. Zur besseren Lesbarkeit wird in dieser Arbeit das generische Maskulinum verwendet. Die in dieser Arbeit verwendeten Personenbezeichnungen beziehen sich, sofern nicht anders kenntlich gemacht, auf alle Geschlechter.

## 2 Vorgehensweise zur Entwicklung der Bewegungssteuerung

Die Vorgehensweise zur Realisierung der Bewegungssteuerung erfolgt in Anlehnung der agilen Projektmanagementmethode mittels *Scrum*. Die *Scrum*-Methode entstammt der Softwareentwicklung, kann allerdings auf viele weitere Bereiche adaptiert werden. Es handelt sich um ein Rahmenwerk für agiles Projektmanagement und gibt daher keine konkreten Techniken vor, sondern Rahmenbedingungen wie z.B. Projektrollen und einen Prozessablauf. (vgl. Preußig, 2020: S.135-142; Wirdemann & Mainusch, 2017: S.28-32)

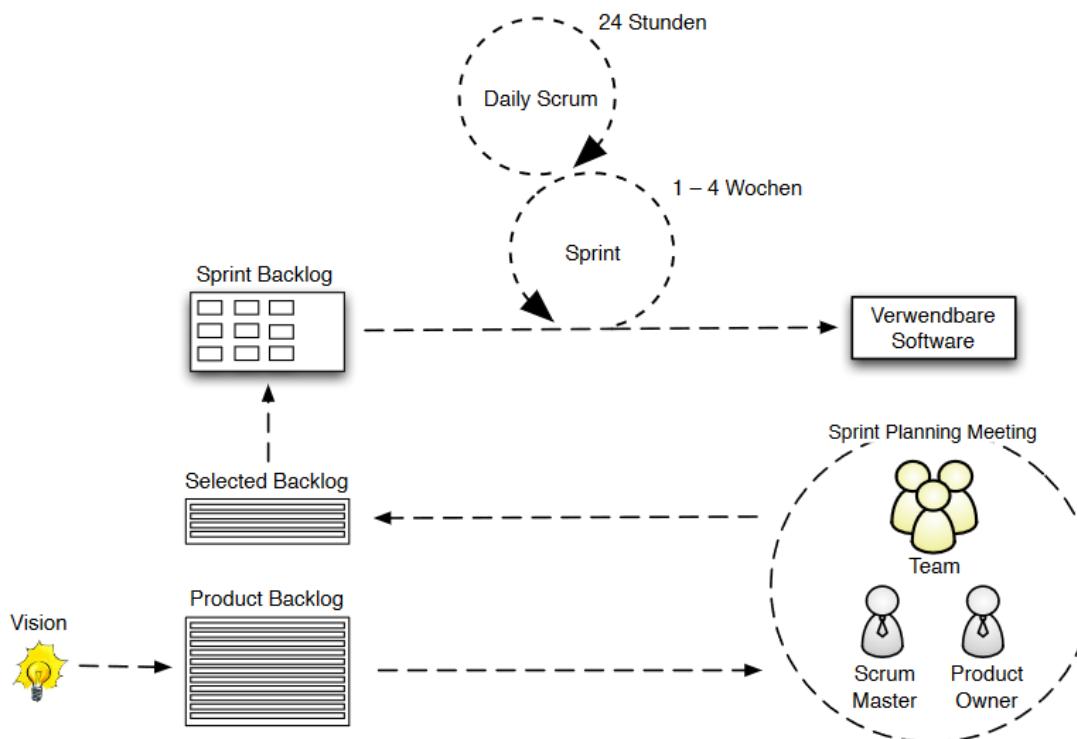


Abbildung 2-1: Prozessablauf *Scrum*  
( Wirdemann & Mainusch, 2017: S.30)

Die Abbildung 2-1 stellt den Prozessablauf eines *Scrum* dar. Zu Beginn des Prozesses steht die Vision bzw. die Idee eines Produktes. Aus dieser Idee wird eine Zielsetzung bzw. Aufgabe definiert. Auf dieser Basis erstellt der *Product Owner* ein *Product Backlog*. Der *Product Owner* ist der Visionär sowie der Besitzer des auszuarbeitenden Produktes. Das *Product Backlog* enthält die insgesamt umzusetzenden Aufgaben bzw. Anforderung und priorisiert diese. Es stellt somit das erste und elementare Artefakt des Prozesses dar. Der nächste Schritt ist das *Sprint Planning Meeting*. Bei diesem Meeting kommen *Scrum Master*, *Product Owner* und das Team zusammen. Hierbei ist der *Scrum Master* verantwortlich für die Einhaltung des *Scrum*-Prozesses. Der *Product Owner* ist als Fachexperte für die Anforderungen verantwortlich. Während des Meetings wird das Ziel des ersten *Sprints* festgelegt und ein *Selected Backlog* bzw. ein *Sprint Backlog* zusammengestellt. Ein *Sprint* entspricht einer Iteration und kann einen Zeitraum von ein bis vier Wochen einnehmen. Im *Sprint Backlog* wiederrum werden die Anforderungen und umzusetzenden Aufgaben für diese Iteration erfasst. Diese Aufgaben werden innerhalb des *Sprints* abgearbeitet, sodass am Ende eines *Sprints* ein Inkrement, ein Teilprodukt, entsteht. Innerhalb eines *Sprints* trifft sich das Projektteam täglich zu einer festen Uhrzeit, um die jeweiligen Tagesaufgaben zu planen sowie ein Feedback zu den bereits absolvierten Ergebnissen und Aufgaben zu geben. Nachdem ein *Sprint* abgearbeitet wurde, wird in einem weiteren *Sprint Planning Meeting* der nächste *Sprint* geplant. Dies wiederholt sich solange bis das Projekt im Idealfall erfolgreich abgeschlossen wurde und eine Endprodukt vorliegt. (vgl. Wirdemann & Mainusch, 2017: S.25-31)

In Anlehnung an diesen Prozess wird nach Betrachtung der Ausgangssituation ebenfalls ein *Product Backlog* erstellt und daraus einzelne *Sprints* sowie *Sprint Backlogs* ausgearbeitet. Allerdings handelt es sich bei dieser Arbeit um eine Einzelarbeit. Daher übernimmt der Verfasser sowohl die Rollen des *Scrum Masters*, des Projektteams und zum Teil auch die Rolle des *Product Owners*. Lediglich die Hochschule Hannover als Besitzer und Aufgabensteller nimmt ebenfalls die Rolle des *Product Owners* mit ein.

Damit der *Product Owner* ein geeignetes Ziel definieren kann, muss die Ausgangssituation bekannt sein. Daher wird diese im ersten Schritt analysiert. Auf Basis dieser Analyse kann der *Product Owner* anschließend sein *Product Backlog* formulieren. Aus dem *Product Backlog* ergeben sich für diese Arbeit vier Sprints. Davon sind drei Sprints Bestandteil dieser Arbeit.

Zur Abarbeitung dieser Aufgabe wird ein FTS sowie ein Trackingsystem zur Verfügung gestellt. Diese Systeme werden zunächst vorgestellt und die Ausgangssituation beschrieben. Anschließend erfolgt die Entwicklung der Bewegungssteuerung. Diese Realisierung erfolgt iterativ. Das bedeutet in diesem Zusammenhang, dass der Programmcode mit jedem Sprint ausgebaut und verbessert wird. Im Folgenden werden die Ausgangssituation und *Sprints* kurz erläutert.

Im ersten Schritt wird die Ausgangssituation analysiert, um ein Verständnis für die Aufgabe zu schaffen. Das FTF besteht aus mehreren Komponenten und wird mit der *Arduino-Umgebung „Arduino IDE“* programmiert. Daher werden sowohl die einzelnen Komponenten und die technischen Grundlagen diesbezüglich betrachtet, als auch die *Arduino IDE* und der Aufbau eines Arduinoprogramms beschrieben. Des Weiteren wird das verwendete Trackingsystem näher erläutert.

Der erste *Sprint* befasst sich mit den Schnittstellen der Systeme. Es wird begutachtet, welche Schnittstellen vorhanden sind und welche Kommunikationswege sich zwischen den Systemen zur Datenübertragung eignen. Anschließend wird eine geeignete Schnittstelle implementiert. Vor der Implementierung der Schnittstelle ist es erforderlich, zunächst das Trackingsystem zu installieren und zu kalibrieren. Bestandteil dieses *Sprints* ist zudem die Programmierung der ersten vereinfachten Bewegungssteuerung in der lediglich Fahrbefehle unabhängig vom Standort des Fahrzeugs in umgewandelt werden.

Im zweiten *Sprint* wird die Bewegungssteuerung durch eine Bahnplanung bzw. Trajektorienplanung erweitert. Es werden nun Start und Ziel erfasst bzw. vorgegeben und eine Punkt-zu-Punkt-Steuerung und Bewegung programmiert. Hierbei werden zudem die Regelung des Systems sowie die Transformation von Koordinaten betrachtet.

Der letzte *Sprint* dieser Arbeit beschäftigt sich mit der autonomen Pfadsuche und Pfadplanung. Im zweiten *Sprint* wird eine geradlinige Bewegung zwischen Start und Ziel generiert. In diesem *Sprint* wird geschaut, welcher Pfad zum Ziel führt, wenn eine direkte Verbindung gestört wird z.B. durch ein Hindernis. Der generierte Pfad wird mittels Bewegungssteuerung für eine Mehrpunktbewegung abgefahren.

Zuletzt erfolgt eine Schlussbetrachtung. Diese fasst die Ergebnisse dieser Arbeit zusammen. Zudem werden die positiven und negativen Aspekte der Arbeit in einer kurzen Diskussion gegenübergestellt. Mittels eines kurzen Ausblicks werden Hinweise für weiterführende Arbeiten gegeben.

### 3 Ausgangssituation

Zur Bearbeitung dieser Masterarbeit stellt die Hochschule Hannover ein FTS zur Verfügung. Wesentlicher Träger dieser Arbeit sind dabei das verwendete FTF und ein Trackingsystem. Im Folgenden werden die einzelnen Komponenten bzw. Systeme näher beschrieben.

#### 3.1 Arduino

Für die Entwicklung der Bewegungssteuerung werden verschieden Programme und Programmiersprachen verwendet. Zum einen wird die *Arduino*-Plattform und zum anderen die Software Visual Studio Code zum Erstellen der Programmcodes benötigt. Im Folgenden soll der *Arduino* im Detail betrachtet werden.

*Arduino* wurde im Jahr 2005 von dem *Interaction Design Institute Ivrea* entwickelt. Das italienische Institut erkannte die Problematik, dass keine einfach anwendbaren und preiswerten Mikrocontrollersysteme auf dem Markt erhältlich waren. Somit wurde in Zusammenarbeit mit Elektroingenieuren ein eigenes integriertes System verwirklicht. Es entstand die erste Serie der *Arduino*. (vgl. Bruhlmann, 2012)

Seither wurde *Arduino* kontinuierlich weiterentwickelt. *Arduino* adaptiert neue Technologien und Komponenten für die sogenannten *Arduino-Boards*. Da es sich sowohl bei der Hardware als auch bei der Software um Open-Source handelt, verbreitet sich der *Arduino* schnell auf dem Markt und findet Einzug in Seminare und Vorlesungen von Hochschulen und Universitäten. Für viele Codebeispiele wird nur wenig technisches Verständnis vorausgesetzt, wodurch *Arduino* sehr benutzerfreundlich ist. So können sehr umfangreiche Projekte entwickelt werden. Aus diesem Grund erfreut sich *Arduino* großer Beliebtheit. (vgl. Margolis, 2012; Odendahl, Finn, & Wenger, 2010)

Meist wird der Begriff „*Arduino*“ lediglich mit den Hardwarekomponenten in Verbindung gebracht, allerdings inkludiert er ebenfalls den gleichnamigen Softwarebereich. Nur durch Zusammenarbeit mit beiden Komponenten kann die physikalische Welt gesteuert und wahrgenommen werden. Die Entwicklungsumgebung trägt den Namen „*Arduino IDE*“. Damit werden sogenannte „*Sketches*“ auf dem Computer programmiert. Diese *Sketche* werden anschließend vom integrierten Compiler in Instruktionen umgewandelt, welche der Mikrocontroller der *Arduino-Boards* interpretieren kann. Die Übertragung des *Sketches* von der Entwicklungsumgebung auf dem Computer zum Speicher des *Boards* wird durch einen USB-Port per serieller Datenübertragung realisiert. (vgl. Margolis, 2012; Bruhlmann, 2012)

Die Abbildung 3-1 zeigt ein Screenshot eines leeren *Arduino-Sketches* der *Arduino IDE*. Es ist zu erkennen, dass sich das Programm in zwei Bereiche untergliedern lässt. Zum einen in den Bereich „*void setup()*“ und zum anderen in den Bereich „*void loop()*“. Im Bereich „*void setup*“ werden jene Programmteile platziert, welche einmalig beim Starten des Programms abgerufen werden. Während die Programmteile des „*void loop()*“-Bereiches dauerhaft in einer Schleife (engl. Loop) ausgeführt werden. Das Schlüsselwort „*void*“ gibt an, dass eine Funktion definiert wird, die keinen Wert zurückliefer, sodass eine Prozedur mit einer bestimmten Tätigkeit ausgeführt wird. Des Weiteren können außerhalb dieser Bereiche weitere Funktionen definiert werden. Auf diese kann dann im „*void setup*“ oder im „*void loop*“ zugegriffen werden. (vgl. Bräunl, 2022: S.54ff)

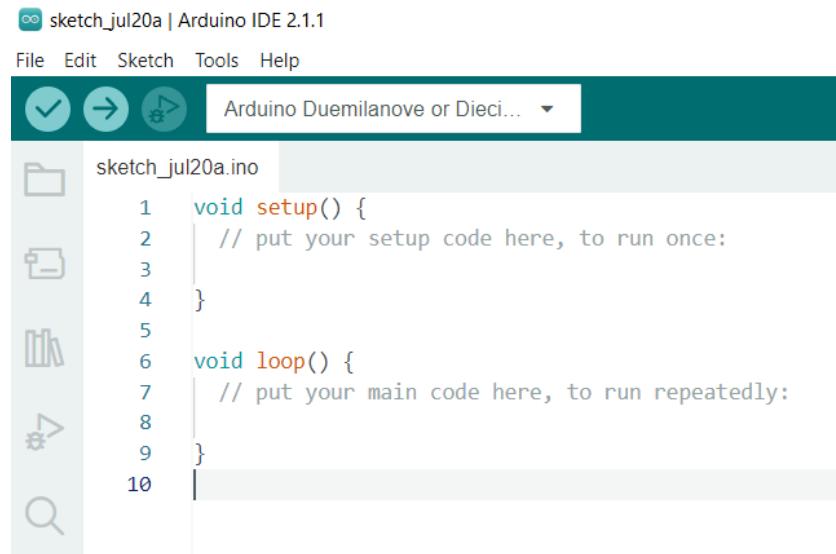


Abbildung 3-1: Leeres Arduino-Sketch

Die *Arduino*-Programmiersprache basiert auf „*Wiring*“. Dies ist ähnlich zu *Arduino* ebenfalls eine Open-Source Programmierumgebung für Mikrocontroller. Der Programmcode wird in einer vereinfachten Form der „C/C++“-Programmiersprache geschrieben. Der Mikrocontroller des *Arduino-Boards* versteht die Sprache C/C++ allerdings nicht, daher ist es notwendig den Programmcode in Maschinensprache zu übersetzen. Dies wird durch den *Compiler* ermöglicht (s. Abbildung 3-2). Dies ist jedoch eine Ein-Weg-Verbindung. Das bedeutet, dass der *Compiler* die C/C++-Sprache in Maschinensprache übersetzen kann, umgekehrt allerdings nicht das Programm in Maschinensprache vom Mikrocontroller in die *Arduino*-Sprache zurücktransformiert werden kann. (vgl. Bartmann, 2012)

Abbildung 3-2: Der Compiler als Dolmetscher  
(Bartmann, 2012)

Sobald der *Arduino-Sketch* fertig geschrieben und kompiliert wurde, kann dieser über *Upload* und dem USB-Port auf das *Board* übertragen werden. Wenn der *Sketch* auf das *Board* geladen wurde, wird auf diesem der geschriebene Code unmittelbar ausgeführt. Auf dem *Board* befinden sich Ein- und Ausgänge. An diese Ein- und Ausgänge (Pins) können Sensoren und Aktuatoren angeschlossen, gesteuert und überwacht werden. Mithilfe der Sensoren werden Werte und bestimmte Aspekte aus der realen, physischen Welt in Strom umgewandelt, woraufhin die *Arduino-Hardware* reagieren kann. Die Aktuatoren hingegen wandeln Strom, den sie vom *Board* erhalten, in mechanische Bewegung um. (vgl. Margolis, 2012)

Unter dem Begriff „*Boards*“ selbst werden Leiterplatten verstanden, die mit elektrischen Komponenten wie Schaltkreise, Widerstände, Kondensatoren etc. bestückt sind. Des Weiteren sind diese Leiterplatten mit diversen Anschlüssen und Schnittstellen versehen. Über diese kann eine Verbindung zur physischen Welt hergestellt werden. (vgl. Bruhlmann, 2012)

Die Abbildung 3-3 zeigt das Standardboard, den *Arduino Uno*. Dieser läuft mit dem Mikrocontroller ATmega328 und besitzt 14 digitale Input/Output-Pins sowie sechs analoge Pins. Die Pin 0 und Pin 1 werden auch als RX/TX-Pins bezeichnet. Über diese erfolgt die serielle Datenübertragung. Neben den Pins verfügt das Board über einen Speicher von 32KB. (vgl. Jänisch & Donges, 2017: S.8)

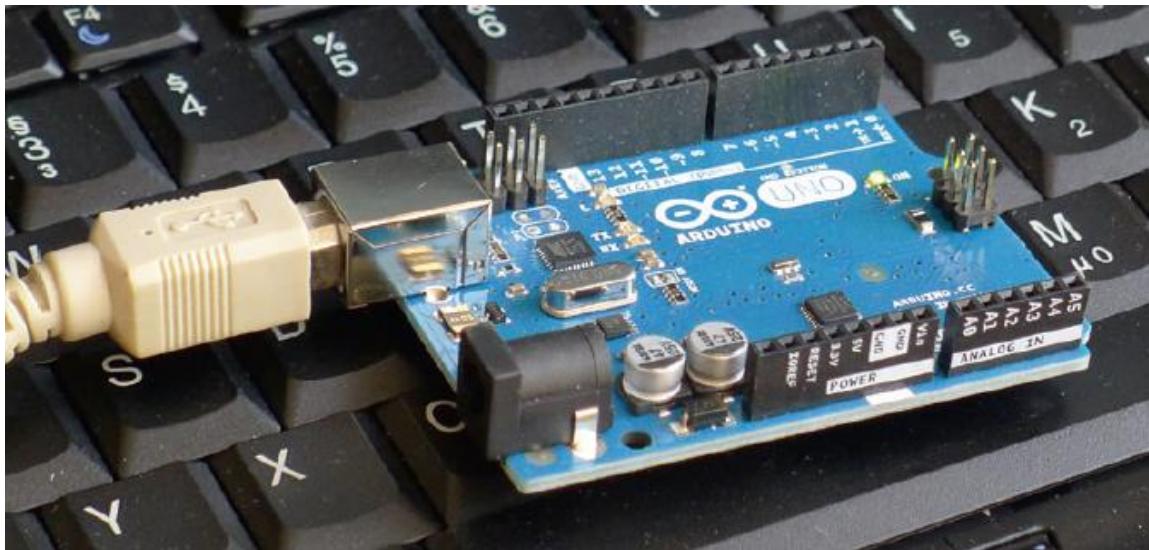


Abbildung 3-3: Das *Arduino-Board Uno*  
( Jänisch & Donges, 2017: S.8)

### 3.2 Fahrerloses Transport Fahrzeug mit Mecanum-Wheels

Die Hochschule stellt ein FTF mit *Mecanum Wheels* zur Verfügung. Dieses verfügt über vier Mecanum Räder (engl. Wheels), vier Servomotoren inklusive Encoder zum Betreiben der Räder und vier Ultraschallsensoren. Diese Komponenten werden über zwei *Arduino-Boards* miteinander verknüpft (s. Abbildung 3-4). Die folgenden Unterkapitel beschreiben die jeweiligen Komponenten. Zudem werden von der Firma NEXUS, die das FTF vertreibt, Skripte und Bibliotheken zur Verfügung gestellt. Kleine Abschnitte dieser Bibliotheken sollen ebenfalls erläutert werden, um die Funktionsweise des FTF nachvollziehen zu können.

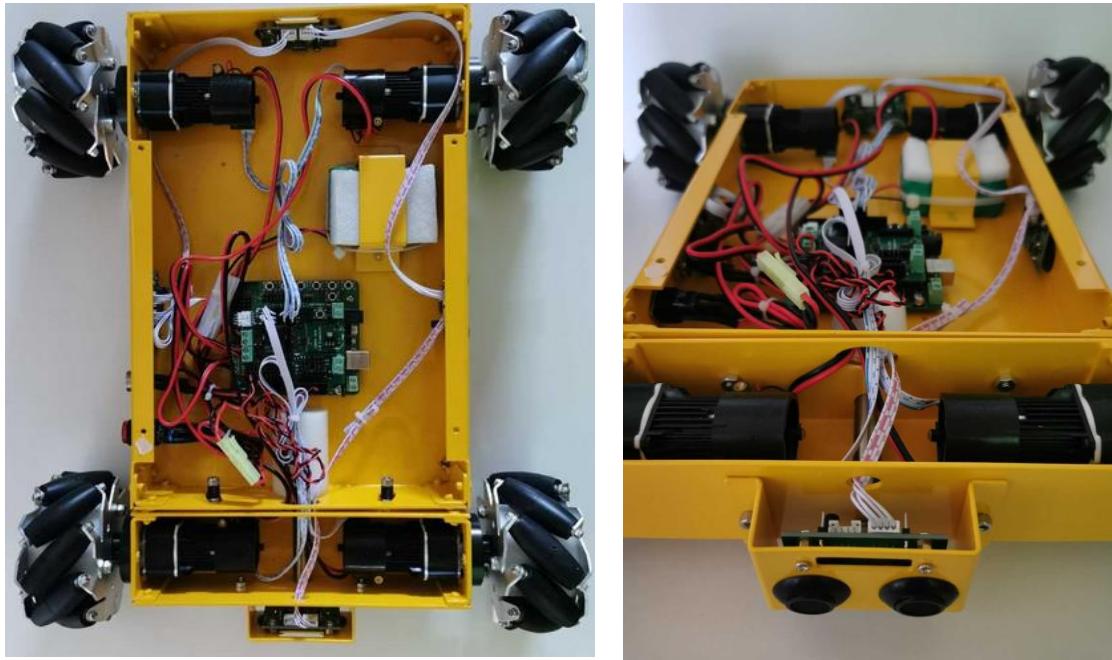


Abbildung 3-4: Bilder des Fahrzeuges aus zwei Perspektiven

### 3.2.1 Aufbau und Kinematik der Mecanum Wheels

Das Mecanum-Rad wurde im Jahre 1973 von Bengt Ilon erfunden. Die Idee des Mecanum-Rades basiert auf dem Konzept eines Zentralrades mit einer Vielzahl an freien Rollen, die in einem bestimmten Winkel am Umfang des Zentralrades angeordnet sind (s. Abbildung 3-5). So entsteht ein zirkuläres Profil. Durch die abgewinkelten Rollen entstehen beim Antreiben des Rades zwei Kraftkomponenten. Einen Teil der Kraft wird in Fahrtrichtung erzeugt, ein anderer Teil orthogonal dazu. Gegeneinander gerichtete Kräfte einzelner Räder an einem Fahrzeug werden über die Achsen und Rahmen kompensiert. Der Winkel zwischen Rollenachse und Zentralrad kann beliebig ausgewählt werden, in der Regel beträgt er allerdings 45°. (vgl. Kanjanawanishkul, 2015)



Abbildung 3-5: Aufbau Mecanum-Rad  
( Generation Robotics, 2023)

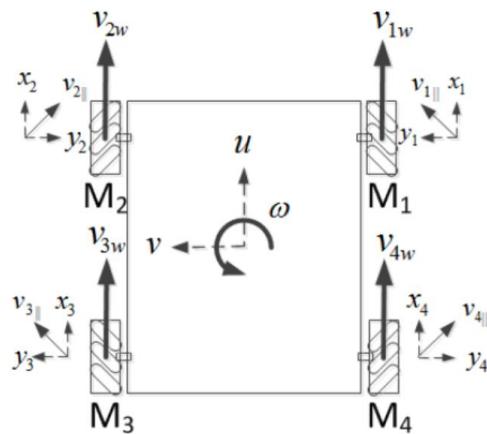


Abbildung 3-6: Geschwindigkeitsvektoren der Mecanum-Räder eines Fahrzeugs  
( Kanjanawanishkul, 2015: S.292)

Während bei herkömmlichen Rädern lediglich zwei Freiheitsgrade (DOF) realisierbar sind, kommt bei dem Einsatz von Mecanum-Rädern ein weiterer Freiheitsgrad hinzu: (vgl. Baumgarten, 1990: S.6)

- 1. DOF: x-Richtung
- 2. DOF: y-Richtung
- 3. DOF: Drehung um z-Achse

Jedes Rad an einem Fahrzeug wird durch einen separaten Motor angetrieben. Durch ein entsprechendes Ansteuern der Drehrichtung und der Drehgeschwindigkeit der Räder kann jedes beliebige Manöver gefahren werden. Die Räder sind rechteckig oder quadratisch an jeder Ecke des Chassis angeordnet (s .Abbildung 3-6). Jedes Rad erzeugt einen individuellen Kraftvektor. Durch die Überlagerung dieser Vektoren ergibt sich in Summe ein Kraftvektor, welcher das Fahrzeug in die gewünschte Richtung bewegt. Um zu beschreiben, wie sich ein Fahrzeug mit Mecanum-Rädern im Raum verhält und bewegt, kann folgende Gleichung verwendet werden: (vgl. Kanjanawanishkul, 2015: S.293)

$$\begin{bmatrix} v_{1w} \\ v_{2w} \\ v_{3w} \\ v_{4w} \end{bmatrix} = \begin{bmatrix} 1 & 1 & l_a + l_b \\ 1 & -1 & -l_a - l_b \\ 1 & 1 & -l_a - l_b \\ 1 & -1 & l_a + l_b \end{bmatrix} x \begin{bmatrix} u \\ v \\ \omega \end{bmatrix} \quad (3-1)$$

Hierbei entsprechen  $v_{1w}, v_{2w}, v_{3w}, v_{4w}$  den translatorischen Geschwindigkeiten der einzelnen Räder und  $u, v, \omega$  den Geschwindigkeiten des Fahrzeuges. Diese können durch Ansteuerung der Motoren erzielt werden, wohingegen die Konstanten  $l_a$  und  $l_b$  für die Distanz zwischen Radachse und Mittelpunkt des

Fahrzeugs in x- und y-Richtung stehen und feste Werte abhängig vom Chassis annehmen (vgl. Kanjanawanishkul, 2015: S.293)

Die Abbildung 3-7 stellt verschiedene Fahrzeugbewegungen dar. Die Kombination von den Bewegungen der vier Räder erlaubt es dem Fahrzeug in jede beliebige Richtung mit jedem beliebigen Winkel zu fahren.

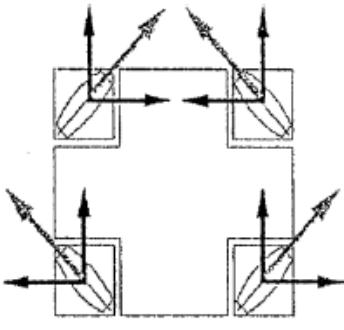
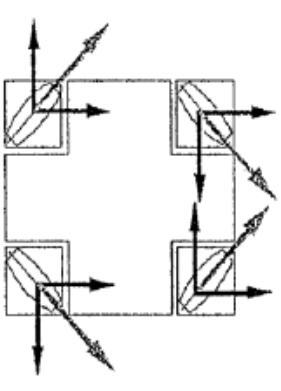
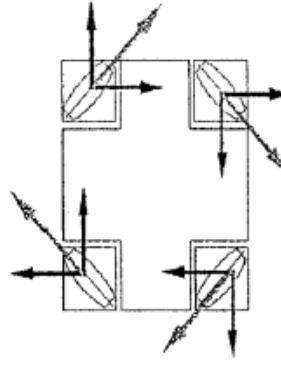
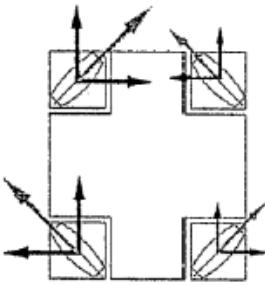
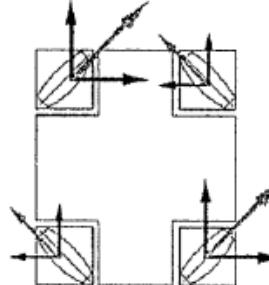
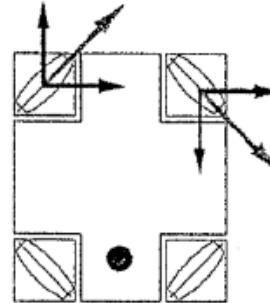
<p><b>(a) Geradeausfahrt</b></p> <p>Alle Räder bewegen sich mit gleicher Geschwindigkeit und gleicher Richtung</p> 	<p><b>(b) Seitwärtsfahrt</b></p> <p>Jedes Rad dreht gegenläufig zum benachbarten, alle Räder bewegen sich mit gleicher Geschwindigkeit</p> 	<p><b>(c) Drehung</b></p> <p>Eine Seite dreht gegenläufig zur anderen Seite, alle Räder bewegen sich mit gleicher Geschwindigkeit</p> 
<p><b>(d) Kurvenfahrt</b></p> <p>Je 2 Räder auf gleicher Seite bewegen sich mit gleicher Geschwindigkeit und alle Räder mit gleichen Drehsinn</p> 	<p><b>(e) Diagonalfahrt</b></p> <p>Je 2 Räder bewegen sich über die Diagonale mit gleicher Geschwindigkeit und alle Räder mit gleichem Drehsinn</p> 	<p><b>(f) Drehung um Mittelpunkt einer Achse</b></p> <p>Je 2 Räder einer Achse laufen gegeneinander mit gleicher Geschwindigkeit</p> 

Abbildung 3-7: Bewegungsrichtung Fahrzeug mit Mecanum-Rädern nach Baumgarten, 1990:S.7f

### 3.2.2 Servomotoren, PWM-Signal und Timer zum Einstellen des PWM-Signals

Die Bewegung des Fahrzeugs wird über vier Servomotoren generiert. Die Bezeichnung Servomotor steht im Allgemeinen für einen drehzahlvariablen elektromotorischen Antrieb, welcher geregelt betrieben wird. Ein Servomotor verfügt über einen Positionssensor, sodass eine präzise Steuerung der Winkelposition sowie der Drehgeschwindigkeit ermöglicht wird. (vgl. Kollmorgen, 2023; Probst, 2022: S.1f)

Die Servomotoren werden über ein PWM-Signal angesteuert. Die Abkürzung PWM steht für Pulsweitenmodulation (engl. Pulse Width Modulation). PWM ist ein Verfahren bei dem an digitalen Ein- und Ausgängen ein analoges Signal simuliert werden kann. Digitale Signale können im Gegensatz zu analogen Signalen lediglich den Wert „An“ und „Aus“ annehmen. Bei PWM wird die Dauer der angelegten Spannung in Abhängigkeit zu einem bestimmten Zyklus an die Ausgänge geliefert. Durch eine Reihe sich wiederholender Ein- und Ausschaltimpulsen wird ein Rechtecksignal generiert. Die „An“-Zeit ist die Zeit, während der eine Spannung anliegt und die „Aus“-Zeit ist jene Zeit, in der die Versorgung ausgeschaltet ist. Die Abbildung 3-8 zeigt beispielhaft drei verschiedene PWM-Signale mit einem Tastverhältnis von 10%, 50% und 90%. Das Tastverhältnis gibt an wie viel Prozent das Signal eingeschaltet ist. Wenn die Versorgung z.B. 5V beträgt und das Tastverhältnis 10%, dann ergibt sich ein analoges Signal von 0,5V. (vgl. Barr, 2001; Hirzel, 2023)

Bei einem *Arduino* geht die Skala für das PWM-Signal von 0 bis 255. Das bedeutet bei 255 liegt das Tastverhältnis bei 100% und die Motoren werden mit maximaler Geschwindigkeit angesteuert. Bei einem Wert von 127 beträgt das Tastverhältnis 50% und die Motoren werden mit halber Geschwindigkeit betrieben. (vgl. Hirzel, 2023)

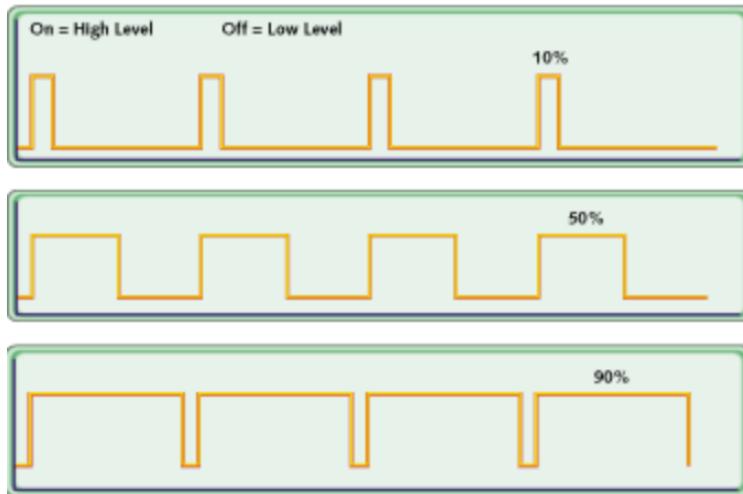


Abbildung 3-8: PWM-Signal mit verschiedenen Tastverhältnissen  
(Barr, 2001)

Mit Hilfe von *Timern* kann das PWM-Signal eingestellt werden. Der Mikrocontroller ATmega328P verfügt über drei *Timer*. Diese werden als *Timer 0*, *Timer 1* und *Timer 2* bezeichnet. *Timer* sind unabhängige Zähler und eine integrierte Hardware im *Arduinocontroller*. Sie funktionieren wie eine Uhr und werden zum Messen von Zeiteignissen verwendet. Zeitfunktionen wie *delay* oder *millis*, aber auch die PWM-Funktionen benötigen einen *Timer*. Alle *Timer* sind abhängig von der *System-Clock*, welche bei einem ATmega328 eine Frequenz von 16MHz hat. Jeder *Timer* verfügt über einen *Prescaler*, welcher den *Timer-Takt* generiert, indem er die *System-Clock* durch einen *Prescaler-Faktor* wie 1,8,64,256 oder 1024 dividiert. *Timer 0* wird für die *Timer-Funktionen* wie *delay()*, *millis()* und *macros()* verwendet, sodass *Timer 1* und *Timer 2* für das Einstellen des PWM-Signals genutzt werden. Jeder *Timer* verfügt über *Timer-Register*. Wenn die Bits dieser Register angepasst werden, wird das *Timer-Verhalten* individuell eingestellt. Die Abbildung 3-9 stellt einen Teil der *Timer-Register* am Beispiel von *Timer 0* dar. (vgl. Atmel, 2015: S.74-88)

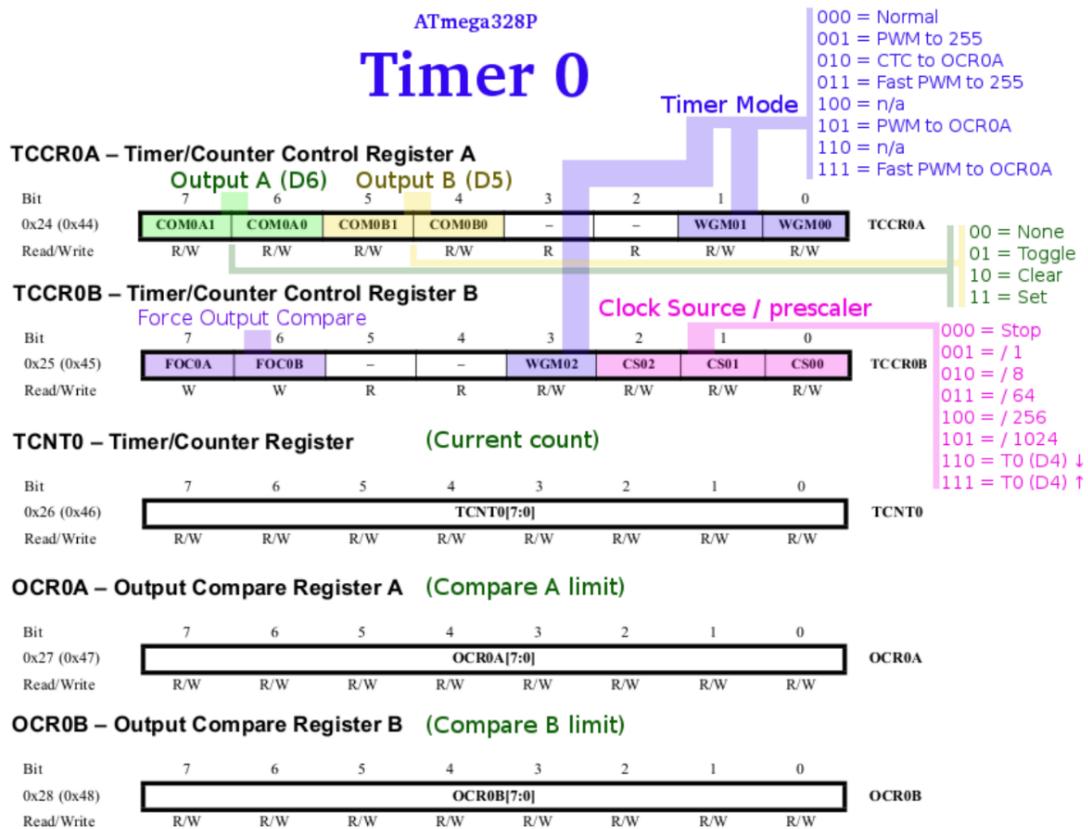


Abbildung 3-9: Timer-Register für Timer 0  
(Gammon, 2015)

Das erste *Timer-Register* ist das *Timer/Counter Control Register TCCR<sub>x</sub>*. Dieses Register dient dazu, den *Timer* zu konfigurieren. Hier wird der *Prescaler* eingestellt. Der *Prescaler* beeinflusst, wie schnell der *Timer* zählt. Wenn ein größerer *Prescaler* gewählt wird, wird der *Timer* langsamer zählen. Das Register unterscheidet sich wiederum in *TCCR<sub>n</sub>A* und *TCCR<sub>n</sub>B*. Diese Register enthalten die Hauptsteuerbits für die *Timer*. Bei den Hauptsteuerbits existieren u.a. die *WGM*-Bits, die *CS*-Bits und die *COM<sub>n</sub>A/COM<sub>n</sub>B*-Bits. *WGM* ist das *Waveform Generation* Bit. Dies fungiert wie ein Modus-Schalter. Es können verschiedene Modi ausgewählt bzw. eingestellt werden. Sie bestimmen, wie der *Timer* zählt und wie er reagiert, wenn er bestimmte Werte erreicht. *CS* sind die *Clock Selected*-Bits. Diese sind zum Einstellen des *Prescalers*. Die *COM<sub>n</sub>A/COM<sub>n</sub>B* sind die *Compare Match Output A/B Mode*-Bits. Diese Modi beeinflussen das Verhalten der Ausgangspins (A oder B) des *Timers*, wenn ein Vergleich zwischen dem *Timer* und einem Vergleichswert stattfindet. Sie können den Ausgang einschalten, ausschalten oder invertieren. Der Vergleichswert wird im *Output Compare Register (OCR<sub>n</sub>A/OCR<sub>n</sub>B)* angegeben. Ein weiteres Register, das *Timer/Counter Register (TCNT<sub>x</sub>)*, speichert den aktuellen Wert des *Timers*. Wenn der *Timer* „tickt“ bzw. zählt, wird dieser Wert erhöht. Es ist so, als würde der *Timer* die Zeit verfolgen, indem er die Anzahl der Ticks zählt. Der *Timer* vergleicht kontinuierlich den aktuellen Wert im *Timer/Counter-Register TCNT<sub>x</sub>* mit dem im *Output Compare Register OCR<sub>n</sub>A/OCR<sub>n</sub>B* gespeicherten Wert. Wenn sie übereinstimmen, kann ein vordefiniertes Ereignis ausgelöst werden. Das x bzw. n in den Registerbezeichnungen kann den Wert 0,1 oder 2 annehmen, je nachdem um welchen *Timer* es sich handelt.

Der PWM-Modus ist ein spezieller Modus des *Timers*. Dieser wird über die *WGM*-Bits eingestellt. Der Modus wird oft für die Steuerung von analogen Signalen wie z.B. Helligkeit oder Geschwindigkeit eingesetzt. Hierbei zählt der *Timer* wiederholt von 0 bis zu einem bestimmten Wert (z.B. 255). Je nachdem, wann der *Timer* ausgelöst wird, ändert sich der Ausgangszustand. So kann das Verhältnis von ein- zu ausgeschaltetem Zustand (Tastverhältnis/engl. *Duty Cycle*) gesteuert werden. Je höher der Wert im *Output Compare Register* ist, desto höher ist der *Duty Cycle*. Mit Hilfe des Datenblatts des Mikrocontrollers kann der *Timer* konfiguriert werden. Es enthält Informationen darüber, wie die Register

eingestellt werden müssen, um die gewünschte Funktionalität zu erreichen. Die Gleichung 3-2 und die Gleichung 3-3 berechnen die gewünschte Frequenz bzw. den *Duty Cycle* im PWM-Modus, wobei "N" den *Prescaler-Faktor* und "MAX" den maximalen Wert im *Output Compare Register* repräsentiert. Die Variable  $f_{clock}$  ist die Taktfrequenz. Bei einem ATmega328 ist diese 16MHz. (vgl. Atmel, 2015: S.74-88)

$$f_{out} = \frac{f_{clock}}{2*N*256} \quad (3-2)$$

$$\text{Duty Cycle} = \frac{(Max+1)}{256} \quad (3-3)$$

### 3.2.3 Ultraschallsensoren Dual UltraSonic und Bus-System RS485

Da das Fahrzeug über vier Ultraschallsensoren verfügt, welche in Reihe geschaltet sind, ist eine Kommunikation bzw. eine serielle Datenübertragung zwischen den Sensoren erforderlich. Diese serielle Datenübertragung erfolgt mittels dem Bussystem RS485. Bevor allerdings das Bussystem betrachtet wird, wird ein kurzer Blick auf die Funktionsweise eines Ultraschallsensors geworfen.

Ultraschallsensoren arbeiten über Schallwellen im Ultraschallbereich. Diese Schallwellen haben eine Frequenz über 20 kHz und sind somit für das menschliche Ohr nicht wahrnehmbar. Das Prinzip der Ultraschallmessung ist in der Natur weit verbreitet. So können beispielsweise Fledermäuse über Ultraschall in der Dunkelheit fliegen und Nahrung suchen. Die Besonderheit von Schallwellen liegt in der hohen Frequenz mit einer kurzen Wellenlänge, sodass eine Ähnlichkeit zu Lichtwellen besteht. In der Technik werden Ultraschallsensoren u.a. für die Abstandsmessung verwendet. Es können z.B. die Abstände zu einem Gegenstand oder einer sich an den Sensor nährenden Person gemessen werden. Der *Ultrasonic-Sensor* (s. Abbildung 3-10) kann Abstände von 4cm bis zu 3m messen. Hierfür sendet der Sensor einen Ultraschallimpuls, der an einem Hindernis wie z.B. einer Wand reflektiert wird. Nach dem Abprallen kehrt der Impuls zurück zum Sensor. Anhand der verstrichenen Zeit zwischen Versand und Rückkehr des Impulses wird die Entfernung berechnet. (vgl. Ericson, 2022)

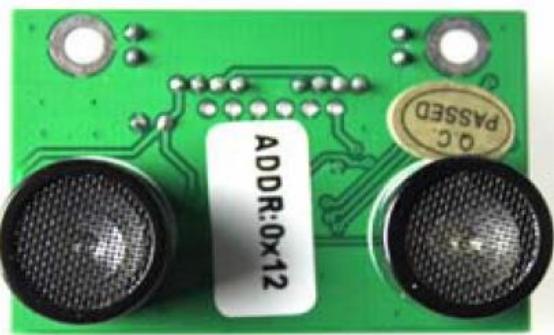


Abbildung 3-10: Ultraschallsensor Dual Ultra Sonic  
( Nexus Robot, 2012: S.23)

Die vier *Dual Ultrasonic Sensors* kommunizieren mittels des Bussystems RS485. RS485 ist eine Methode zur seriellen Kommunikation zwischen Computern oder anderen Geräten. Das System zeichnet sich durch die einfache Verkabelung, die lange Übertragungsdistanz und eine hohe Zuverlässigkeit aus. Zudem können mehrere Geräte an denselben Bus angeschlossen werden. Somit können mehrere Knoten miteinander verbunden werden. (vgl. Bies, 2017)

Das Kommunikationsnetzwerk besteht aus mehreren Transceivern. Diese sind über zwei verdrillte Drähte miteinander verbunden. Die Grundidee der Schnittstelle ist demnach eine differentielle Datenübertragung. Das bedeutet es wird ein Signal über zwei Drähte transportiert. Der erste Draht überträgt das Orginalsignal, während der zweite eine inverse Kopie transportiert. Diese Art der Datenübertragung gewährleistet eine hohe Beständigkeit gegen Störungen. Die Datenübertragung ist

über das Datenübertragungsprotokoll festgelegt. Über das Protokoll wird geregelt, wie die Datenpakete gesendet und empfangen werden. Bei einer RS485-Schnittstelle können nicht alle Geräte gleichzeitig Daten senden und empfangen. Dies würde zu einem Konflikt zwischen den Sendern führen und es kann zur Kollision von Datenpaketen kommen. Daher werden die Befehle von einem definierten Master gesendet (s. Abbildung 3-11). Die anderen Geräte werden als Slave bezeichnet. Diese erhalten die gesendeten Daten über RS485-Ports und können abhängig von den gesendeten Informationen auf der Leitung des Masters antworten. Weis, 2021

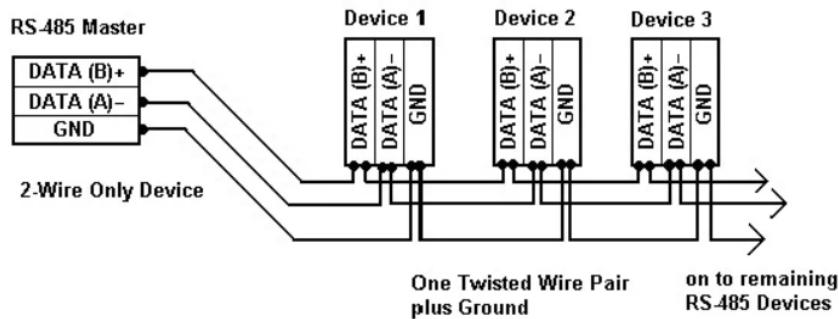


Abbildung 3-11: Datenübertragung vom Master zu den Slaves über RS485  
( Weis, 2021)

### 3.2.4 Arduino-Board

Das zentrale Element des Fahrzeugs ist das *Arduino-Board* bestehend aus dem Arduino-Board V1.1 mit ATmega328P-Mikrocontroller und einer Erweiterungsplatine. Es ist für die Verarbeitung von Signalen und Ansteuerung der Motoren sowie Sensoren zuständig. Im Folgenden werden die Hardwarekomponenten näher erläutert und anschließend der Anschluss der einzelnen Komponenten betrachtet.

Die Abbildung 3-12 stellt das Arduino Board V1.1 dar. Dieses zeigt alle Input/Output-Pins sowie die Anschlüsse auf dem Board. Es beinhaltet u.a.: (vgl. Nexus Robot, 2012: S.2f)

- Einen Motor Power Input Terminal (7V -12V)
- Einen Servo Power Input Terminal (4V – 7.2V)
- Zwei DC Motor Terminals
- Einen analogen Port mit acht analogen Input-Pins
- Einen digitalen I/O-Port mit 13 digitalen Pins, wovon die Pins 4,5,6,7 als Motor Control-Pins verwendet werden können
- Einen USB-Anschluss
- Einen Stromanschluss
- Einen APC220-Anschluss: Bietet die Möglichkeit ein Modul zur kabellosen Kommunikation anzuschließen
- APC220 *Wireless Selection Jumper*: Dieser wird benötigt um die kabellose Kommunikation über ein APC220-Modul zu ermöglichen. Wenn dieser entfernt wird, so kann keine Kommunikation darüber hergestellt werden.

Die digitalen Pins können als Eingänge und als Ausgänge verwendet werden. Sie operieren auf einer Spannung von 5V. Einige dieser Pins haben eine spezielle Funktion. Pin 0 (RX) und Pin 1 (TX) werden genutzt, um serielle Daten zu erhalten (RX) oder zu übertragen (TX). Die beiden Pins des Boards sind mit USB-Anschluss verbunden. Zudem gibt es zwei externe *Interrupt-Pins* Pin 2 und Pin 3. Diese können für *Hardware-Interrupts* genutzt werden (vgl. Kapitel 3.2.5 *PinChange*). Die Pins 3, 5, 6, 9, 10 und 11 bieten ein 8Bit-PWM-Ausgang. Zudem existiert eine integrierte LED, welche mit dem Pin 13 verbunden ist. Wenn der Pin den Wert HIGH annimmt, so ist die LED an. Wenn der Pin den Wert LOW besitzt, ist sie aus.

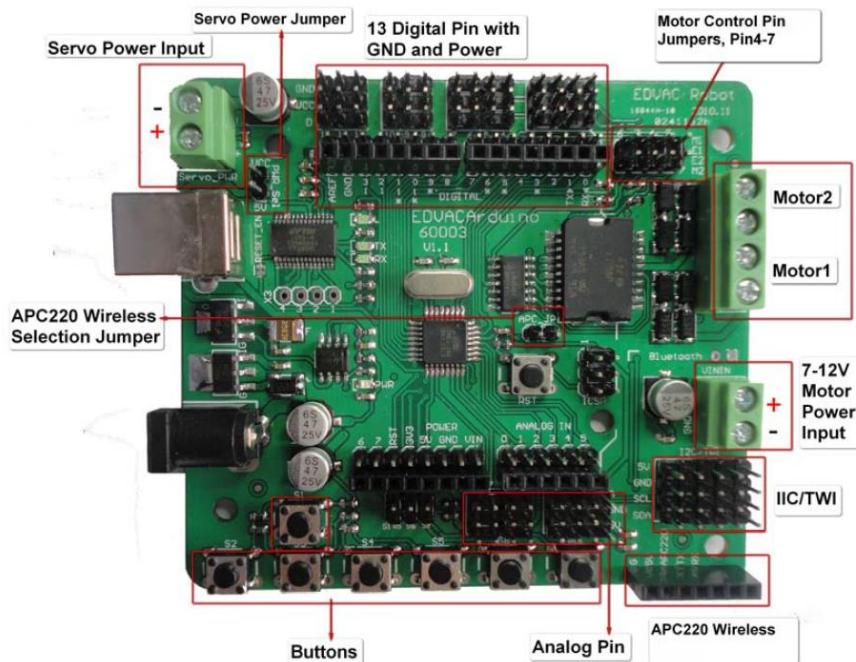


Abbildung 3-12: Arduino-Board V1.1  
( Nexus Robot, 2012: S.2)

Die Erweiterungsplatine *Arduino IO Expansion Board* kann für eine serielle Kommunikation über RS485 sowie zum Anschluss zweier weiterer Motoren genutzt werden (s. Abbildung 3-13). Das Erweiterungsboard lässt sich einfach auf das *Arduino-Board V1.1* aufstecken. Das *Board* stellt weitere digitale und analoge Pins zur Verfügung, wovon auch einige PWM unterstützen. Auf dem Board sind u.a. folgende Anschlüsse integriert: Nexus Robot, 2012: S. 4f

- Einen Anschluss für XBee/Bluetooth Bee
- Einen RS485-Ausgang, um RS485-Geräte anzuschließen
- Einen Anschluss für ein APC220-Modul

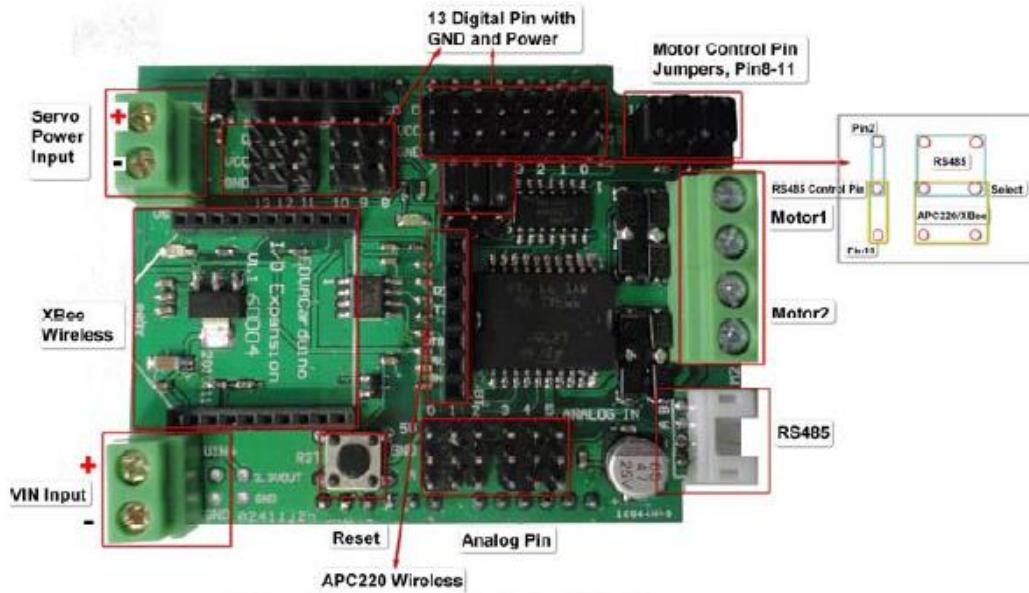


Abbildung 3-13: Arduino IO Expansion Board  
( Nexus Robot, 2012: S.4)

An den *Arduino-Boards* sind die Motoren, Sensoren und Batterie angeschlossen (s. Abbildung 3-14). Die Motoren werden jeweils an einen DC Motor Terminals angeschlossen. Motor 1 und Motor 2 werden an der *Arduino Board V1.1* angeschlossen, während Motor 3 und Motor 4 mit der Erweiterungsplatine verbunden werden. Die Motoren werden jeweils über vier weitere Pins mit dem Board verbunden, wovon 2 Pins für den Encoder sind, einer für das PWM-Signal und ein weiterer für die Richtungsangabe (DIR). Die Encoder-Pins unterscheiden sich in Phase A und Phase B. Der Pin, welcher mit Phase A des Encoders verbunden ist, dient zur Eingabe von Befehlen an den Motor. Der Pin, der mit Phase B des Encoders verbunden ist, dient zur Ausgabe des Signals des Motors. Über den Anschluss *Motor Power Input* wird eine 12V-Batterie mit dem Board verbunden. Dadurch kann das Fahrzeug ohne Netzanschluss fahren. Die Anbindung der Sensoren erfolgt an den RS485-Ausgang. Hieran wird lediglich der erste Sensor angebunden. Die weiteren Sensoren werden anschließend in Reihe geschaltet.

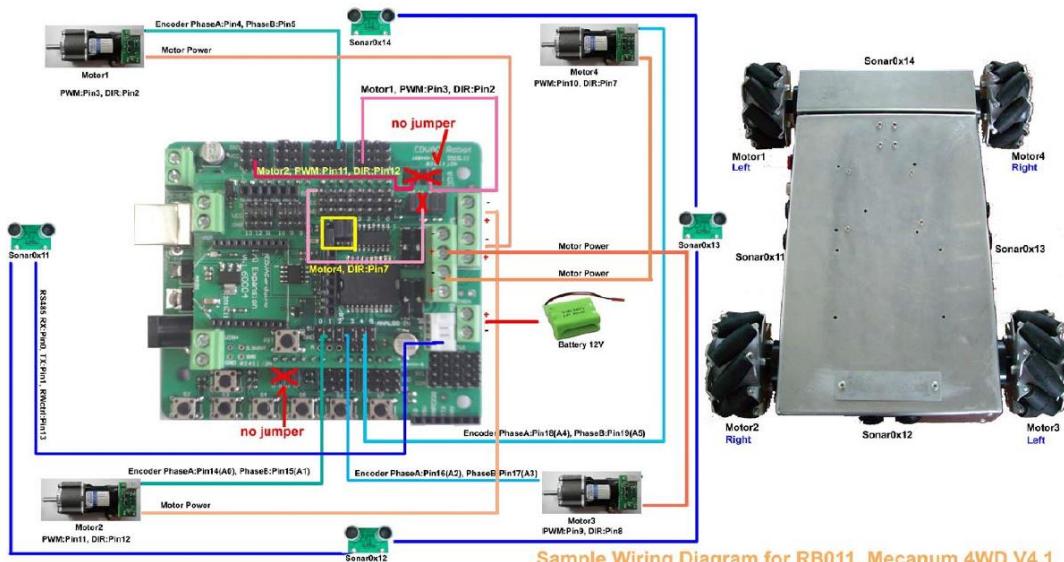


Abbildung 3-14: Anschluss der Komponenten an Arduino-Board  
( Nexus Robot, 2012: S.114)

Die Tabelle 3-1 listet übersichtshalber die Pin-Belegung der einzelnen Pins auf. Diese enthält zum einen die jeweilige Pin-Nummer, zum anderen wird angegeben, welche Funktion der jeweilige Pin bezogen auf die angeschlossene Komponente hat. Zuletzt wird dem Pin die jeweilige Komponente zugeordnet.

Tabelle 3-1: Pin-Belegung

Pin Nr.	Funktion	Komponente
Pin 0	RX	Rs485 Sonar
Pin 1	TX	Rs485 Sonar
Pin 2	Dir	Motor1
Pin 3	PWM	Motor1
Pin 4	Encoder Phase A	Motor1
Pin 5	Encoder Phase B	Motor1
Pin 6	-	-
Pin 7	Dir	Motor4
Pin 8	Dir	Motor3
Pin 9	PWM	Motor3
Pin 10	PWM	Motor4
Pin 11	PWM	Motor2
Pin 12	Dir	Motor2
Pin 13	RWctrl	Sonar
Pin 14	Encoder Phase A	Motor2
Pin 15	Encoder Phase B	Motor2
Pin 16	Encoder Phase A	Motor3
Pin 17	Encoder Phase B	Motor3
Pin 18	Encoder Phase A	Motor4
Pin 19	Encoder Phase B	Motor4

### 3.2.5 Bibliotheken und Demoskript

Die *Arduino*-Umgebung kann durch die Verwendung von Bibliotheken erweitert werden. Diese Bibliotheken bieten zusätzliche Funktionen für die Verwendung in *Sketches*, wie z.B. das Arbeiten mit bestimmter Hardware oder das Bearbeiten von Daten. Eine Vielzahl von Standardbibliotheken sind bereits in der *Arduino IDE* hinterlegt und können importiert werden. Allerdings können auch eigene Bibliotheken erstellt und heruntergeladen werden. (vgl. Arduino, 2023)

Die Bibliotheken werden oftmals in eine Header-Datei *header.h* und eine Quelldatei *Quelldatei.cpp* unterteilt. In der Header-Datei werden Deklarationen bspw. von Datentypen, Strukturen, Funktionen oder Klassen vorgenommen. Sie gibt somit an, dass diese existieren, jedoch nicht wie sie funktionieren. In der Quelldatei werden diese Deklarationen verwendet und anschließend implementiert. Zum Beispiel wird hier definiert, wie eine deklarierte Funktion arbeitet bzw. funktioniert. Das Aufteilen in zwei separate Dateien hat den Vorteil, dass die Kompilierungszeit erheblich reduziert wird, da die Quelldatei bereits in kompilierter Form auf der Plattform vorliegt und der Compiler lediglich die Deklaration der Header-Datei abruft. Eine Bibliothek bzw. eine Header-Datei wird wiederum aufgerufen, indem diese durch *#include* im *Sketch* hinzugefügt wird. Die Verwendung und der Aufbau der Bibliotheken wird beim Betrachten der folgenden Bibliotheken und des Demoskriptes verständlicher.

Die Firma Nexus bietet neben der Hardware des FTF auch Bibliotheken und ein Demoskript zum Bewegen des Fahrzeuges an. Das Demoskript integriert u.a. die Bibliotheken *MotorWheel.h*, *Omni4WD.h*, *PID\_Beta6.h*, *sonar.h* und *PinChangeInt.h*. Die Funktionsweise und Verwendung dieser Bibliotheken werden in den jeweiligen Abschnitten behandelt.

## Pin Change Interrupt

Bei dieser Bibliothek ist vor allem wichtig zu verstehen, was ein *Pin Change Interrupt* ist und wozu dieser gebraucht wird. Daher wird das Skript selbst nicht näher betrachtet, sondern lediglich seine Funktionsweise erklärt.

Mikrocontroller haben oftmals eine Vielzahl an Aufgaben, die sie zu bewältigen haben und sind daher ziemlich beschäftigt. Dennoch müssen externe Ereignisse und interne *Timing-Ereignisse* kontrolliert ausgeführt werden. Dafür eignet sich die Verwendung von *Interrupts*. *Interrupts* sind eine Methode, um den aktuellen Programmausführung zu unterbrechen. Das Programm wird angehalten und seine Daten beiseitegelegt, damit diese später an derselben Stelle fortgesetzt werden können. Dann wird der Programmcode ausgeführt, welcher sich auf den *Interrupt* bezieht, die sogenannte *Interrupt Service Routine ISR* (s. Abbildung 3-15). Sobald diese beendet ist, wird das Programm dort fortgesetzt, wo es aufgehört hat. In der Regel werden zwei Arten von *Interrupt*-Funktionen unterschieden. Es gibt sowohl die *Hardware-Interrupts* als auch die *Software-Interrupts*. Die *Hardware-Interrupts* werden durch externe Signale ausgelöst. Die *Software-Interrupts* werden durch interne Signale wie durch *Timer* oder softwarebezogene Ereignisse gesteuert. (vgl. DroneBot-Workshop, 2022; mwwalk, 2014)

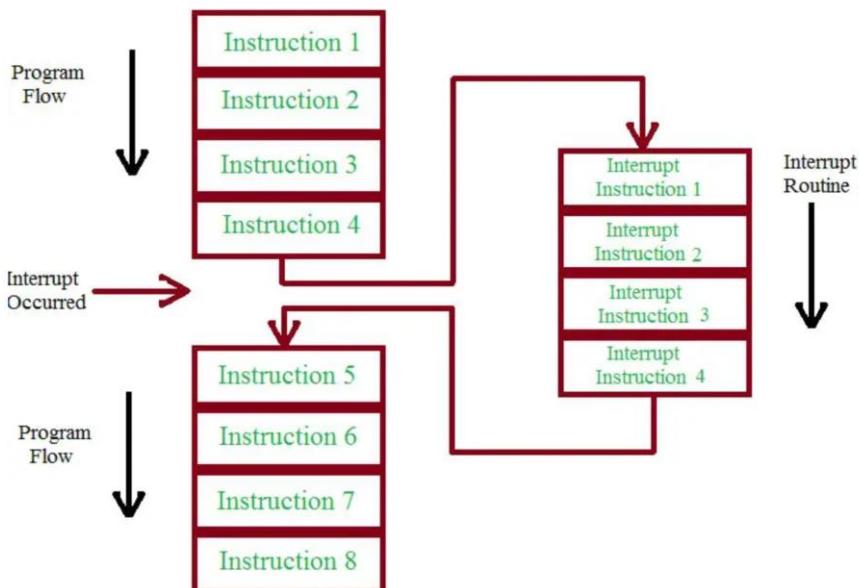


Abbildung 3-15: Interrupt Service Routine  
( Carrasco, 2021)

Des Weiteren werden die Interrupts in drei Arten untergliedert:

1. *Hardware-Interrupts* - Externe *Interrupt-Signale* an bestimmten Pins
2. *Pin Change Interrupts* - Externe *Interrupts* an jedem Pin, gruppiert in Ports
3. *Timer-Interrupts* – Interne *Timer-generierte Interrupts*, die in der Software manipuliert werden

Dabei zählen die *Pin Change Interrupts* und die *Timer-Interrupts* zu der übergeordneten Kategorie der *Software-Interrupts*. Wie der Name schon verrät, beschreibt die *Pin-Change-Interrupt-Bibliothek* die zweite Art von *Interrupts*, die „*PinChange Interrupts*“. Da *Hardware-Interrupt* lediglich auf wenige Pins beschränkt sind und oftmals mehr Pins Verwendung finden müssen, muss die Anzahl an Pins zur Benutzung von *Interrupts* erweitert werden. In der *PinChange-Interrupt-Bibliothek* werden dementsprechend Funktionen festgelegt, die angeben, wie ein *Pin-Change-Interrupt* ausgelöst wird, an welchen Pins er ausgelöst wird, wie er zurückgenommen wird etc., sodass sich dann das *Arduino*-Programm bzw. andere Bibliotheken sich der Pin-Change-Interrupt-Funktionen bedienen kann.

## PID Beta6

Die *PID Beta6* implementiert die Funktionen für einen PID-Regler. Ähnlich wie bei der *PinChange-Interrupt*-Bibliothek sollen an dieser Stelle allerdings die Funktionen nicht näher betrachtet werden, sondern zunächst im Allgemeinen erläutert, was eine PID-Regelung ist.

Ein PID-Regler wird dazu genutzt eine Differenz zwischen einem gemessenen Wert und Soll-Wert auszugleichen, sodass keine bleibende Regelabweichung vorhanden ist. In Abbildung 3-16 ist ein einfacher Regelkreis dargestellt. Dieser besteht aus einem Regler und einer Regelstrecke. Der Regler ist in diesem Fall ein PID-Regler und die Regelstrecke ist das Fahrzeug bzw. die einzelnen Motoren des Fahrzeugs. Zunächst wird ein Sollwert z.B. eine Sollgeschwindigkeit vorgegeben. Diese wird verglichen mit dem Ist-Wert. Aus diesen beiden Werten wird eine Regelabweichung bestimmt, die auf den Regler gegeben wird. Der Regler gibt als Ausgang eine Stellgröße auf die Regelstrecke, damit die Abweichung zwischen Soll und Ist verringert wird. Am Ende wird letztendlich wieder ein Ist-Wert ermittelt und zurückgeführt und die Regelung beginnt von vorne. (vgl. RN-Wissen, 2023)

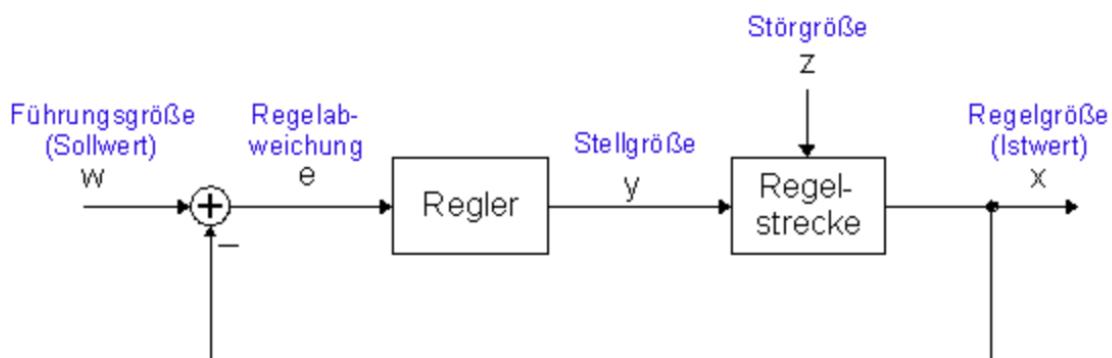


Abbildung 3-16: Vereinfachter Regelkreis  
(Tobola, 2014)

Die Problematik wird am besten anhand eines Beispiels verdeutlicht. Es soll beispielsweise eine Geschwindigkeit von 50km/h erzielt werden. Das bedeutet, der Sollwert beträgt 50km/h. Allerdings wird ein Ist-Wert von 45km/h gemessen, sodass eine Regelabweichung von 5km/h entsteht. Mit Hilfe der Regelparameter wird dann eine Stellgröße ermittelt. Diese kann z.B. den Wert 7km/h annehmen und stellt die notwendige Änderung dar. Die Stellgröße wird auf die Regelstrecke gegeben und so verarbeitet, dass sich wieder der Ist-Wert der Geschwindigkeit ergibt. Diese kann z.B. einen Wert von 51km/h annehmen.

Der Regler soll dabei möglichst schnell und mit kleinem bzw. ohne Überschwinger auf eine Abweichung reagieren. Der PID-Regler besteht aus drei Anteilen dem Proportionalanteil (P-Anteil), dem Integralanteil (I-Anteil) und dem Differentialanteil (D-Anteil). Jedes dieser Anteile hat unterschiedliche Funktionen. Der P-Regler wirkt proportional zur Regelabweichung. Er reagiert unmittelbar auf jede Veränderung der Regelabweichung, allerdings kann die Regelabweichung nicht vollständig beseitigt werden. Der I-Regler reagiert hingegen sehr langsam, kann dafür aber die Ist-Größe exakt auf die Soll-Größe einstellen. Der D-Regler reagiert auf die zeitliche Änderung der Regelabweichung. Bei konstanten Abweichungen bewirken sie daher keinen Stelleingriff. Der alleinige Einsatz eines D-Regler ist ungeeignet. In Kombination als PID-Regler ermöglicht der D-Anteil ein schnelles, überschwingfreies Anfahren an den Sollwert. Über die Regelparameter kann das Regelverhalten eingestellt werden, wobei jeder Regleranteil durch den jeweiligen Regelparameter angepasst wird. (vgl. RN-Wissen, 2023)

## Sonar

Die Bibliothek *sonar* dient zur Beschreibung der Funktionsweise des Kommunikationsprotokolls der Ultraschallsensoren. In Kapitel 3.2.3 wurde bereits die Funktionsweise der Ultraschallsensoren und deren Kommunikationsweg beschreiben. Die Sensoren kommunizieren über eine RS485-Schnittstelle miteinander. Mit Hilfe eines Kommunikationsprotokolls wird beispielsweise festgelegt, wie die Adresse eines Sensors aussieht oder wie eine Messung ausgelöst wird. Die Beschreibung dessen erfolgt innerhalb der Bibliothek. Hier werden verschiedene Kommandos definiert. Es werden u.a. folgenden Kommandos bzw. Funktionen beschrieben: (vgl. Nexus Robot, 2012: S.27f)

- *Set Device Address (SONAR::setAddr(addr))*: Hier wird einem Sensor eine bestimmte Adresse zugeordnet
- *Trigger measurement (SONAR::trigger())*: Löst einen Trigger aus, durch welchen eine Distanz gemessen wird. Diese Funktion gibt allerdings keine Daten zurück.
- *Get distance (SONAR::getDist())*: Diese Funktion gibt den gemessenen Abstandswert zurück.

## MotorWheel und Omni4WD

Die beiden zentralen Bibliotheken sind die Bibliothek *MotorWheel* und die Bibliothek *Omni4WD*. Sie greifen zum einen die Funktionen der anderen Bibliotheken auf und zum anderen beschreiben sie Funktionen zur Ansteuerung und Regelung der Motoren. Wie die anderen Bibliotheken sind auch diese beiden untergliedert in *MotorWheel.h* und *Motorwheel.cpp* bzw. *Omni4WD.h* und *Omni4WD.cpp*. Während in der Header-Datei der meisten Bibliotheken größtenteils die Funktionen deklariert werden, werden vor allem in der Headerdatei *MotorWheel.h* ebenfalls Parameter für Chassis, Räder, Regelung, Motor etc. definiert. In Programmcode 3-1 ist beispielhaft die Definitionen der Parameter für Motor, PWM-Signal und maximale Drehzahl angegeben. Da es sich bei den Motoren, um Faulhaber-Motoren handelt, werden dementsprechend die Parameter ausgewählt. Zudem wird eine maximales PWM von 255 und eine maximale Drehzahl von 8000rpm angegeben. Diese Werte finden sich im Datenblatt des Fahrzeugs wieder.<sup>1</sup>

```
/*for arduino*/
#ifndef _NAMIKI_MOTOR
#define MAX_PWM 255
#endif

#define TRIGGER CHANGE
#define CPR 4 // Namiki motor
#define DIR_INVERSE
#define REDUCTION_RATIO 80
#endif

#define TRIGGER RISING
#define CPR 12 // Faulhaber motor
#define DIR_INVERSE !
#define REDUCTION_RATIO 64
#endif

#define MAX_SPEEDRPM 8000
```

Programmcode 3-1: Definieren von Fahrzeug Parametern

---

<sup>1</sup> Das Datenblatt befindet sich im Anhang A

In der Quelldatei *MotorWheel.h* werden die Funktionen für die einzelnen Motoren sowie die *Interrupt*- und Regelungsfunktionen beschrieben. Diese Funktionen werden in der Quelldatei *Omni4WD.cpp* aufgegriffen. Bei dieser Bibliothek wird eine Klasse erzeugt, welche die vier Motoren miteinander verknüpft, sodass Funktionen eingeführt werden können, welche sich auf alle Motoren gleichermaßen beziehen. In Folgenden sollen zwei dieser Funktionen kurz erläutert werden.

Im Programmcode 3-2 wird die Funktion *Omni4WD::setCarMove(...)* beschrieben. Bei dieser Funktion werden drei Werte angegeben. Zum einen eine Geschwindigkeit *speedMMPS*, zum anderen einen Winkel *rad* und zuletzt eine Winkelgeschwindigkeit *omega*. Die Geschwindigkeit gibt einen Wert zwischen 0 und 255 an, mithilfe des Winkels wird die Fahrtrichtung festgelegt und über die Winkelgeschwindigkeit wird eine Rotation um die Fahrzeugachse ermöglicht. Für jedes einzelne Rad werden dann die jeweiligen Anteile ausgerechnet.

```
int Omni4WD::setCarMove(int speedMMPS, float rad, float omega) {
    wheelULSetSpeedMMPS(speedMMPS*sin(rad)+speedMMPS*cos(rad)-omega*WHEELSPAN);
    wheelLLSetSpeedMMPS(speedMMPS*sin(rad)-speedMMPS*cos(rad)-omega*WHEELSPAN);
    wheelLRSetSpeedMMPS(-(speedMMPS*sin(rad)+speedMMPS*cos(rad)+omega*WHEELSPAN));
    wheelURSetSpeedMMPS(-(speedMMPS*sin(rad)-speedMMPS*cos(rad)+omega*WHEELSPAN));

    return getCarSpeedMMPS();
}
```

Programmcode 3-2: Definition der Funktion "SetCarMove"

Die Funktion *setCarMove()* wird anschließend aufgegriffen, um die Bewegungs- bzw. Fahrtrichtungen zu definieren. So werden Funktionen für die Vorausfahrt, Rückwärtsfahrt, Seitwärtsfahrt nach links und rechts, Diagonalfahrt (oben links, unten links, oben rechts, unten rechts), Rotation um die Fahrzeugachse links- und rechtsherum sowie zum Stoppen der Fahrtbewegung festgelegt. Der Programmcode 3-3 beschreibt die Vorausfahrt. Dafür wird ein Status auf Voraus (engl. *Advance*) gesetzt. Anschließend wird über *setCarMove* die Fahrtrichtung und Geschwindigkeit festgelegt. Damit das Fahrzeug vorausfährt wird ein Winkel von  $\pi/2$  benötigt. In Abbildung 3-17 ist dargestellt, bei welchem Winkel, das Fahrzeug in welche Richtung fährt.

```
int Omni4WD::setCarAdvance(int speedMMPS) {
    setCarStat(STAT_ADVANCE);
    //wheelULSetSpeedMMPS(speedMMPS,DIR_ADVANCE);
    //wheelLLSetSpeedMMPS(speedMMPS,DIR_ADVANCE);
    //wheelLRSetSpeedMMPS(speedMMPS,DIR_BACKOFF);
    //wheelURSetSpeedMMPS(speedMMPS,DIR_BACKOFF);
    //return wheelULGetSpeedMMPS();
    return setCarMove(speedMMPS,PI/2,0);
}
```

Programmcode 3-3: Definition der Funktion  
"setCarAdvance"

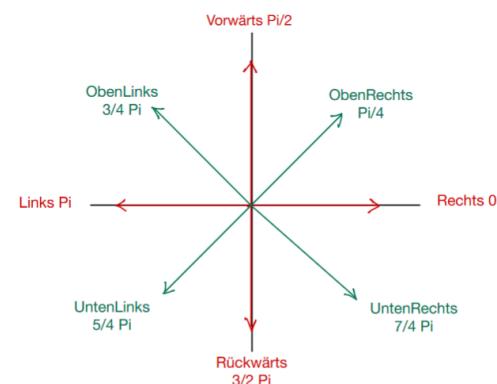


Abbildung 3-17: Fahrtrichtung abhängig von den Winkeln

## Demoskript

Die Firma Nexus Robot stellt für das Fahrzeug ein Demoskript zur Verfügung. Ausschnitte von diesem sollen an dieser Stelle kurz erläutert werden.<sup>2</sup>

Der erste Ausschnitt ist unter Programmcode 3-4. zu finden. Dieser zeigt, wie ein Objekt für Motor 4 erstellt wird. Zunächst wird die *Interrupt*-Funktion *irqISR(irq4, isr4)* für den Motor 4 erstellt. Dieser arbeitet auf Basis von Pulsen, welche vom Encoder generiert werden. Als nächstes wird dann das Objekt *MotorWheel wheel4(10,7,18,19&irq4)* für den Motor selbst erzeugt. In Klammern befinden sich die Angaben für die Pins. Pin 10 ist für PWM, Pin 7 der Richtungspin, Pin 18 steht für Encoder Phase A und Pin 19 für Encoder Phase B. Die anderen 3 Motoren werden analog dazu erstellt. Als nächstes wird ein Objekt *Omni4WD Omni(&wheel1, &wheel2, &wheel3, &wheel4)* hervorgerufen. Dieses Objekt bindet alle Motoren ein, sodass es möglich wird Funktionen zu erstellen bzw. aufzurufen, welche alle vier Motoren gleichzeitig betreffen. Im nächstes Schritt werden den Ultraschallsensoren ihre Adresse zugeordnet und ein *Buffer distBuf[4]*, in welchem die gemessenen Distanzen der Sensoren gespeichert werden können, erzeugt. Im letzten Teil des Programmcodeausschnittes wird die Funktion *sonarsUpdate()* definiert. Bei dieser wird zunächst eine Laufvariable *sonarCurr* festgelegt und auf den Wert 1 gesetzt. Sobald die Laufvariable den Wert 4 annimmt, wird sie immer wieder auf den Wert eins zurückgesetzt, ansonsten wird sie um eins hochgesetzt. Wenn die Laufvariable *sonarcurr==1* ist, so wird die Distanz des Ultraschallsensors mit der Adresse 0x12 in den *Buffer* an Position 1 geschrieben. Über *showDat()* kann diese auf einem seriellen Monitor ausgegeben werden. Anschließend wird ein *Trigger* gesetzt, um eine neue Messung auszulösen. Wenn die Laufvariable *sonarcurr==2* wird, dann das gleiche für den Sensor *sonar13* durchgeführt, wobei seine Distanzwerte an die Position 2 des *Buffer* geschrieben werden. Für die übrigen zwei Sensoren gilt dasselbe Prinzip. Zuletzt wird der Wert der Laufvariable *sonarcurr* zurückgegeben.

```

43  irqISR(irq4,isr4);
44  MotorWheel wheel4(10,7,18,19,&irq4);
45
46  Omni4WD Omni(&wheel1,&wheel2,&wheel3,&wheel4); //Alle Motoren inbegriffen
47
48  SONAR sonar11(0x11),sonar12(0x12),sonar13(0x13),sonar14(0x14);
49  unsigned short distBuf[4];
50
51  unsigned char sonarsUpdate() {
52      static unsigned char sonarCurr = 1;
53      if(sonarCurr==4) sonarCurr=1;
54      else ++sonarCurr;
55      if(sonarCurr==1) {
56          distBuf[1]=sonar12.getDist();
57          sonar12.showDat();
58          sonar12.trigger();
59      } else if(sonarCurr==2) {
60          distBuf[2]=sonar13.getDist();
61          sonar13.showDat();
62          sonar13.trigger();
63      } else if(sonarCurr==3){
64          distBuf[3]=sonar14.getDist();
65          sonar14.showDat();
66          sonar14.trigger();
67      } else {
68          distBuf[0]=sonar11.getDist();
69          sonar11.showDat();
70          sonar11.trigger();
71      }
72
73      return sonarCurr;
74  }

```

Programmcode 3-4: Deklarierung der Motoren und Sensoren sowie Funktion *sonarsUpdate()*

<sup>2</sup> Das vollständige Demoskript befindet sich im Anhang B

Der nächste Programmcodeausschnitt beschreibt, wie eine Funktion zur Vorausfahrt erstellt wird. Zunächst wird der Status überprüft. Entspricht der Status nicht dem Status „Vorausfahrt“, sondern z.B. „Rückwärtsfahrt“ so werden die Motoren langsam gestoppt. Ansonsten wird über die Funktion *Omni.setCarAdvance* die Richtung des Fahrzeuges auf „Voraus“ gestellt und über die Funktion *Omni.setCarSpeedMMPS(speedMMPS,300)* die Geschwindigkeit und die Dauer eingestellt, wie lange das Fahrzeug in diese Richtung fahren soll, hier 300ms. Analog dazu werden Funktionen für die anderen Fahrtrichtungen bzw. Drehrichtungen erzeugt.

```

76 void goAhead(unsigned int speedMMPS){
77     if(Omni.getCarStat()!=Omni4WD::STAT_ADVANCE) Omni.setCarSlow2Stop(300);
78     Omni.setCarAdvance(0);
79     Omni.setCarSpeedMMPS(speedMMPS, 300);
80 }
```

Programmcode 3-5: Funktion „goAhead“

Der nächste Ausschnitt des Demoskriptes bringt ein Array und eine Funktion hervor. Mit Hilfe dieser wird die Fahrtrichtung des Fahrzeuges in Abhängigkeit der Sensoren bestimmt. Das Array *motion[16]* enthält diverse Funktionen für die Fahrtrichtungen. Wird beispielsweise die Position vier des Arrays aufgerufen, so soll sich das Fahrzeug nach links (engl. *turnleft*) bewegen. Welches dieser Positionen aufgerufen wird, wird über die Funktion *demoWithSensors(speedMMPS,distance)* ermittelt. Innerhalb der Funktion existieren zwei if-Schleifen. Die erste sorgt, dass alle 80ms die Funktion *SonarUpdate* aufgerufen wird. Diese Funktion (s. oben) gibt einen Wert zwischen 0 und 4 zurück. Sobald der Wert *sonarcurrent==4* entspricht wird die zweite if-Funktion aufgerufen. Hier wird für jeden Sensor jeweils ermittelt, ob die im *Buffer* gespeicherten Distanzen kleiner als eine vorgegebene Distanz sind. Ist dies für einen Sensor der Fall wird bei einem Byte eine 1 gesetzt. Je nach Sensor wird diese an eine andere Stelle im Byte positioniert, sodass die Variable *bitmap* letztlich einen Wert zwischen Null und 15 annehmen kann. Das *bitmap* gibt anschließend an, welche Position im Array *motion[bitmap]* ausgewählt wird, wodurch wiederum die Fahrtrichtung festgelegt wird. Zuletzt wird die Funktion *Omni.PIDRegulate()* aufgerufen, welche das Regeln der Bewegung ermöglicht.

```

120 void(*motion[16])(unsigned int speedMMPS) = {goAhead, turnRight, goAhead, turnRight,
121     turnLeft, goAhead, turnLeft, goAhead,
122     rotateRight, rotateRight, turnRight, turnRight,
123     rotateLeft, backOff, turnLeft, allStop};
124
125 unsigned long currMillis=0;
126 void demoWithSensors(unsigned int speedMMPS,unsigned int distance) {
127     unsigned char sonarcurrent = 0;
128     if(millis()-currMillis>SONAR::duration + 20) {
129         currMillis=millis();
130         sonarcurrent = sonarsUpdate();
131     }
132
133     if(sonarcurrent == 4){
134         unsigned char bitmap = (distBuf[0] < distance); //right
135         bitmap |= (distBuf[1] < distance) << 1; // back
136         bitmap |= (distBuf[2] < distance) << 2; // left
137         bitmap |= (distBuf[3] < distance) << 3; // front
138
139         (*motion[bitmap])(speedMMPS);
140         Serial.println(bitmap);
141     }
142
143     Omni.PIDRegulate();
144 }
```

Programmcode 3-6: Array *motion* und Funktion *demoWithSensors*

Der letzte Teil des Quellcode beschreibt die Funktionen `void setup()` und `void loop()`. Im `void setup()` wird zunächst die Frequenz des PWM-Signals eingestellt (vgl. Kapitel 3.2.2) und anschließend über `SONAR::init` der Kontrollpin `pinCtrl=13` festgelegt, über welchen die Sensoren aufgerufen werden. Danach wird über die Funktion `Omni.PIDEnable` der PID-Regler aktiviert und dessen Parameter festgelegt. Der Parameter für den P-Anteil beträgt 0,35, für den I-Anteil 0,02 und für den D-Anteil 0. Das bedeutet, dass der D-Anteil wegfällt und die Regelung lediglich über einen PI-Regler erfolgt. Letztlich wird im `void loop()` nur noch die Funktion `demoWithSensors` aufgerufen und die Fahrgeschwindigkeit (160mm/s) und Distanz, bei der ein Sensor als „ausgelöst“ betrachtet wird, angegeben.

```
146 void setup() {
147     delay(2000);
148     TCCR1B=TCCR1B&0xf8|0x01;      // Pin9,Pin10 PWM 31250Hz
149     TCCR2B=TCCR2B&0xf8|0x01;      // Pin3,Pin11 PWM 31250Hz
150
151     SONAR::init(13);
152     Omni.PIDEnable(0.35,0.02,0,10);
153 }
154
155 void loop() {
156     demoWithSensors(160,20);
157 }
```

Programmcode 3-7: Funktion `void setup()` und `void loop()`

### 3.3 Trackingsystem OptiTrack

Trackingsysteme werden zur Lokalisierung und Nachverfolgung von physischen Objekten verwendet. Für diese Masterarbeit wird eine Trackingkamera von OptiTrack S250e sowie ein dazugehöriges Anwendungsprogramm zum Auslesen der Trackingdaten zur Verfügung gestellt (s. Abbildung 3-18). In diesem Kapitel wird zunächst die Funktionsweise der Kamera selbst beschrieben. Auf das Anwendungsprogramm wird in Kapitel 5.3.1. eingegangen.

OptiTrack-Kameras verwenden Infrarot-LEDs zur Nachverfolgung passiver runder Marker. Die passiven Marker sind mit einem reflektierenden Material beschichtet. Sie reflektieren das Licht der Infrarot-LEDs größtenteils zurück, wodurch die Kamera die Position des Markers detektiert wird. Mit Hilfe einer Kamera kann eine 2D-Position eines Markers ermittelt werden. Werden jedoch mehrere Kameras eingesetzt, so kann durch Überlappung der Bilder eine dreidimensionale Position bestimmt werden. Das Trackingsystem verfügt über eine sehr hohe Genauigkeit, sodass alle Arten von Bewegungen erfasst werden können. Wenn mindestens zwei Markierungen sichtbar sind, kann die Orientierung eines Objektes angegeben werden. Sobald ein Marker allerdings überdeckt wird, kann die Trackingkamera dessen Position nicht mehr erfassen. Mit der Kamera können über 80 Objekte gleichzeitig verfolgt werden. Zudem ist bei den meisten Anwendungen eine Datenverarbeitung in Echtzeit möglich. (vgl. NatrualPoint Inc., 2023)



Abbildung 3-18: OptiTrack S250e  
(NatrualPoint Inc., 2012)

## 4 Product Backlog für die Bewegungssteuerung

Nachdem die Ausgangssituation und die Funktionsweise des Fahrzeugs und Trackingsystems erläutert wurde, sollen nun die Anforderungen an die Bewegungssteuerung in einem *Product Backlog* gelistet werden. Das *Product Backlog* ist in Tabelle 4-1 und Tabelle 4-2 enthalten. Es besteht aus den Anforderungen an die Steuerung sowie der Priorisierung der Anforderungen. In diesem Kapitel wird nicht im Detail auf die einzelnen Anforderungen eingegangen, sondern lediglich die Priorisierung und Abarbeitung erklärt.

Das *Product Backlog* ist in vier Prioritätsklassen untergliedert, wobei die Priorität 4 die höchste Priorität darstellt und die Priorität 1 die niedrigste Priorität. Mithilfe der Prioritäten werden die Anforderungen klassifiziert und den jeweiligen *Sprints* zugewiesen. All jene Anforderung, die eine Priorität von vier aufweisen, werden im ersten *Sprint* umgesetzt. Alle Anforderungen mit Priorität 3 im zweiten *Sprint* usw. Es ergeben sich damit zunächst vier *Sprints*, wobei lediglich drei dieser Sprints Bestandteil dieser Arbeit darstellen. Die Anforderungen enthalten zudem keine technischen, räumlichen oder sonstigen Spezifikationen. Sie stellen nur die groben Anforderungen bzw. Aufgaben der Bewegungssteuerung dar.

Aus der Prioritätsklasse 4 ergeben sich Anforderungen bezüglich der Schnittstellen und der Kommunikation. Zudem sind Anforderungen an die Art der Bewegungen gestellt. Die Prioritätsklasse 3 enthält Anforderungen zur gezielten Kontrolle und Regelung der Fahrbewegung entlang einer Punkt-zu-Punkt-Verbindung. Die Anforderungen innerhalb der Prioritätsklasse 2 beziehen sich auf die räumliche Betrachtung der Fahrbewegung bzw. der Bewegung entlang eines Pfades. Die letzte Prioritätsklasse beschreibt alle Anforderungen für eine optimierte Bewegungssteuerung beispielsweise das Fahren entlang von Kurven oder in dynamischen Umgebungen.

Tabelle 4-1: *Product Backlog* Teil 1

Product Backlog		
Nr.	Anforderung	Priorität
1	Datenübertragung und Kommunikation zwischen Trackingsystem und FTF	4
2	Implementieren der Hardware	4
3	Kalibrieren des Trackingsystems	4
4	Ansteuern von FTF über eine kabellose Schnittstelle	4
5	Überlagerung von Bewegungen	3
6	Fahren von geraden Strecken	4
7	Rotieren um die eigene Achse des FTF	4
8	Fahren von Kurven	1
9	Direktes Fahren zu einem vorgegebenen Punkten	3
10	Lageregelung des FTF	3
11	Stetiger Geschwindigkeitsverlauf	3
12	Interpolation einer Trajektorie	3
13	Synchrone Bahoplanung	3
14	Fahren entlang eines vorgegebenen Pfades über mehrere Punkte	2
15	Autonomes Suchen eines Pfades zu einem vorgegebenen Punkt	2
16	Fahren in einer statischen Umgebung	2

Tabelle 4-2: *Product Backlog Teil 2*

Product Backlog		
Nr.	Anforderung	Priorität
17	Kollisionsfreies Fahren in einer dynamischen Umgebung	1
18	Dynamische Hindernisse detektieren	1
19	Kompensation von Bewegungsungenauigkeiten	2
20	Erstellen von virtuellen Hindernissen	2
21	Kollisionsfreies Fahren im freien Konfigurationsraum	2
22	Definierbarer Arbeits- und Konfigurationsraum	2
23	User Interface	1
24	Visualisierung/Simulation des Fahrweges des FTS im Konfigurationsraum	1
25	Verschleifen von Punkten entlang des Fahrweges	1
26	Stetiger und Ruckfreier Geschwindigkeitsverlauf	1

4	Höchste Priorität
3	Hohe Priorität
2	Mittlere Priorität
1	Niedrige Priorität

## 5 Schnittstellen zur Datenübertragung

Es wurde bereits die Kommunikation bzw. die Datenübertragung zwischen den einzelnen Komponenten des Fahrzeuges betrachtet. Dieses Kapitel beschäftigt sich nun mit einer Ebene darüber; Der Kommunikation zwischen dem Fahrzeug selbst und dem Trackingsystem. Nur durch die Verknüpfung dieser beiden einzelnen Systeme wird ein fahrerloses Transportsystem entwickelt. Dazu werden zunächst im *Sprint Backlog* die Anforderungen an das System beschrieben. Um eine Datenübertragung zwischen den Systemen herzustellen, muss anschließend betrachtet werden, welche Schnittstellen bei den jeweiligen Systemen zur Verfügung gestellt werden bzw. welche Möglichkeiten der Anbindung einer Schnittstelle zur Verfügung stehen. Abschließend wird die entsprechende Hardware implementiert und die Schnittstellen programmiert.

### 5.1 Sprint Backlog #1

In dem ersten *Sprint* werden alle Anforderungen umgesetzt, welche im *Product Backlog* (vgl. Kapitel 4) eine Priorität 4 erhalten haben. Das *Sprint Backlog #1* (s. Tabelle 5-1) fasst die Anforderungen nochmals zusammen. Die erste Anforderung fordert eine Datenübertragung bzw. einen Kommunikationsweg zwischen dem Trackingsystem und dem FTF. Dabei soll das FTF über eine kabellose Schnittstelle angesteuert werden. Hierfür muss die Hardware für die Schnittstelle implementiert sowie das Trackingsystem installiert und kalibriert werden. Zudem wird vorgeschrieben, dass das Fahrzeug gerade Strecken fahren soll und ein Rotieren um die eigene Achse des Fahrzeugs ermöglicht werden muss, sodass das Fahrzeug in jeder Bewegungsrichtung frei agieren kann.

Tabelle 5-1: *Sprint Backlog #1*

Sprint Backlog #1	
Nr.	Anforderung
1	Datenübertragung und Kommunikation zwischen Trackingsystem und FTF
2	Implementieren der Hardware
3	Kalibrieren des Trackingsystems
4	Ansteuern von FTF über eine kabellose Schnittstelle
5	Fahren von geraden Strecken
6	Rotieren um eigene Achse des FTF

## 5.2 Schnittstellen der vorhandenen Systeme

Wie bereits beschrieben soll eine FTS aufgebaut werden. Dieses besteht aus einem FTF und einem Trackingsystem. Damit ein FTS erzielt wird, ist eine Datenübertragung zwischen den Systemen notwendig. In diesem Kapitel werden die vorhandenen Schnittstellen der einzelnen Komponenten betrachtet.

Die Abbildung 5-1 stellt die Schnittstellen der einzelnen Systeme dar. Dabei wird deutlich, dass keine direkte Datenübertragung zwischen dem Trackingsystem und dem Fahrzeug möglich ist. Das Trackingsystem detektiert zwar die Position des FTF, hat jedoch keine Möglichkeit selbst Daten an das FTF zu senden. Daher muss ein Umweg über eine weitere Schnittstelle gegangen werden. Über ein Skript kann eine Schnittstelle generiert werden, welche die Fahrzeukoordinaten von Trackingsystem einliest und des Weiteren die Fahrbefehle an das Fahrzeug übergibt. Das Trackingsystem überträgt seine Daten über eine REST API. Das *Arduino Board* des Fahrzeugs bietet hingegen mehrere Anschlussmöglichkeiten, um dieses mit einem Bluetooth-Modul oder anderen Funkmodulen wie z.B. ZigBee oder APC220 auszustatten. Aus Verfügbarkeitsgründen, Platzgründen sowie Gründen des minimalen Aufwands wird ein Bluetooth-Modul für die Kommunikation gewählt. Über das Skript wird das Bluetooth-Modul angesprochen. Die folgenden Abschnitte sollen daher die Funktionsweise von REST API und Bluetooth beschreiben.

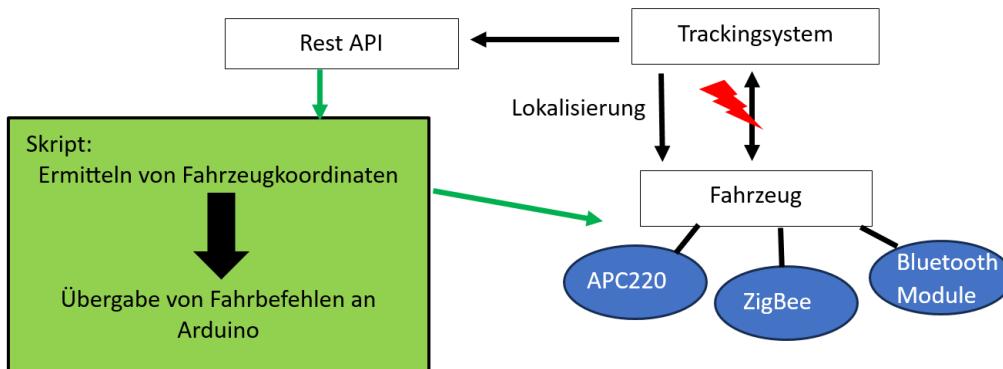


Abbildung 5-1: Schnittstellen zwischen den Systemen

### 5.2.1 Rest API

Eine API (*Application Programming Interface*) ist eine Schnittstelle zum Austausch von Daten, in der Regel zwischen einem *Client* und einem *Server*. REST ist die Abkürzung für *Representational State Transfer*. Es bezeichnet einen standardisierten Weg zum Austausch dieser Daten. REST APIs sind Zustandslos (engl. *stateless*). Das bedeutet *Client* und *Server* müssen Zustandslos miteinander kommunizieren. Jede Anfrage eines *Clients* beinhaltet daher alle Informationen, die ein *Server* benötigt. Der *Server* selbst kann auf keinen gespeicherten Kontext zurückgreifen. REST wird insbesondere für Webservices benutzt und basiert auf der Web-Technologie *World Wide Web (WWW)*. Das *WWW* ist ein interaktives Informationssystem zum weltweiten Austausch von Daten. Es besteht aus Hypertext-Systemen, den sogenannten Websites. Eine Website umfasst in der Regel mehrere zusammenhängende Webdokumente. Diese befinden sich auf speziellen Rechnern, welche an das Internet angebunden sind. Von den Rechnern werden sie mithilfe von Web-Server-Diensten bereitgestellt. Um auf die Dokumente zugreifen zu können, sind Browser notwendig. Die Adressierung eines Webdokumentes erfolgt durch die URI (*Uniform Ressource Identifier*). Für die Hypertext-Dokumente im *WWW* wird eine plattformunabhängige Sprache *HTML (Hypertext Markup Language)* verwendet. Zum Aufrufen der Dokumente im Netz über Hyperlinks, wurde für die Übertragung der Daten das Hypertext-Transferprotokoll (abk. http) entwickelt. Die Abbildung 5-2 stellt dieses Prinzip in vereinfachter Form dar. Über einen Webbrowser wird eine Website mittels HTML dargestellt. Wird nun ein Link auf dieser Website angeklickt, so wird über die entsprechende URI ein Webdokument auf dem

Webserver angefragt. Mithilfe von HTTP werden die Inhalte abgerufen. Über HTML werden diese Inhalte wiederum auf dem Browser angezeigt. (vgl. Lackes, 2018; FH Bielefeld, 2017)

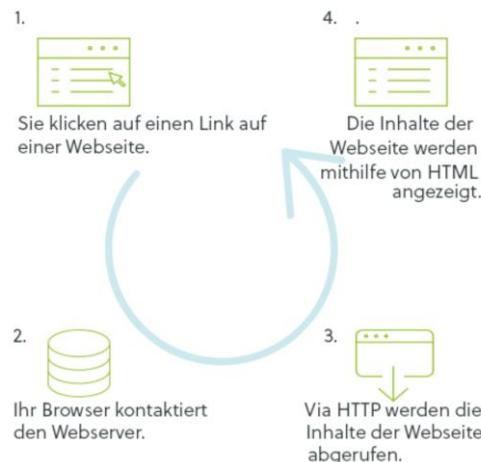


Abbildung 5-2: Funktionsweise zum Abrufen von Webdokumenten im WWW  
(fonial GmbH, 2023)

Die REST API funktioniert auf ähnliche Weise. Der *Client* fragt über die REST API Informationen von einer Ressource an (s. Abbildung 5-3). Die REST API bearbeitet diese Anfrage und gibt alle relevanten Informationen über die Ressource an den *Client* zurück. Dabei werden die Informationen in ein Format übersetzt, das der *Client* leicht interpretieren kann. Zudem können *Clients* über eine REST-API Elemente auf einem Server ändern und auch neue Elemente hinzufügen. Die Anfragen erfolgen über HTTP-Befehle. Diese sind ähnlich zu den typischen Datenbankbefehlen *create*, *read*, *update* und *delete*. Ebenso gibt es vier HTTP-Befehle: (vgl. Parmar, 2022)

- POST um eine Ressource mittels *create* zu erstellen
- GET um eine Ressource mittels *read* abzurufen bzw. zu lesen
- PUT um eine Ressource mittels *update* zu bearbeiten bzw. zu aktualisieren
- DELETE um eine Ressource mittels *delete* zu löschen

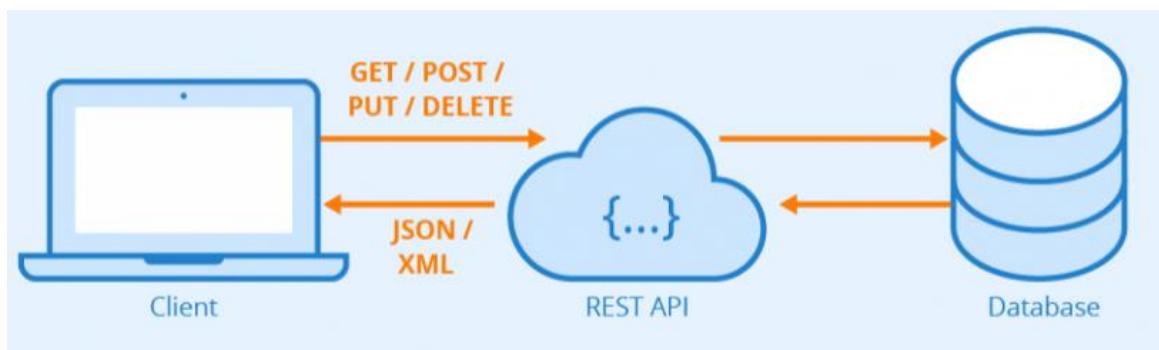


Abbildung 5-3: REST API  
(Seobility, 2023)

Um eine Anfrage an eine Ressource zu senden, muss die Ressource bekannt sein bzw. identifiziert werden. Dies erfolgt ähnlich wie beim WWW über eine URI. Die URI definiert den Zugriffsort einer einzelnen Ressource in einem Netzwerk. Dabei muss der Identifier nach einem standardisierten Schema aufgebaut sein, welches fünf Teile spezifiziert. (vgl. FH Bielefeld, 2017)

- **Schema:** Unter dem Schema wird das Protokoll verstanden, mit dem die jeweilige Ressource anzufragen ist. Für REST wird in der Regel HTTP verwendet.
- **Autorität:** Die Autorität bezeichnet die Organisation, der die angebotene Ressource unterliegt
- **Pfad:** Der Pfad beschreibt an welcher Stelle sich die Ressource innerhalb der Organisation befindet und ist typischerweise hierarchisch aufgebaut. Die einzelnen Hierarchieebenen werden hierbei mit ein *Backslash* voneinander getrennt.
- **Query:** Ein Query ist optional um eine Abfrage zu erweitern. Sie wird genutzt, wenn der Ort der Ressource allein durch die Pfadangabe nicht genau angegeben werden kann. Sie werden zudem verwendet, um aus einer Quelle, welche durch den Pfad bezeichnet wird, spezifische Informationen abzurufen wie z.B. einen Datensatz aus einer Datenbank.
- **Fragment:** Ein Fragment wird ausschließlich seitens des *Clients* ausgelöst und nicht per Anfrage an den *Server* übertragen. Je nach verwendetem Protokoll kann auch die Verwendung von Fragmenten variieren. Wenn ein Fragment beispielsweise Bestandteil einer HTTP-URI ist, der ein HTML-Objekt liefert, scrollt der Browser nach Darstellung der kompletten Webseite automatisch zu dem von dem Fragment beschriebenen HTML-Tag.

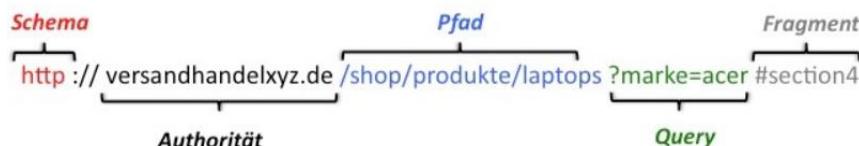


Abbildung 5-4: Bestandteile einer URI  
(FH Bielefeld, 2017)

### 5.2.2 Datenaustausch über Bluetooth

Bluetooth ist eine Funktechnologie über die Geräte drahtlos und ohne direkte Sichtverbindung miteinander kommunizieren können. Daher wird es für eine Vielzahl von Anwendungsfällen eingesetzt. Einer der häufigsten Anwendungsfälle für Bluetooth ist eine kabellose Verbindung zwischen einem Smartphone und Audiogeräten wie z.B. Kopfhörer oder Freisprecheinrichtungen. Da heutzutage eine Vielzahl an unterschiedlichen Herstellern Bluetooth-Geräte anbieten, ist eine einwandfreie Interoperabilität grundlegende Voraussetzung für eine reibungslose Kommunikation. Dies wird mithilfe von Standards sichergestellt. Die Kommunikation zwischen den Geräten erfolgt über Piconetze. Damit mehrere Piconetze gleichzeitig an einem Ort betrieben werden können, verwendet jedes Piconetz seine eigene *Hopping Sequenz*. Dafür stehen Bluetooth 79 Kanäle zur Verfügung. Diese Anzahl ist ausreichend, um an einem Ort viele Bluetooth Netzwerke gleichzeitig und ohne wesentliche gegenseitige Beeinflussung zu betreiben. Die *Hopping Sequenz* des Piconetzes wird durch die HardwareAdresse des Endgerätes berechnet, das als erstes Kontakt zu einem anderen Endgerät aufnimmt. Dieses Endgerät baut auch das Piconetz auf. In einem Piconetz existiert immer ein Master. Der Master kann mit bis zu sieben *Slave*-Endgeräten verbunden sein. Da Bluetooth oftmals nur bei Punkt-zu-Punkt-Verbindungen eingesetzt wird, ist diese Anzahl an Geräte innerhalb eines Netzes vollkommen genügend. Jedes Endgerät kann *Master* oder *Slave* eines Endgerätes sein. Dabei ist jenes Endgerät der *Master*, welches das Piconetz ursprünglich aufbaut. Der *Master* hat stets die Kontrolle und entscheidet, wer zu welchem Zeitpunkt Daten übertragen darf. (vgl. Sauter, 2022: S.339-346)

### 5.3 Implementierung der Hardware und der Softwareschnittstellen

Die Implementierung der Hardware und der Softwareschnittstellen sind elementar für den Aufbau einer Bewegungssteuerung eines FTS. Als Hardwarekomponenten sollen zum einen das Trackingsystem implementiert werden, zum anderen soll die Einbindung eines Bluetooth-Moduls erfolgen. Als Bluetooth-Modul wird ein *HC05 Bluetooth Wirelss RF-Transceiver-Modul RS232* eingesetzt.

#### 5.3.1 Installation und Kalibrierung der Trackingkamera

Diese Kapitel beschäftigt sich mit der Installation und Kalibrierung des Trackingsystems. Da zu Beginn ausschließlich die Trackingkamera sowie einer Kalibriervorlage der Größe A0 vorliegt, gilt es zunächst die Trackingkamera aufzubauen und die notwendigen Komponenten für die Kalibrierung zur Verfügung zu stellen. Die Trackingkamera wird an einen Querbalken aus Holz geschraubt. Diese wird gestützt von zwei Stativen. Über die Stativen kann die Höhe der Trackingkamera eingestellt werden. Dabei ist es wichtig darauf zu achten, dass die Stativen dieselbe Höhe aufweisen (s. Abbildung 5-5).



Abbildung 5-5: Aufbau des Trackingsystems

Sobald die Trackingkamera aufgebaut ist, wird diese verkabelt. Die Verkabelung erfolgt entsprechend der Installationsanleitung (s. Abbildung 5-6). Für die Installation wird ein *PoE-Swtich (Power over Ethernet)* benötigt. Ein *PoE-Switch* ist ein Netzwerk-Switch, über den netzwerkfähige Geräte per Ethernetkabel mit elektrischem Strom versorgt werden. Gleichzeitig können ebenfalls Netzwerksignale übertragen werden (vgl. Intellinet, 2023). Die Trackingkamera wird via Ethernetkabel an den *PoE-Switch* angeschlossen. Der *PoE-Switch* wird über ein Netzteil mit Strom versorgt. Des Weiteren wird der *PoE-Switch* mit dem Laptop bzw. mit dem Computer verbunden. Der Computer bzw. Laptop wiederrum wird über ein LAN-Kabel mit dem lokalen Netzwerk verbunden.

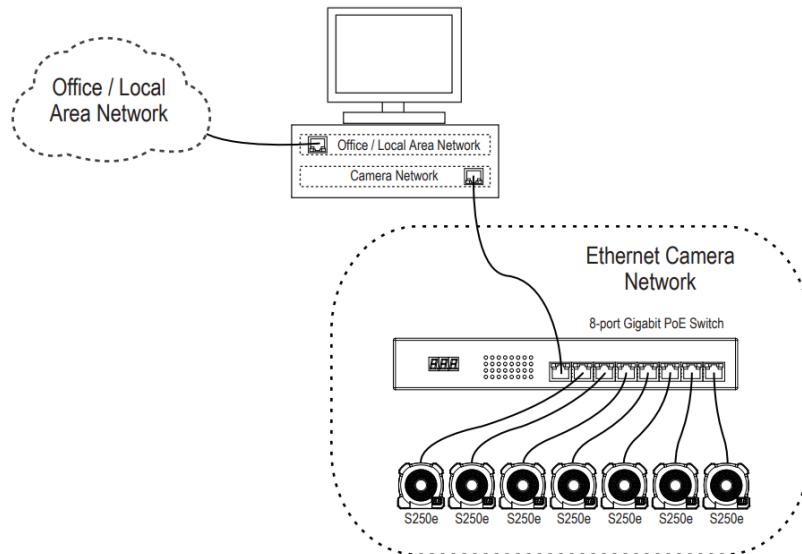


Abbildung 5-6: Installation der Trackingkamera  
( Natrual Point, 2010)

Nachdem die Trackingkamera erfolgreich aufgebaut ist, ist nun die Kalibrierung der Kamera erforderlich. Zur Kalibrierung bzw. zum Betrieb der Trackingkamera stellt die Hochschule eine 2D-Tracker-Anwendung bereit. Diese verfügt über eine HTML-Seite, die die Nutzung dokumentiert und ermöglicht (s. Abbildung 5-7). Die REST-Schnittstelle des Trackers ist über die URI <http://laptop-n1q2j4ee:1201> zu erreichen. Über die URI <http://laptop-n1q2j4ee:1201/data?homography=-2> können die getrackten Objekte abgefragt werden. Wenn bei dem Query der Wert -2 auf -1 geändert wird, so kann die Kalibrierung erfolgen. Nach der einmaligen anfänglichen Kalibrierung, kann der Wert des Querys wieder auf -2 zurückgesetzt werden. Dieser ruft dann automatisch die kalibrierte Konfiguration ab.

## 2dTracker

2dTracker is using [Naturalpoint Optitrack](#) cameras to recognize reflective markers and provide their 2d-coordinates as a REST-server.

All provided positions and lengths are in mm; orientations in rad

<a href="#"><u>/index.html</u></a>	This list of GET-query commands.
<a href="#"><u>/cameras</u></a>	List of found OptiTrack cameras.
<a href="#"><u>/image.png</u></a>	Current camera image with tracking data.
<a href="#"><u>/data?homography=-2</u></a>	Sets the homography calculation interval and shows current homography-matrix.
<a href="#"><u>/data?markers</u></a>	Lists all recognized marker objects.
<a href="#"><u>/data?bodies</u></a>	Lists all recognized body pairs of marker with its distances.
<a href="#"><u>/data?body=dist</u></a>	Replies with the body whose marker-distance is closest to the requested <i>dist</i> .
<a href="#"><u>/data?body=dist\$tol</u></a>	Replies with the body whose marker-distance is closer than <i>tol</i> to the requested <i>dist</i> .

Abbildung 5-7: HTML-Seite des Trackers

Zur Durchführung der Kalibrierung muss also der *Query* der URI auf den Wert -1 gesetzt werden. Zudem wird unter die Trackingkamera die Kalibriervorlage gelegt. Auf der Kalibriervorlage sind vier Kreise markiert. Der erste Kreis 0 hat einen Durchmesser 70mm. Bezogen auf das eingezeichnete Koordinatensystem der Vorlage liegt dieser bei den Koordinaten  $x=-525$  und  $y=-350$ . Die übrigen Kreise weisen einen Durchmesser von 40mm auf. Sie befinden sich an den Punkten:

- a.  $X=525/Y=-350$
- b.  $X=525/Y=350$
- c.  $X=-525/Y=350$

An diese Koordinaten müssen reflektierende Marker der entsprechenden Größe positioniert werden. Die reflektierenden Marker werden von der Trackingkamera erfasst und zur Kalibrierung genutzt. Da das zu verfolgende Fahrzeug allerdings eine gewisse Höhe aufweist, ist es erforderlich, dass die Marker auf derselben Höhe eingelesen werden. Ansonsten kommt es zu Parallaxenfehlern. Damit die Marker dieselbe Höhe annehmen, werden mittels 3D-Drucker sogenannte Kalibriertürme additiv gedruckt (s. Abbildung 5-8). Die Kalibriertürme weisen eine Höhe von 74mm und entsprechen somit der Höhe des Fahrzeugs. Die Kalibriertürme werden anschließend auf die Positionen der Kalibriervorlage gestellt. Der Kalibriervorgang kann beginnen (s. Abbildung 5-9). Dazu wird das Anwendungsprogramm der Trackingkamera gestartet. Sobald die Trackingkamera die Marker erfasst, wird ein Koordinatenkreuz in die berechnete Mitte gelegt. Der *Query* der URI wird wieder auf -2 gesetzt und der Vorgang ist abgeschlossen. Die Kalibriertürme können beiseitegestellt werden.



Abbildung 5-8: Kalibriervorlage (links) und Kalibrierturm (rechts)

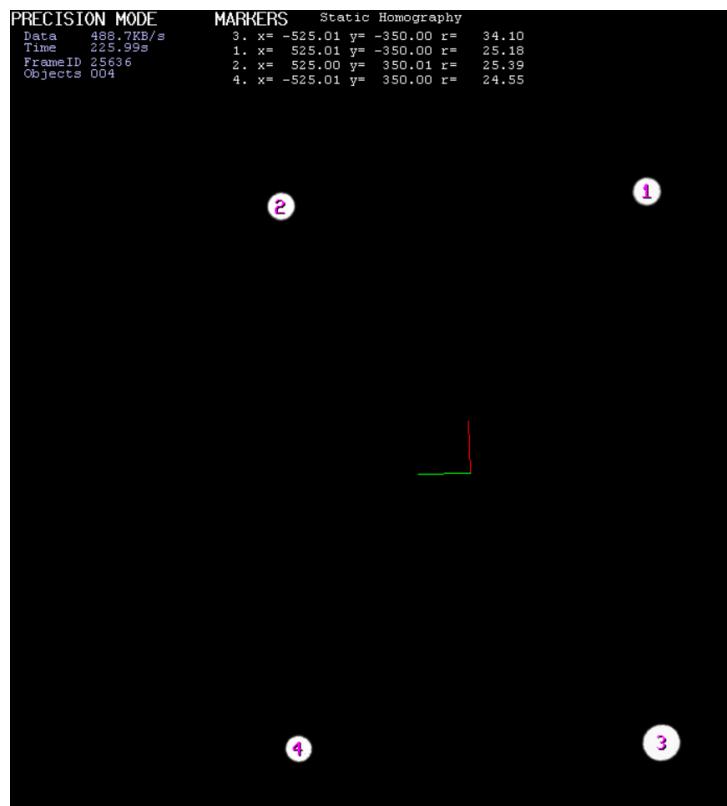


Abbildung 5-9: Koordinatensystem des Trackers nach Kalibrierung

Als nächstes soll das Verfolgen des Fahrzeuges getestet werden. Dafür wird das Fahrzeug mit zwei reflektierenden Markern unterschiedlicher Größe ausgestattet, deren Positionen in der Ebene durch den optischen Tracker erfasst werden können. Das System erkennt das Fahrzeug bzw. eine Komponente anhand des Abstands zwischen den Markern und legt ein Koordinatensystem für die Komponente fest. Dieses Koordinatensystem hat seinen Ursprung in der Position des größeren Markers und die x-Achse zeigt in Richtung des kleineren Markers. Dadurch kann nicht nur eine Fahrzeugposition, sondern auch eine Fahrzeugorientierung ausgegeben werden. Wichtig dabei ist, dass für das Anwendungsprogramm immer vier Marker gleichzeitig von der Kamera erfasst werden, da es sonst zum Absturz des Anwendungsprogramms kommen kann und keine Positionen ausgegeben werden. Das bedeutet, selbst wenn lediglich zwei Marker für die Verfolgung des Fahrzeugs genutzt werden, ist es notwendig, dass sich zwei weitere Marker im Raum befinden. Die Abbildung 5-10 zeigt beispielhaft die HTML-Seite, die die vorhandenen Marker beschreibt. Die Marker werden über die Distanzen jeweils zu *Bodies* zusammengefasst.

## Bodies

```

1: dist=53.5
marker1 : {nr : 4, x : -348.8, y : 135.6, r : 14.6}
marker2 : {nr : 3, x : -307.7, y : 169.9, r : 9.8}

2: dist=111.9
marker1 : {nr : 2, x : 545.1, y : -353.4, r : 25.0}
marker2 : {nr : 1, x : 574.2, y : -245.3, r : 24.9}

3: dist=974.8
marker1 : {nr : 1, x : 574.2, y : -245.3, r : 24.9}
marker2 : {nr : 3, x : -307.7, y : 169.9, r : 9.8}

```

Abbildung 5-10: Trackingsystem "Bodies"

### 5.3.2 Implementieren des Bluetooth-Moduls HC05

In diesem Kapitel wird die Implementierung eines Bluetooth-Moduls betrachtet. In dieser Arbeit soll das *HC05 Bluetooth Wireless RF-Transceiver-Modul RS232* eingesetzt werden (s. Abbildung 5-11). Dies ist ein leistungsfähiges Bluetooth-Gerät, das entweder als *Master* oder *Slave* eingestellt werden. Mit dem integrierten Bluetooth-Transceiver-Gerät können serielle Schnittstellen in eine Bluetooth-Schnittstelle umgewandelt werden. Daher wird es häufig verwendet, um eine Kommunikation zwischen einem *Arduino-Board* und einem anderen Gerät wie z.B. einem Computer herzustellen. Voraussetzung ist dabei, dass der Computer bzw. das andere Gerät mit einer Bluetooth-Verbindung ausgestattet ist. (vgl. AZ-Delivery, 2023)



Abbildung 5-11: *HC05 Bluetooth Wireless RF-Transceiver-Modul RS232*  
( AZ-Delivery, 2023)

Bevor das HC05-Modul an das *Arduino-Board* des Fahrzeugs angeschlossen werden kann, ist es erforderlich das Modul einzustellen. Dies gestaltet sich einfacher mit Hilfe eines zweiten *Boards* wie z.B. einem *Arduino Uno* oder einem *Arduino Mega*. Da ein *Arduino Mega-Board* verfügbar ist, soll dieses im Folgenden benutzt werden. Das HC05-Modul kann in zwei verschiedenen Modi betrieben werden. Es gibt den AT-Befehlsmodus und den Datenmodus. Zum Einstellen des Moduls wird der AT-Befehlsmodus verwendet, während zum Austausch von Daten der Datenmodus genutzt wird.

Das HC05-Modul besitzt fünf Pins, einen Druckknopf und eine LED. Es werden die Pins *State*, *VCC*, *Ground*, *RX*, *TX* und *Enable/Key* unterschieden: (vgl. Components101, 2021)

- **Enable/Key:** Dieser Pin wird zum Umschalten zwischen Datenmodus und AT-Befehlsmodus verwendet. Standardmäßig befindet sich das HC05-Modul im Datenmodus.
- **VCC:** Versorgt das Modul mit Strom und ist an eine 5V-Versorgungsspannung anzuschließen
- **Ground:** Ist der Erdungsstift des Moduls. Dieser muss entsprechend mit dem „*Ground*“ des *Arduino Boards* verbunden werden.
- **TX-Sender:** Überträgt serielle Daten. Alles, was über Bluetooth empfangen wird, wird über diesen Pin als serielle Daten ausgegeben.
- **RX-Empfänger:** Empfängt serielle Daten. Alle an diesen Pin übermittelten seriellen Daten werden über Bluetooth übertragen.
- **State:** Der State-Pin ist mit der integrierten LED verbunden und kann als Rückmeldung verwendet werden, um zu überprüfen, ob Bluetooth ordnungsmäßig funktioniert.
- **LED:** Die LED zeigt den Status des Moduls an:
  - Einmaliges Blinken alle 2s bedeutet, das Modul ist in den AT-Befehlsmodus übergegangen.
  - Wiederholtes Blinken bedeutet das Modul wartet auf eine Verbindung im Datenmodus
  - Ein Blinken zweimal pro Sekunde weist daraufhin, dass die Verbindung im Datenmodus erfolgreich war
- **Druckknopf:** Wird zur Steuerung des *Key/Enable*-Pins zum Umschalten zwischen Daten und Befehlsmodus verwendet.

Die Abbildung 5-12 zeigt wie das HC05-Modul an das *Arduino Mega Board* angeschlossen werden. Der *Ground-Pin* wird mit dem *Ground-Pin* über ein Kabel verbunden (orange), RX geht auf RX (blau), TX wird mit TX (rot) verkabelt, der VCC-Pin erhält seine Versorgungsspannung über den 5V-Pin des *Boardes* (grün) und der *Enable/Key-Pin* wird mit dem 3,3V-Pin gekoppelt (gelb).

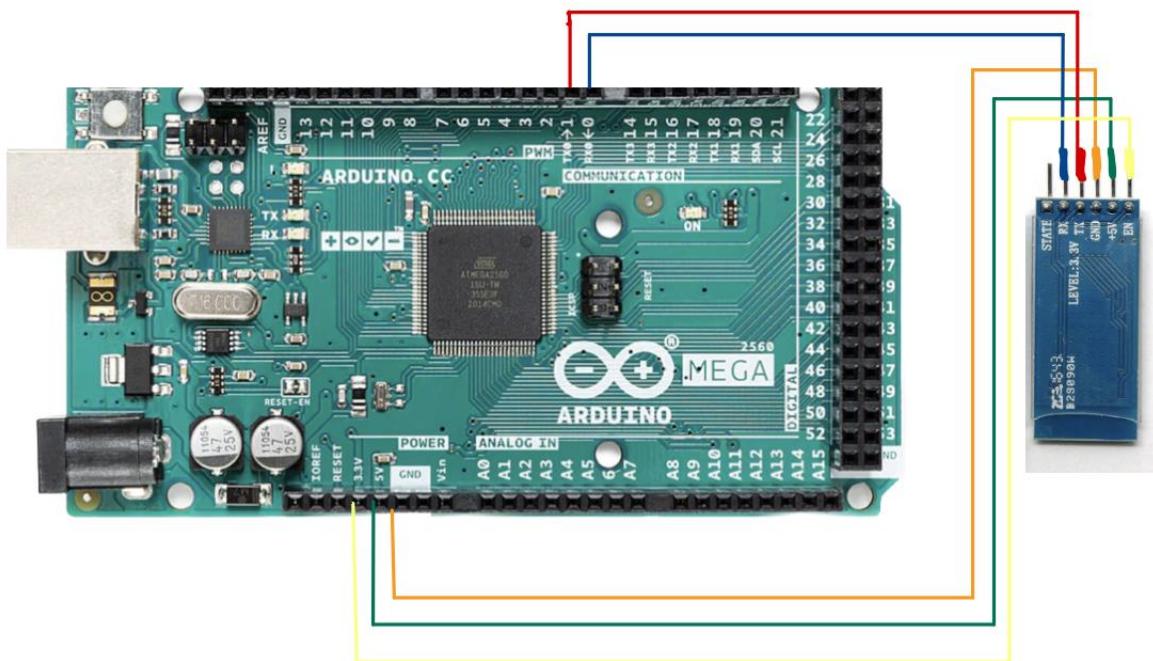


Abbildung 5-12: Verbinden der HC05-Moduls mit dem *Arduino Board "Arduino Mega"*

Im nächsten Schritt wird auf dem Computer die *Arduino IDE* geöffnet und ein leeres Skript geladen. Das *Arduino Board* wird mittels USB-Kabel mit dem Computer verbunden und das leere Skript auf das *Board* geladen. Anschließend wird die Verbindung zwischen VCC und 5V nochmals getrennt und wieder miteinander verbunden, allerdings muss der Druckknopf während dem Verbinden durchgängig gedrückt werden. Dadurch wird das Modul in den AT-Befehlsmodus versetzt. Die LED blinkt alle zwei Sekunden und bestätigt somit den Modus. Da sich das Modul nun im richtigen Modus befindet, kann dieses eingestellt werden. Dafür wird der serielle Monitor der *Arduino IDE* geöffnet. Hier werden die Befehle zum Einstellen des Moduls eingegeben. Die Abbildung 5-13 zeigt die Befehlskette, die an dieser Stelle genutzt wird, um Informationen vom Modul abzufragen und um die Baudrate einzustellen. Die Baudrate gibt an, wie viele Symbole pro Sekunde übertragen werden. Ein Baud entspricht einem Symbol pro Sekunde (vgl. Kunbus, 2023).

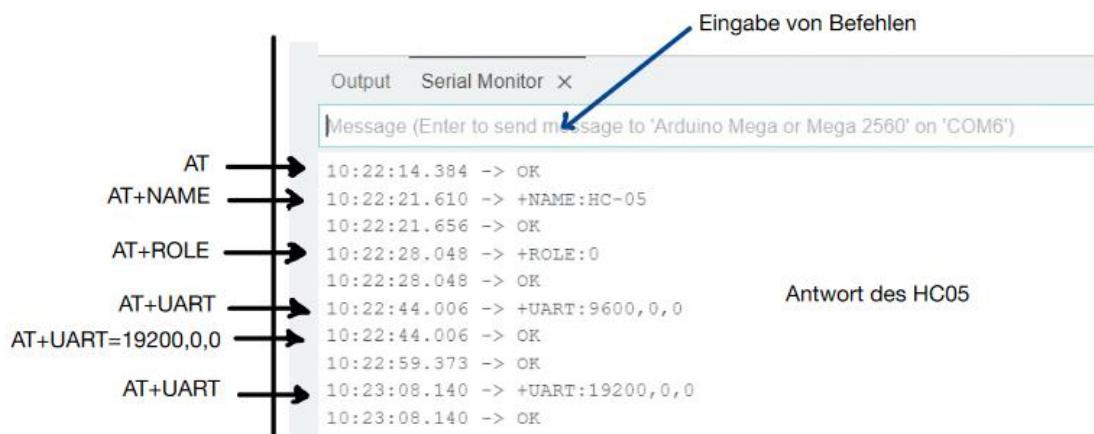


Abbildung 5-13: Befehlsfolge zum Einstellen des HC05-Moduls

Zunächst wird das Modul über den Befehl „AT“ angefragt, dieser bestätigt die Verbindung mit der Rückgabe „Ok“. Anschließend werden mit „AT+NAME“, „AT+ROLE“ und „AT+UART“ Informationen über das Modul abgefragt. Durch den Befehl „AT+ROLE“ soll festgestellt werden, ob das Modul als Master oder als Slave eingestellt ist. Durch die Rückgabe des Wertes 0, gibt dieser an, dass er derzeit ein Slave ist. Somit ist er für den Anwendungsfall dieser Arbeit im korrekten Zustand. Über den Befehl „AT+UART“ wird die Baudrate abgefragt. Diese liegt standardmäßig bei 9600Baud. Damit eine schnellere Datenübertragung möglich ist, soll die Baudrate auf 19200Baud erhöht werden. Dies erfolgt über den Befehl „AT+UART=19200,0,0“. Durch erneutes Abfragen der Baudrate wird diese bestätigt.

Nachdem das Modul erfolgreich eingestellt ist, wird dieses an das *Arduino Board* des Fahrzeugs angeschlossen. Der Anschluss erfolgt ähnlich wie zuvor, allerdings müssen ein paar Änderungen vorgenommen werden. Anders als beim Anschluss zum *Arduino Mega* werden die Pins RX und TX vertauscht und die Verbindung zwischen dem *Enable/Key-Pin* und dem 3,3V-Pin entfällt (s. Abbildung 5-14). Die LED des Moduls blinkt und wartet somit auf den Aufbau einer Verbindung.

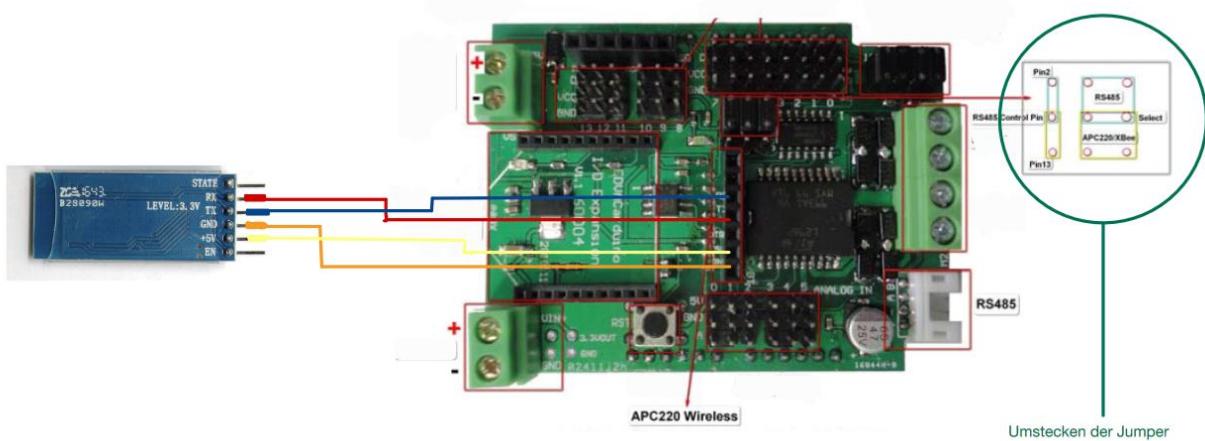


Abbildung 5-14: Verbinden des HC05-Moduls mit *Arduino Board* des Fahrzeuges

Zum Aufbauen einer Verbindung wird in den Bluetooth-Einstellungen des Computers das HC05-Modul hinzugefügt und mithilfe eines Passwortes (standardmäßig 1234) können die Geräte miteinander gekoppelt werden. Eine Kopplung der Geräte bedeutet jedoch nicht automatisch, dass diese miteinander zum Datenaustausch verbunden sind. Für den Datenaustausch muss der COM-Port des Moduls ermittelt werden. Dies geschieht ebenfalls über die Bluetooth-Einstellungen des Computers. In diesem Fall wird das Bluetooth-Modul über den COM5 angesprochen. Über die Zuhilfenahme eines Python Skriptes, das die Verbindung testet, sollte nun ein Austausch von Daten möglich sein. Jedoch scheitert der Verbindungsauflauf bei jedem Versuch. Nach mehrmaligem Testen wird festgestellt, dass dies mit der Jumperbelegung auf der *Arduino*-Erweiterungsplatine zusammenhängt (s. Abbildung 5-14 grün markiert). Das *Pinout* der Platine gibt an, dass je nach Jumperbelegung entweder die Schnittstelle RS485 oder die Schnittstelle APC220 verwendet werden kann. Da das Bluetooth-Modul an der APC220-Schnittstelle angeschlossen ist, müssen die Jumper dementsprechend umgesteckt werden. Dies hat allerdings den Nachteil, dass die RS485-Schnittstelle nicht mehr verwendet werden kann. Allerdings kommunizieren hierüber die Ultraschallsensoren. Da die Ultraschallsensoren für diese Arbeit zunächst keine Anwendung finden, ist das Umstecken der Jumper legitim. Wenn die RS485- und die APC220-Schnittstelle jedoch gleichzeitig betrieben werden soll, muss eine andere Lösung geschaffen werden. Nachdem sich die Jumper an der richtigen Position befinden, kann eine Verbindung hergestellt werden. Die LED des Moduls bestätigt dies durch wiederholtes zweimaliges Blinken innerhalb einer Sekunde. Die Implementierung des Moduls ist erfolgreich abgeschlossen.

### 5.3.3 Programmcode zum Datenaustausch

Nach erfolgreicher Implementierung der Hardwarekomponenten sollen nun die Softwareschnittstellen zum Datenaustausch betrachtet werden. Zunächst soll ein Skript geschrieben werden, welches die Position und Orientierung des Fahrzeuges im Trackingsystem ausgibt.

Die REST API stellt die Daten als JSON bereit. Wenn die Koordinaten angefragt werden, wird dementsprechend ein Status Wert sowie eine JSON-Struktur mit den angeforderten Daten zurückgegeben. JSON oder *JavaScript Object Notation* ist ein text-basiertes Dateiformat. JSON ist komplett unabhängig von Programmiersprachen, daher eignet sich dieses Format besonders zum Datenaustausch. Bei JSON werden die Daten in Paaren Name/Wert bzw. Schlüssel/Wert ausgedrückt (vgl. 4D SAS, 2023).

Damit Python die JSON-Daten verarbeiten kann, ist der Import der JSON-Bibliothek notwendig (s. Programmcode 5-1). Des Weiteren werden die Bibliotheken *Requests* und *Time* benötigt. Die Bibliothek *Requests* erlaubt es, HTTP-Anfragen einfach zu versenden. Die Bibliothek *Time* bietet zeitbezogene Funktionen an, welche nach dem Import der Bibliothek genutzt werden können.

Der Befehl `r=requests.get(URI)` liest die Daten von einer Ressource mit der entsprechenden URI aus. Die URI beinhaltet die Adresse des Trackers. Über den Query „body=56“ am Ende der URI wird die Komponente ausgewählt, welche eine Markerdistanz von 56mm aufweist. Dadurch kann das Fahrzeug eindeutig identifiziert werden und es werden die Fahrzeugkoordinaten sowie der Status 200 zurückgegeben. Zuletzt werden über den Befehl `print()` die Positionen x und y sowie die Orientierung *phi* in der Konsole ausgegeben, sodass der User eine grafische Rückmeldung der Informationen erhält.. Diese HTTP-Anfrage ist in eine *while True*-Schleife mit einer Zeitverzögerungsfunktion `time.sleep(2)` eingebunden. Daher erfolgt die Anfrage der Daten alle zwei Sekunden. Es können somit Änderungen der Position und Orientierung des Fahrzeugs direkt wahrgenommen werden. Letztlich kann festgestellt werden, dass die Daten über das Python-Skript erfolgreich von der REST API ausgelesen werden.

```

1 import json
2 import requests
3 import time
4
5 if __name__ == '__main__':
6     while True:
7         r = requests.get("http://laptop-n1q2j4ee:1201/data?body=56")
8         r=json.loads(r.text)
9         print("x="+str(r["x"]))
10        print("y="+str(r["y"]))
11        print("phi="+str(r["phi"]))
12        print("")
13
14        time.sleep(2)

```

Programmcode 5-1: Auslesen der Trackingdaten

Im nächsten Schritt soll nun eine Verbindung zwischen dem Computer und dem HC05-Modul des Fahrzeugs hergestellt werden. Dafür wird auf der einen Seite ein Python Code benötigt und auf der anderen Seite ein entsprechendes *Arduino Sketch*, welches auf das Fahrzeug geladen werden kann. Als erstes wird der Python Code betrachtet.

Um eine serielle Kommunikation herstellen zu können, wird die Bibliothek *Serial* bzw. *pyserial* gebraucht. Zu Beginn wird diese daher importiert. Anschließend wird eine Verbindung über die Funktion `arduinoData=serial.Serial('COM5', 19200)` aufgebaut, wobei `arduinoData` eine beliebige Variable darstellt. Der COM-Port COM5 gibt dabei den Ausgang des HC05-Moduls an. Die Zahl 19200 ist die verwendete Baudrate. Die angegebene Baudrate muss die des HC05-Moduls entsprechen. Im *Arduino Sketch* ist ebenfalls diese Baudrate anzugeben. Nun wird eine *while*-Schleife gestartet, in der der

Benutzer eine Eingabe vornehmen kann. Das eingegebene Kommando wird im Anschluss um den String '\n' erweitert. Bei der Datenübertragung wird damit das Ende des Kommandos angezeigt. Letztlich wird mit dem Befehl `arduino.write(cmd.encode())` das Kommando bzw. die Eingabe an die serielle Schnittstelle übertragen. Bevor die Übertragung erfolgt, wird durch die Funktion `cmd.encode()` das Kommando in eine andere Kodierungsform umgewandelt. Standardmäßig wird hierzu UTF8 genutzt. UTF8 ist die am weitesten Verbreitung für Unicode-Zeichen. Es wird jedes Zeichen in ein Binärcode aus Nullen und Einsen umgewandelt. Dies ist erforderlich, da die serielle Schnittstelle ausschließlich Bits übertragen kann und keine Strings. Nachdem eine Eingabe erfolgt ist, kann der Anwender direkt die nächste Eingabe starten.

```

1 import serial
2 arduinoData=serial.Serial('COM5',19200)
3
4 while True:
5     cmd=input('Please Enter Your Command: ')
6     cmd=cmd+'\n'
7     arduinoData.write(cmd.encode())

```

Programmcode 5-2: Senden von Daten über serielle Schnittstelle von Python

Zur Entgegennahme des gesendeten Kommandos wird ein *Arduino-Sketch* aufgebaut. Dazu wird der Programmcode des Demoskriptes aufgegriffen (vgl. Kapitel 3.2.5 Demoskript). Es werden jedoch alle Funktionen entfernt, die mit den Ultraschallsensoren in Verbindung stehen, da diese derzeit nicht relevant sind. Da in dem SprintBacklog #1 die Anforderungen gestellt werden, dass das Fahrzeug zum einen gerade Strecken fahren soll und zum anderen eine Rotation um die Fahrzeugachse ausgeführt werden soll, werden die Funktionen für die Fahrrichtungen von dem Demoskript übernommen. Das Array `void motion[]` wird jedoch angepasst. Es enthält nun sechs Bewegungsfunktionen sowie eine Stopp-Funktion (s. Programmcode 5-3). Außerdem wird eine neue Funktion `driveCar` mit einer Variablen `direction` eingeführt. Im Folgenden soll diese Funktion kurz beschrieben werden.

```

85 void(*motion[7])(unsigned int speedMMPS) = {goAhead,turnLeft,turnRight,backOff,rotateLeft,rotateRight,allStop};
86
87 void driveCar(unsigned speedMMPS){
88     unsigned int direction;
89     while(Serial.available()==0){
90     }
91
92     direction=Serial.readStringUntil('\n').toInt();
93     (*motion[direction])(speedMMPS);
94
95     Omni.PIDRegulate();
96
97 }
98
99 void setup() {
100 Serial.begin(19200);
101
102 delay(2000);
103 TCCR1B=TCCR1B&0xf8|0x01;      // Pin9,Pin10 PWM 31250Hz
104 TCCR2B=TCCR2B&0xf8|0x01;      // Pin3,Pin11 PWM 31250Hz
105
106 Omni.PIDEable(0.35,0.02,0,10);
107 }
108
109 void loop() {
110     driveCar(100);
111 }

```

Programmcode 5-3: Arduino-Sketch zur Annahme und Verarbeitung von Befehlen

Mithilfe einer *While-Schleife* wird solange keine Aktion durchgeführt bis Daten über die serielle Schnittstelle ankommen. Sobald Daten eintreffen, werden diese mit der Funktion `Serial.readStringUntil('\n').toInt()` gelesen. Das bedeutet, es werden alle Bits gelesen, die bis zur Endmarkierung '\n' gesendet werden. Diese werden wieder in einen String zurücktransformiert. Dieser String soll der Variablen *direction* zugeordnet werden. Da diese jedoch als *Integer*-Wert definiert ist, muss der String nochmals umgewandelt werden in einen *Integer*. Das übernimmt die Funktion `.toInt()`. Im Anschluss wird die Variable *direction* dazu verwendet, um aus dem Array *motion[]* den entsprechenden Fahrbefehl auszuwählen. Wird vom Benutzer in der Konsole vom Visual Studio Code<sup>3</sup> beispielsweise der Wert 0 eingegeben, so wird der Fahrbefehl *goAhead* ausgeführt. Bei einem Wert von 5 wird die Funktion *rotateRight* aktiviert und das Fahrzeug soll rechtsherum drehen. Zuletzt soll mit der Funktion *Omni.PIDRegulate()* das Fahrverhalten geregelt werden.

Nun wird das `void setup()` betrachtet. Dieses enthält ebenfalls einen ähnlichen Code wie das Demoskript. Es wird lediglich die Zeile `Serial.begin(19200)` ergänzt. Durch diese wird die serielle Schnittstelle aktiviert. Der Datenaustausch erfolgt mit 19200baud und entspricht somit demselben Wert wie im Python-Skript. In der `void loop()` wird letztendlich nur noch die Funktion *driveCar(100)* aufgerufen, wobei der Wert 100 die Fahrgeschwindigkeit in mm/s angibt.

Nachdem die Schnittstelle durch diverse Eingaben des Benutzers getestet ist, kann festgestellt werden, dass eine stabile Schnittstelle zwischen dem Computer und Fahrzeug implementiert ist. Zudem werden über ein weiteres Skript die Daten des Trackers abgefragt, sodass ebenfalls eine Schnittstelle zwischen dem Computer und dem Trackingsystem vorhanden ist. Die beiden Skripte stehen jedoch bislang nicht in Verbindung zueinander. Das bedeutet es ist zwar eine Steuerung des Fahrzeugs möglich, jedoch ist diese bisher vollkommen unabhängig von dem Trackingsystem. Diese Abhängigkeit wird in dem nächsten Kapitel bzw. in dem nächsten *Sprint* geschaffen.<sup>4</sup>

---

<sup>3</sup> Visual Studio Code wird als Umgebung verwendet, um das Python Skript zu schreiben

<sup>4</sup> Der vollständige Arduino-Sketch zu dem Sprint 1 befindet sich im Anhang C.

## 6 Bewegungssteuerung entlang einer Punkt-zu-Punkt-Verbindung

Dieses Kapitel behandelt die Bewegungssteuerung hinsichtlich einer Lageregelung und Bahn- bzw-Trajektorienplanung für eine Punkt-zu-Punkt-Verbindung. Dazu werden in einem *Sprint Backlog* kurz die Anforderungen an den Sprint zusammengefasst. Anschließend werden die Grundlagen zu den Themen Kaskaden- und Lageregelung, Trajektorienplanung sowie der Transformation von Koordinatensystemen beschrieben. Diese Abschnitte enthalten zudem jeweils Herleitungen von Formeln. Diese werden für die einzelnen Programmcodes in Python benötigt. Im letzten Teil werden eben diese Programmcodes beschrieben und deren Zusammenhänge erläutert.

### 6.1 Sprint Backlog #2

Das Sprint Backlog #2 fasst die Anforderungen zusammen, die im zweiten Sprint umgesetzt werden sollen. Die erste Anforderung besagt, dass eine Überlagerung von Bewegungen möglich sein soll. Bislang wurden die Bewegungen über Stati definiert. Das heißt, wenn der Status auf „Vorausfahrt“ gesetzt wurde, wurde eine Vorwärtsbewegung ausgeführt. Nun soll es jedoch möglich sein die Bewegungsrichtungen sowie die Rotation zu überlagern. Das Fahrzeug soll also in eine beliebige Richtung im Winkel *rad* fahren können und sich gleichzeitig um die eigene Achse drehen können. Die zweite Anforderung fordert das direkte Fahren zu einem vorgegebenen Punkt. Liegt ein Startpunkt und ein gegebener Zielpunkt vor, so soll das Fahrzeug auf direktem Wege zum Ziel fahren. Dabei soll ein stetiger Geschwindigkeitsverlauf vorliegen. Dafür muss eine Trajektorie interpoliert werden. Das Bewegen entlang dieser Trajektorie bzw. der Bahn soll synchron sein. Das bedeutet, dass sowohl Zielposition als auch Zielorientierung zur selben Zeit erreicht werden sollen.

Tabelle 6-1: Sprint Backlog #2

Sprint Backlog #2	
Nr.	Anforderung
1	Überlagerung von Bewegungen
2	Direktes Fahren zu einem vorgegebenen Punkt
3	Lageregelung des FTF
4	Stetiger Geschwindigkeitsverlauf
5	Interpolation einer Trajektorie
6	Synchrone Bahnplanung

## 6.2 Grundlagen zu Kaskadenregelung, Trajektorienplanung und Transformation von Koordinatensystemen

Um eine Lageregelung und Trajektorienplanung zu implementieren erfordert es Wissen zu diesen Themenbereichen. Daher werden die Kaskadenregelung, Trajektorienplanung und Koordinatentransformation beschrieben und benötigte Formeln hergeleitet.

### 6.2.1 Kaskadenregelung

Lageregulierungen bei Robotern bzw. Fahrzeugen mit elektrischen Antrieben werden in den meisten Fällen als Kaskadenregelung ausgeführt. Jeder Motor wird dabei einzeln geregelt. Die Regelung erfolgt auf verschiedenen Ebenen. Daher wird es auch als unterlagerter Regelkreis bezeichnet. Auf der untersten Ebene arbeitet der Stromregler, welcher meistens den Typ PI-Regler entspricht. Dieser Stromregelkreis weist die höchste Dynamik auf. Ihm übergeordnet ist der Geschwindigkeitsregler. Dieser liefert den Sollwert der Motorstromes  $i_{soll}$ . Dieser ist meistens einen PI-Regler oder einen PID-Regler ausgeführt. In der obersten Ebene liegt der Lagerregler. Dieser gibt den Geschwindigkeitssollwert  $v_{soll}$  aus. Im einfachsten Fall wird hier für die Regelung ein Proportionalregler verwendet. Der Lagesollwert wird mit  $S_{soll}$  bezeichnet. Die Abbildung 6-1 zeigt eine solche Kaskadenregelung. Meist ist der Stromregler in der Leistungselektronik des Motors enthalten. Die Kaskadenregelung hat den Vorteil, dass Störungen im inneren der Kaskade mit hoher Bandbreite ausgeregelt werden. (vgl. Winkler, 2016: S.30f)

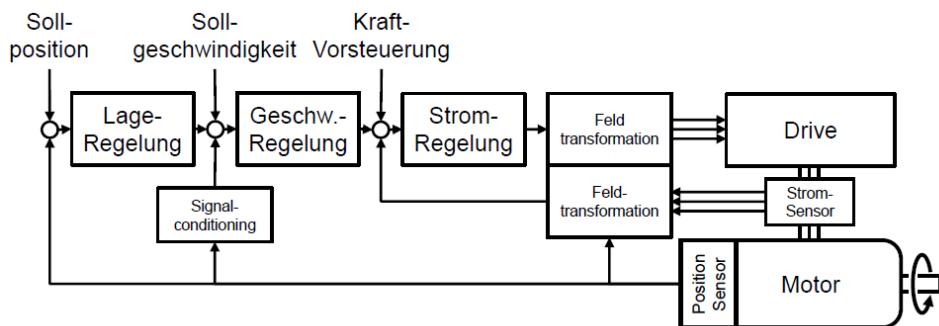


Abbildung 6-1: Kaskadenregelung  
(Hofschulte, 2022: S.230)

Auch bei dem vorhandenen System ist die Implementierung einer Kaskadenregelung erforderlich, wobei die Stromregelung bereits in der Leistungselektronik der Motoren integriert ist. Das bedeutet, es ist eine Geschwindigkeitsregelung und eine Lageregelung einzuführen. Die Geschwindigkeitsregelung wird seitens des *Arduino* implementiert, während die Lageregelung über ein Python-Skript realisiert wird. Die Auslegung des Geschwindigkeitsreglers ist allerdings nicht notwendig, da dies über die eingebundenen *Arduino*-Bibliotheken verwirklicht wird. Daher wird lediglich die Lageregelung ausgelegt.

Aus dem Regelkreis ist ersichtlich, dass bei der Lageregelung die Positionsabweichung in eine Geschwindigkeit umgewandelt wird. Bei der Sollstellung werden drei Parameter betrachtet. Zum einen die Positionskoordinaten  $x$  und  $y$ , zum anderen die Orientierung über den Winkel  $\varphi$ . Diese Sollwerte sollen erreicht werden. Daher wird aus der Sollposition und der aktuellen Position die Regelabweichung bestimmt. Die aktuelle Position wird mit  $x_{FTS}, y_{FTS}$  und  $\varphi_{FTS}$  bezeichnet, da sich an dieser Stelle das Fahrzeug befindet.

$$\Delta x = x_{soll} - x_{FTS} \quad (6-1)$$

$$\Delta y = y_{soll} - y_{FTS} \quad (6-2)$$

$$\Delta \varphi = \varphi_{soll} - \varphi_{FTS} \quad (6-3)$$

Die Regelabweichung wird über die Lageregelung in eine Geschwindigkeit umgerechnet werden. Dafür eignet sich am einfachsten ein P-Regler. Das bedeutet es wird ein Regelparameter  $k_p$  für die Umrechnung eingeführt. Bei der Regelung werden zwei Geschwindigkeiten ermittelt. Die Erste ist die translatorische Geschwindigkeit. Sie ergibt sich aus den Geschwindigkeitsanteilen in x-Richtung und y-Richtung.

$$v_x = k_p * (x_{soll} - x_{FTS}) \quad (6-4)$$

$$v_y = k_p * (y_{soll} - y_{FTS}) \quad (6-5)$$

$$v_{trans} = \sqrt{v_x^2 + v_y^2} \quad (6-6)$$

$$v_{trans} = k_p * \sqrt{(x_{soll} - x_{FTS})^2 + (y_{soll} - y_{FTS})^2} \quad (6-7)$$

Die zweite Geschwindigkeit ist die Winkelgeschwindigkeit. Diese kann unabhängig von der translatorischen Geschwindigkeit eingestellt werden. Daher wird ein weiterer Proportionalitätsfaktor  $k_\varphi$  eingeführt. Die Winkelgeschwindigkeit  $\omega$  ergibt sich aus der Winkeldifferenz und dem Proportionalitätsfaktor  $k_\varphi$ .

$$\omega = k_\varphi * (\varphi_{soll} - \varphi_{FTS}) \quad (6-8)$$

Bei den eingeführten Gleichungen sind die Sollposition und die aktuelle Position bekannt, sodass lediglich die Proportionalitätsfaktoren bestimmt werden müssen. Die Bestimmung der beiden Werte soll experimentell erfolgen. Dabei sind zwei Bedingungen zu beachten. Erstens soll der Regler nicht zu langsam auf Regelabweichungen reagieren. Zweitens soll es nicht zum Aufschwingen des Fahrzeugs und möglichst nicht zu Überschwingen kommen. Folglich dürfen die Werte weder zu klein, noch zu groß sein. Sie müssen entsprechend gut eingestellt werden.

### 6.2.2 Bahn- und Trajektorienplanung einer Punkt-zu-Punkt-Bewegung

Aufgabe eines fahrerlosen Transportfahrzeuges ist es häufig ein Produkt o.ä. zu einem bestimmten Standort zu liefern. Das Fahrzeug soll dementsprechend eine Bewegung von seinem Startpunkt zu einem Zielpunkt ausführen. Die Menge aller Punkte, die zwischen einem gegebenen Startpunkt zu einem gegebenen Endpunkt durchlaufen werden, werden in der Bahnplanung als Pfad oder Bahn bezeichnet. Ein Pfad gibt keine Auskunft über die Geschwindigkeit und Beschleunigung. Wird jedem Punkt des Pfads zu einem bestimmten Zeitpunkt eine Geschwindigkeits- und Beschleunigungsinformation zugewiesen, so ist dies eine Trajektorie. Die Trajektorie stellt die Position und Orientierung eines Körpers als Funktion der Zeit dar. Ein wichtiger Bestandteil der Trajektorienplanung ist es, Zeitfunktionen bzw. Geschwindigkeitsfunktionen für die Bahn aufzustellen. Der Zeitverlauf kann dabei linear oder eine Polynomfunktion höherer Ordnung sein. Bei der Bahnplanung können sogenannte Durchgangs- bzw. Via-Punkte festgelegt werden. Zwischen diesen Durchgangspunkten werden weitere Positionen interpoliert. Die Abbildung 6-2 verdeutlicht dieses. Die rot dargestellten Punkte sind die interpolierten Punkte. In orange dargestellt ist die Trajektorie zu einem bestimmten Zeitpunkt. Sowohl Trajektorie als auch die interpolierten Punkte sind nicht für die gesamte Bahn eingezeichnet, sondern lediglich beispielhaft an einigen Positionen. Die Trajektorienplanung liefert also eine Parameterdarstellung mit der Zeit oder einem anderen Parameter für eine Sequenz hintereinander abzufahrender Punkte.(vgl. Mareczek, 2020, S.17f])

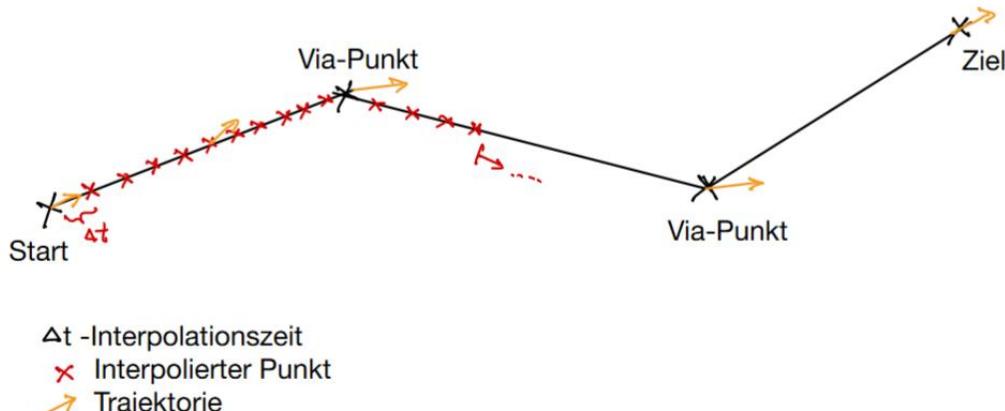


Abbildung 6-2: Trajektorie nach Hofschulte, 2022: S.191

Es gibt diverse Verfahren, um eine Trajektorienplanung durchzuführen. An dieser Stelle soll eine Trajektorie mit trapezförmigen Geschwindigkeitsverlauf betrachtet werden. Bei diesem Verfahren werden keine Via-Punkte festgelegt. Die Bahn soll somit direkt vom Start- zum Endpunkt verlaufen. Die einfachste Form ist die Interpolation der Punkt-zu-Punkt-Bewegung mit Rampenprofil. Dieser trapezförmige Geschwindigkeitsverlauf besteht aus drei Phasen: (vgl. Mareczek, 2020: S.24f)

- Phase 1: Gleichförmige Beschleunigung
- Phase 2: Fahren mit konstanter Geschwindigkeit
- Phase 3: Gleichförmige Verzögerung

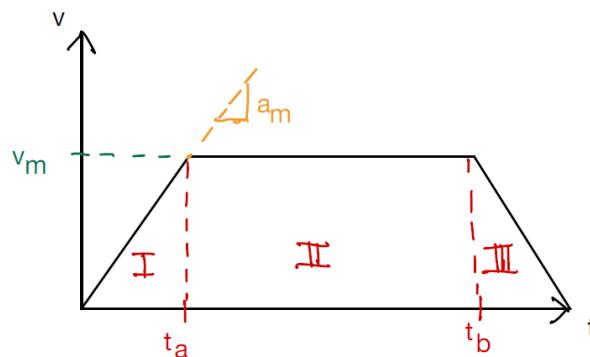


Abbildung 6-3: Geschwindigkeitsverlauf mit Rampenprofil nach Mareczek, 2020: S. 27

Vorteil dieses Verfahrens ist die einfache Berechnung der Trajektorienparameter. Nachteilig ist die sprungförmige Aufschaltung der Beschleunigung, sodass die Bewegung nicht frei von Ruck ist. Für die Beschreibung dieses Geschwindigkeitsverlaufs werden folgende Größen eingeführt:

- Start- und Endzeitpunkt  $t_0$  und  $t_e$
- Beschleunigungsdauer  $t_a$
- Zeitpunkt, ab welchem der Bremsvorgang beginnt  $t_b$
- Betrag der Beschleunigung und der Verzögerung  $a_m$
- Maximaler Betrag der Geschwindigkeit  $v_m$
- Betrachteter Zeitpunkt  $t$
- Länge der Gesamtstrecke  $s_e$

Mithilfe dieser Parameter kann der Geschwindigkeitsverlauf einer Trajektorie und die Position eines Körpers bezogen auf die Zeit beschrieben werden (s. Abbildung 6-3). Für die Beschreibung des Verlaufs müssen jedoch zuerst Randbedingungen festgelegt werden. Folgende Randbedingungen sind zunächst einzuhalten:

- Start- und Zielzustand sind bekannt
- Geschwindigkeit zu Beginn ist Null
- Geschwindigkeit am Ende ist Null
- Geschwindigkeiten und Beschleunigungen sind begrenzt

Um nun eine Punkt-zu-Punkt-Steuerung für ein Fahrzeug zu realisieren bzw. die Punkte der Bahn zu realisieren, wird aus den bekannten bzw. festgelegten Parametern Geschwindigkeit, Beschleunigung, Startpunkt und Endpunkt die zu fahrende Strecke  $s_e$  berechnet. Anschließend werden die Angaben für die Geschwindigkeit und Beschleunigung modifiziert und die Gesamtzeit  $t_e$ ,  $t_a$ , und  $t_b$  berechnet. Auf dieser Basis erfolgt die Interpolation, sprich die Berechnung der Zwischen- und Sollwerte.

Zur Berechnung der Gesamtstrecke werden die Strecken der einzelnen Phasen getrennt ermittelt. In der ersten Phase ergeben sich folgende Gleichungen für die Beschleunigung, die Geschwindigkeit und die Strecke:

$$\begin{aligned} 0 &\leq t \leq t_a \\ \ddot{s}(t) &= a_m \\ \dot{s}(t) &= a_m * t + \dot{s}(0) \text{ mit } \dot{s}(0) = 0 \end{aligned} \tag{6-9}$$

ergibt sich für die Geschwindigkeit

$$\begin{aligned} \dot{s}(t) &= a_m * t \\ s(t) &= \frac{1}{2} * a_m * t^2 + s(0) \text{ mit } s(0) = 0 \end{aligned} \tag{6-10}$$

ergibt sich für den Weg

$$s(t) = \frac{1}{2} * a_m * t^2 \tag{6-11}$$

Die zweite Phase ist die gleichmäßige Fahrt. Die gleichmäßige Fahrt beginnt nach der Beschleunigungszeit und endet, sobald die Zeit zum Abbremsen erreicht ist. Für diese Phase ist die Beschleunigung gleich Null. Zudem ist die Geschwindigkeit bekannt.

$$t_a \leq t \leq t_b$$

$$\ddot{s}(t) = a_m \tag{6-12}$$

$$\dot{s}(t) = v_m \tag{6-13}$$

Somit bleibt die Beschreibung des Weges. Der zurückgelegte Weg setzt sich zusammen aus dem Anteil der Fahrstrecke während der Beschleunigung und der Fahrstrecke mit gleichmäßiger Bewegung.

$$s(t) = v_m * (t - t_a) + s(t_a)$$

Aus der Gleichung 6-10 der ersten Phase kann die Beschleunigungszeit ermittelt werden:

$$\begin{aligned}\dot{s}(t) &= v_m = a_m * t_a \\ t_a &= \frac{v_m}{a_m}\end{aligned}\quad (6-14)$$

Damit ergibt sich für den Streckenverlauf:

$$\begin{aligned}s(t) &= v_m * \left( t - \frac{v_m}{a_m} \right) + \frac{1}{2} * a_m * t_a^2 \\ &= v_m * t - \frac{1}{2} * \frac{v_m^2}{a_m}\end{aligned}\quad (6-15)$$

Zuletzt bleiben die Verläufe für den Bremsvorgang in Phase 3. Der Bremsvorgang startet ab dem Zeitpunkt  $t_b$  und dauert dieselbe Zeit wie der Beschleunigungsvorgang. Daher kann die Zeit, ab der der Bremsvorgang beginnt, als Differenz zwischen der Gesamtzeit  $t_e$  und der Beschleunigungszeit  $t_a$  berechnet werden. Während dieser Phase liegt eine konstante negative Beschleunigung vor.

$$t_b \leq t \leq t_e \text{ mit } t_b = t_e - t_a$$

$$\ddot{s}(t) = -a_m$$

Die Gleichung für die Geschwindigkeit des Bremsvorgangs ergibt sich aus der Geschwindigkeit zum Beginn des Bremsvorgangs abzüglich des Produktes aus der Beschleunigung und der Zeitdifferenz zwischen dem betrachteten Zeitpunkt und dem Startzeitpunkt des Bremsvorgangs.

$$\dot{s}(t) = -a_m * (t - t_b) + \dot{s}(t_d) = -a_m * (t - t_b) + v_m\quad (6-16)$$

Für die Gleichung der Strecke wird die Fläche unter dem Rampenprofil betrachtet. Die gesamte Fläche unter der Kurve wird berechnet aus

$$A = v_m * (t_e - t_a)\quad (6-17)$$

Von dieser Fläche wird lediglich der letzte Teil vom Zeitpunkt  $t$  bis zum Ende abgezogen. Damit ergibt sich für die Strecke folgende Gleichung:

$$s(t) = v_m * (t_e - t_a) - \frac{a_m}{2} * (t_e - t)^2\quad (6-18)$$

Die gesamte Strecke kann ebenfalls mit der Formel für die Fläche berechnet werden. Es lässt sich die Fahrzeit für die Strecke bestimmen.

$$\begin{aligned}s(t_e) &= s_e = v_m * (t_e - t_a) \\ t_e &= \frac{s_e}{v_m} + t_a = \frac{s_e}{v_m} + \frac{v_m}{a_m}\end{aligned}\quad (6-19)$$

Bei der Bahnplanung wird zudem zwischen asynchroner und synchroner Bahnplanung unterschieden. Das bedeutet, wenn mehrere Achsen vorliegen oder die Bewegung sich in mehrere Teile untergliedert, kann es passieren, dass die einzelnen Achsen nicht gleichzeitig am Ziel ankommen. Beim Fahrzeug ist das z.B. die Drehung des Fahrzeuges und die translatorische Bewegung. Bei einer asynchronen Bewegung ist die Orientierung bzw. der Zielwinkel beispielsweise früher erreicht als die Zielposition. Sie enden somit unabhängig voneinander. Bei einer synchronen Bewegung würde Orientierung und Fahrzeug gleichzeitig am Ziel erreicht werden. Sie beginnen und beenden ihre Bewegung zum gleichen

Zeitpunkt. Die Fahrzeit der einzelnen Achsen bzw. Richtung und Position müssen folglich übereinstimmen. Als Wert für die gesamte Zeit wird der Wert angenommen, welcher der Maximale ist. Die Geschwindigkeiten der übrigen Achsen o.ä. werden dementsprechend angepasst. Durch Umstellen der Formel für die gesamte Fahrzeit 6-19, ergeben sich die neuen Geschwindigkeiten der einzelnen Komponenten.

$$v_{m,i} = \frac{a_{m,i} * t_e}{2} - \sqrt{\frac{a_{m,i}^2 * t_e^2}{4} - s_{e,i} * a_{m,i}} \quad (6-20)$$

Die Interpolation der einzelnen Punkte erfolgt letztlich anhand der einzelnen Gleichungen:

$$s(t) = \begin{cases} \frac{1}{2} * a_{m,i} * t^2 & 0 \leq t \leq t_a \\ v_{m,i} * (t - \frac{1}{2} * \frac{v_{m,i}}{a_{m,i}}) & t_a \leq t \leq t_b \\ \frac{1}{2} * \frac{v_{m,i}^2}{a_{m,i}} + v_{m,i} * (t_b - t_a) + v_{m,i} * (t - t_b) - a_{m,i} * (t - t_b)^2 * \frac{1}{2} & t_b \leq t \leq t_e \end{cases} \quad (6-21)$$

$$v(t) = \begin{cases} a_{m,i} * t & 0 \leq t \leq t_a \\ v_{m,i} & t_a \leq t \leq t_b \\ v_{m,i} - a_{m,i} * (t - t_b) & t_b \leq t \leq t_e \end{cases} \quad (6-22)$$

### 6.2.3 Transformation zwischen den Koordinatensystemen

Zur Beschreibung der Position und der Bewegung des Fahrzeugs sind Koordinatensysteme erforderlich. Das System verfügt über zwei verschiedene Koordinatensysteme, zum einen ein körperfestes Koordinatensystem, zum anderen ein Bezugskoordinatensystem. Das körperfeste Koordinatensystem befindet sich in der Fahrzeugmitte. Das Bezugskoordinatensystem wird durch das Trackingsystem vorgegeben. Dieses wird bei der Kalibrierung der Trackingkamera ermittelt. Die Abbildung 6-4. stellt die beiden Koordinatensysteme in der Ebene dar. Es wird ersichtlich, dass die z-Achse des körperfesten Koordinatensystems orthogonal in den Boden zeigt. Das Bezugskoordinatensystem liegt hingegen umgekehrt im Raum. Um eine Fahrzeugbewegung zwischen der Ist-Position und der Sollposition durchzuführen, benötigt das Fahrzeug die Informationen über die Fahrtrichtung, Drehrichtung und Geschwindigkeit. Diese beziehen sich auf das körperfeste Koordinatensystem des Fahrzeugs. Die Ist-Position und die Sollposition des Fahrzeugs werden vom Trackingsystem allerdings in Bezug auf das Bezugskoordinatensystem ausgegeben. Zur Ermittlung der Informationen für die Fahrzeugbewegung ist daher eine Koordinatentransformation erforderlich.

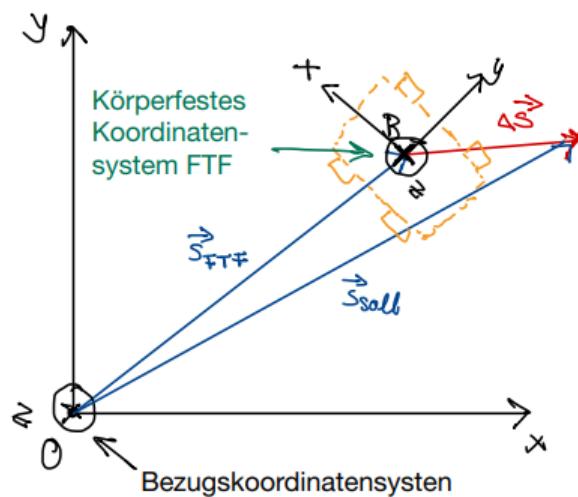


Abbildung 6-4: Koordinatensysteme

Die Transformation der Koordinatensysteme erfolgt über einen Verschiebungsvektor und Elementardrehungen. Dabei soll das Bezugskoordinatensystem in Einklang mit dem körperfesten Koordinatensystem des FTF gebracht werden. Um dies zu erreichen werden zwei Elementardrehungen benötigt, zuerst eine Drehung um die z-Achse, anschließend eine Drehung um die z-Achse. Für die Elementardrehungen gelten folgende Formeln (vgl. Hofschulte, 2022: S.69):

$$R_z = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6-23)$$

$$R_y = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \quad (6-24)$$

Der Wert für die Elementardrehung um die z-Achse entspricht der Orientierung des Fahrzeugs  $\alpha$ . Die Elementardrehung um die y-Achse wird gebraucht, um die Richtung der z-Achse umzukehren, sodass diese ebenfalls orthogonal in den Boden zeigt. Dazu wird das Koordinatensystem um  $180^\circ$  um die y-Achse gedreht. Durch Einsetzen der  $180^\circ$  in die Matrix für die Elementardrehung um die y-Achse ergibt sich folgende Matrix:

$$R_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Die Elementardrehungen können in eine einzelne Rotationsmatrix zusammengefasst werden. Hierbei ist die Reihenfolge der Multiplikation entscheidend. Da erst um die z-Achse und danach um die y-Achse rotiert wird, muss eben diese Reihenfolge bei der Multiplikation eingehalten werden.

$$R_{ges} = R_z * R_y = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -\cos(\alpha) & -\sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad (6-25)$$

Nachdem zunächst die Rotation betrachtet wurde, wird nun ein Blick auf die gesamte Transformation geworfen. Für die Transformation ergibt sich folgende Formel

$$\overrightarrow{s_{soll}} = \overrightarrow{s_{FTF}} + R_{ges} * \overrightarrow{\Delta s} \quad (6-26)$$

Hierbei ist die unbekannte Größe  $\overrightarrow{\Delta s}$ . Diese Formel kann ebenfalls in Koordinatenschreibweise formuliert werden.

$$\begin{pmatrix} x_{soll} \\ y_{soll} \\ 0 \end{pmatrix} = \begin{pmatrix} x_{FTF} \\ y_{FTF} \\ 0 \end{pmatrix} + \begin{bmatrix} -\cos(\alpha) & -\sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & -1 \end{bmatrix} * \begin{pmatrix} \Delta s_x \\ \Delta s_y \\ 0 \end{pmatrix}$$

Da eine Fahrzeugbewegung in der Ebene betrachtet wird, nimmt die z-Koordinate den Wert Null an. Eine weitere Form der Darstellung dieser Gleichung ist mit Hilfe der homogenen Transformationsmatrix. Die homogene Transformationsmatrix besteht aus der Rotationsmatrix und dem Positionsvektors des Fahrzeugs. (vgl. Hofschulte, 2022: S.104)

$${}^0H_{FTF} = \begin{bmatrix} R_{ges} & \overrightarrow{s_{FTF}} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -\cos(\alpha) & -\sin(\alpha) & 0 & x_{FTF} \\ -\sin(\alpha) & \cos(\alpha) & 0 & y_{FTF} \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6-27)$$

Daraus ergibt sich für die Gleichung

$$\begin{pmatrix} x_{soll} \\ y_{soll} \\ 0 \\ 1 \end{pmatrix} = \begin{bmatrix} -\cos(\alpha) & -\sin(\alpha) & 0 & x_{FTF} \\ -\sin(\alpha) & \cos(\alpha) & 0 & y_{FTF} \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} \Delta s_x \\ \Delta s_y \\ 0 \\ 1 \end{pmatrix} \quad (6-28)$$

Wie zuvor erwähnt, ist die unbekannte Größe hierbei  $\overrightarrow{\Delta s}$ . Daher muss die Gleichung nach dieser Größe umgestellt werden. Dafür ist die Berechnung der inversen homogenen Transformationsmatrix erforderlich. Diese besteht aus der transponierten Rotationsmatrix sowie dem Produkt aus negativ transponierter Rotationsmatrix und dem Positionsvektor des Fahrzeugs. (vgl. Hofschulte, 2022: S.104)

$${}^0H_{FTF}^{-1} = {}^{FTF}H_0 = \begin{bmatrix} {}^{R_{ges}}^T & -{}^{R_{ges}}^T * \overrightarrow{s_{FTF}} \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \quad (6-29)$$

Mit

$${}^{R_{ges}}^T = \begin{bmatrix} -\cos(\alpha) & -\sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

und

$$-{}^{R_{ges}}^T * \overrightarrow{s_{FTF}} = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ \sin(\alpha) & -\cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x_{FTF} \\ y_{FTF} \\ 0 \end{pmatrix} = \begin{pmatrix} x_{FTF} * \cos(\alpha) + y_{FTF} * \sin(\alpha) \\ x_{FTF} * \sin(\alpha) - y_{FTF} * \cos(\alpha) \\ 0 \end{pmatrix}$$

ergibt sich die inverse homogene Transformationsmatrix zu

$${}^0H_{FTF}^{-1} = {}^{FTF}H_0 = \begin{bmatrix} -\cos(\alpha) & -\sin(\alpha) & 0 & x_{FTF} * \cos(\alpha) + y_{FTF} * \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) & 0 & x_{FTF} * \sin(\alpha) - y_{FTF} * \cos(\alpha) \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die gesuchte Wegdifferenz bezogen auf das Fahrzeug wird nun durch folgende Gleichung bestimmt:

$$\begin{pmatrix} \Delta s_x \\ \Delta s_y \\ 0 \\ 1 \end{pmatrix} = \begin{bmatrix} -\cos(\alpha) & -\sin(\alpha) & 0 & x_{FTF} * \cos(\alpha) + y_{FTF} * \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) & 0 & x_{FTF} * \sin(\alpha) - y_{FTF} * \cos(\alpha) \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x_{soll} \\ y_{soll} \\ 0 \\ 1 \end{pmatrix} \quad (6-30)$$

Durch Ausmultiplizieren dieser Formel folgt daraus der Vektor für die Wegdifferenz.

$$\begin{pmatrix} \Delta s_x \\ \Delta s_y \\ 0 \end{pmatrix} = \begin{pmatrix} -\cos(\alpha) * x_{soll} - \sin(\alpha) * y_{soll} + x_{FTF} * \cos(\alpha) + y_{FTF} * \sin(\alpha) \\ -\sin(\alpha) * x_{soll} + \cos(\alpha) * y_{soll} + x_{FTF} * \sin(\alpha) - y_{FTF} * \cos(\alpha) \\ 0 \end{pmatrix}$$

Das Ziel der Koordinatentransformation ist erreicht. Nun kann der Vektor der Fahrbewegung in Bezug auf das Fahrzeug angegeben werden.

### 6.3 Programmcodes Sprint #2

Für die Bewegungssteuerung des FTS werden Programmcodes seitens *Arduino* und seitens Python verwendet. Mit Hilfe des Python-Skriptes sollen Trackingdaten des Fahrzeugs verarbeitet und in Fahrbefehle umgewandelt werden. Diese Daten werden über eine serielle Schnittstelle an das Fahrzeug übergeben, wodurch die Fahrbewegung ausgeführt wird. Das Python-Skript soll dabei eine Funktion zur Lageregelung und Trajektorienplanung einer Punkt-zu-Punkt-Bewegung integrieren. In den folgenden Abschnitten wird daher näher auf die einzelnen Programmcodes eingegangen.

#### 6.3.1 Programmcode für Fahrbewegung in *Arduino*

Das *Arduino Board* ist für die Ansteuerung der Motoren und somit für die Ausführung der Fahrbewegung verantwortlich. Damit das Board die richtigen Anweisungen an die Motoren weiterleiten kann, ist es erforderlich einen entsprechenden Programmcode auf das Board zu laden. Dieser Programmcode gibt die Fahrrichtung des Fahrzeugs vor und regelt die Geschwindigkeit. Dem *Sprint Backlog* ist zu entnehmen, dass eine Überlagerung der Fahrbewegung gefordert ist. Das bedeutet das Fahrzeug soll in eine bestimmte Richtung fahren können und gleichzeitig soll eine Drehbewegung des Fahrzeuges realisiert werden.

Die bisher betrachteten Programmcodes sind so formuliert, dass keine Überlagerung möglich ist. Sie arbeiten durch Setzen eines bestimmten Status. Wird z.B. der Status auf „Vorwärts“ gesetzt, so wird eine Vorwärtsbewegung ausgeführt. Für die Fahrtbewegung selbst wird der Befehl *Omni.setCarMove* verwendet. Dieser Befehl verwendet Informationen über die Geschwindigkeit, die Fahrrichtung und die Drehgeschwindigkeit und verarbeitet sie entsprechend für die Ansteuerung der Motoren. Die Informationen werden an den *Arduino* über die serielle Schnittstelle des Bluetooth-Moduls übergeben. Der Programmcode 6-1 zeigt wie dies in dem *Arduino-Sketch* umgesetzt wird.

```

79 void loop() {
80   while(Serial.available()==0){
81     }
82
83   cmd=Serial.readStringUntil('\n');
84   i=cmd.indexOf(':');
85   speed=cmd.substring(0,i).toInt();
86   j=cmd.indexOf(':',i+1);
87   direction=cmd.substring(i+1,j).toFloat();
88   k=cmd.indexOf('\n');
89   rotation=cmd.substring(j+1,k).toFloat();
90
91   Omni.setCarMove(speed,direction,rotation);
92 }
```

Programmcode 6-1: Auslesen und Verarbeiten von seriellen Daten zur Bewegungsausführung

Zunächst wird solange keine Aktion durchgeführt bis Daten über die serielle Schnittstelle übertragen werden. Der *Arduino* erhält dann über die serielle Schnittstelle Daten und liest diese bis zu einem Charakter „\n“ aus (vgl. Kapitel 5.3.3). Der erhaltene Datensatz wird der Variable Kommando „cmd“ zugeordnet. Das Kommando kann beispielsweise folgender Maßen aussehen:

cmd=200:1.5783:0.0000\n

Die erste Zahl ist hierbei die Geschwindigkeitsangabe. Nach dem Doppelpunkt folgt die Richtungsangabe in rad, dann wiederum nach dem nächsten Doppelpunkt kommt die Information über die Drehgeschwindigkeit. Letztlich wird das Kommando mit „\n“ abgeschlossen. Der Charakter „\n“ gibt immer das Ende eines übermittelten Datensatzes an. Es ist erforderlich, damit eindeutig ist, bis wohin die Daten ausgelesen werden sollen.

Um die Informationen über Geschwindigkeit, Fahrtrichtung und Drehgeschwindigkeit weiter verarbeiten zu können, wird das Kommando geteilt. Die getrennten Kommandoteile werden jeweils einer Variablen zugeordnet. Die Trennung des Datensatzes erfolgt über den Befehl *substring*. Wie der Befehl angibt, liegen die Daten weiterhin in Form von *strings* vor. Der Befehl *setCarMove* arbeitet allerdings mit *Integer*- oder *Float*-Zahlen. Das bedeutet es ist eine Umwandlung in eine Zahl erforderlich. Dies wird mit den Funktionen „*toInt*“ bzw. „*toFloat*“ durchgeführt. Nun können die Zahlenwerte dem Befehl *serCarMove* übergeben werden und die Bewegung kann ausgeführt werden.

Damit Daten über die serielle Schnittstelle empfangen werden können, muss zu Anfang eine serielle Verbindung aufgebaut werden. Dies wird in der Funktion *void setup()* realisiert (s. Programmcode 6-2). Die Baudrate für die Datenübertragung wird nochmals auf 57600baud erhöht.<sup>5</sup> Die Erhöhung ist notwendig, da vom Python-Skript eine erhebliche Anzahl an Bit pro Sekunde gesendet werden soll. Damit keine Daten verloren gehen, muss die Datenübertragung ausreichend schnell sein. Mit 56700baud ist das Bluetooth-Modul so eingestellt, dass dies der Fall ist.

```

65 void setup() {
66   pinMode(13,OUTPUT);
67   Serial.begin(57600);
68
69   delay(2000);
70   TCCR1B=TCCR1B&0xf8|0x01;      // Pin9,Pin10 PWM 31250Hz
71   TCCR2B=TCCR2B&0xf8|0x01;      // Pin3,Pin11 PWM 31250Hz
72
73   OCR0A=100;
74   TIMSK0 |=2;
75
76   Omni.PIDEnable(0.35,0.02,0,10);

```

Programmcode 6-2: Funktion *void setup()* für Bewegungssteuerung

Neben dem Starten der seriellen Schnittstelle wird im *void setup()* des Weiteren eine LED über *pinMode(13, OUTPUT)* deklariert. Bei der LED handelt es sich um eine im *Board* integrierte LED, welche über den Pin 13 aufgerufen wird. Die LED soll als grafische Ausgabe dienen und hängt mit der Regelung der Geschwindigkeit zusammen.

In den bisherigen Kapiteln erfolgte die Regelung der Geschwindigkeit innerhalb der *void loop()*. Aus dem Programmcode 6-1 ist ersichtlich, dass der Befehl für die Regelung nun nicht mehr in der Schleife enthalten ist. Zuvor wurde die Regelung immer dann aktiviert, wenn ein neuer Schleifen-Durchlauf erfolgt. Dadurch wird die Regelung allerdings zu selten aufgerufen und das Fahrzeug neigt dazu zu schwingen. Dementsprechend muss die Regelung häufiger durchgeführt werden. Sie wird daher in einen *Timer* integriert. *Timer1* und *Timer2* werden bereits für die Motoren verwendet (vgl. Kapitel 3.2.2). Daher soll die Regelung in eine *Interrupt Service Routine ISR* des *Timer 0* integriert werden. Die *ISR* wird immer dann ausgeführt, wenn der Wert von *Output Compare Register OCR0A* mit dem Zählerwert des *Timer 0* (*TCNT0*) übereinstimmt. Im *void setup()* wird der Wert im Register *OCR0A* auf 100 gesetzt. Das bedeutet immer, wenn der Timer den Compare Wert 100 erreicht, wird die Regelung ausgeführt. Die Auslösefrequenz ist dabei abhängig davon wie der Timer eingestellt wird (vgl. Kapitel 3.2.2).

Damit das Blinken der LED als grafische Ausgabe für den Anwender erkennbar ist, soll diese nicht bei jedem Aufruf der *ISR* ausgelöst werden. Mit Hilfe einer Zählvariablen *count* wird die Leuchtfrequenz der LED reduziert. Die Zählvariable wird bei jedem Aufruf der *ISR* um Eins hochgezählt. Sobald sie einen Wert von 100 erreicht wird die LED aktiviert. Das bedeutet die LED blinkt 100mal so langsam wie die Regelung aufgerufen wird (s. Programmcode 6-3).

---

<sup>5</sup> In Kapitel 5.3.2 ist ausgeführt wie die Erhöhung der Baudrate für das Bluetooth-Modul funktioniert

```

52 int count=0;
53 ISR(TIMER0_COMPA_vect)
54 {
55
56     count++;
57     if (count==100) {
58         digitalWrite(13,!digitalRead(13));
59         count=0;
60     }
61
62     Omni.PIDRegulate();
63 }
```

Programmcode 6-3: Interrupt Service Routine für Timer 0

Die wesentlichen Aufgaben des Programmcodes wurden nun betrachtet. Neben diesen Aufgaben werden außerdem die Motoren deklariert, die entsprechenden Pins zugewiesen, die PWM-Frequenz eingestellt etc. Da diese Teile des Programmcodes ähnlich zu dem Demoskript bzw. identisch zum Skript „Datenaustausch“ aus dem ersten Sprint sind, werden diese hier nicht nochmal erläutert. Der vollständige Programmcode ist im Anhang D zu finden.

### 6.3.2 Programmcode für Lageregelung

Eine Anforderung des *Sprint Backlog* ist die Realisierung einer Lageregelung des Fahrzeugs. Das Kapitel 6.2.1 thematisiert dazu die theoretischen Grundlagen einer Lagereglung sowie die Herleitung der Formeln. Dieses Kapitel beschäftigt sich nun mit der Umsetzung der Lagerregelung in Form eines Python-Skriptes.

Für die Lageregelung wird eine neue Funktion „Lageregelung“ definiert. Durch Aufrufen dieser Funktion wird die Lagerregelung durchgeführt. Die Funktion Lageregelung benötigt die Positions Werte  $x_{soll}$  und  $y_{soll}$  sowie die Sollorientierung  $\phi_{soll}$ . Diese Werte werden der Funktion beim Aufrufen übergeben. Innerhalb der Funktion werden zunächst die Werte für die Proportionalregler definiert. Dabei werden zwei getrennte Werte für die translatorische und rotatorische Bewegung unterschieden. Die Parameter werden in dieser Arbeit experimentell bestimmt. Nachdem das Skript vollständig formuliert ist, werden die Parameter durch Testen ermittelt. Dabei sollen sie so eingestellt werden, dass sie nicht zu langsam auf Regelabweichungen reagieren. Gleichzeitig soll das Fahrzeug nicht zu stark überschwingen.

Im nächsten Schritt werden die Daten von dem Trackingsystem über einen http-Request angefragt und die Werte der zurückgegebenen JSON werden den Variablen  $x_{FTS}$ ,  $y_{FTS}$  und  $\phi_{FTS}$  zugeordnet. Diese Werte geben Auskunft über die Position und Orientierung des Fahrzeugs im Bezugskoordinatensystem (s. Programmcode 6-4).

```

8 def Lageregelung(x_soll,y_soll,phi_soll):
9     kp=1.2
10    k_phi=0.75
11
12    r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
13    obj = json.loads(r.text)
14    x_FTS = float(obj["x"])
15    y_FTS = float(obj["y"])
16    phi_FTS = float(obj["phi"])
17
18    alpha=phi_FTS-math.pi/2
```

Programmcode 6-4: Lageregelung „Auslesen von Trackingdaten und Korrektur des Winkels“

Nachdem die Werte von der Trackingkamera ausgelesen wurden, ist es erforderlich den Winkel nochmal umzurechnen. Das liegt daran, dass das Trackingsystem beim Erkennen zweier Marker ein Koordinatensystem für einen Körper erstellt. Dabei zeigt das Koordinatensystem mit der x-Achse in Richtung kleineren Markers (s. Abbildung 6-5). Bei dem Fahrzeug sind die Marker jedoch so positioniert, dass der kleinere Marker auf der y-Achse liegt (s. Abbildung 6-6). Es liegt demnach eine Differenz von  $\pi/2$  vor. Diese wird durch Umrechnung innerhalb des Skriptes korrigiert.

**Positionsmarker**  
Position:  $(x, y, \varphi)$

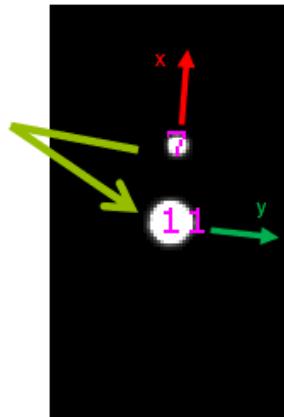


Abbildung 6-5: Markerposition in Trackingsystem  
(Hofschulte & Waldt, 2021: S10)

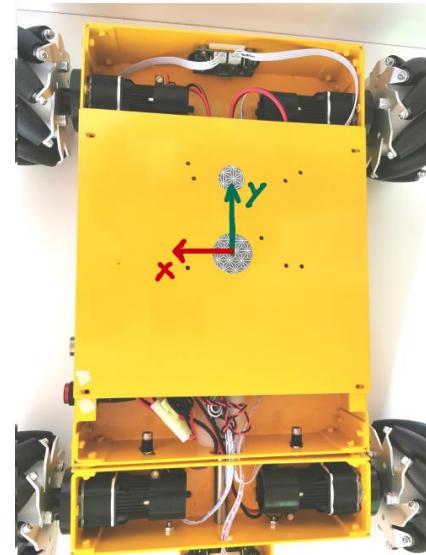


Abbildung 6-6: Markerposition auf Fahrzeug

Das System kennt die Positions- und Orientierungswerte für den Sollzustand. Für den Ist-Zustand allerdings beziehen sich diese auf das Bezugskoordinatensystem des Trackingsystems. Damit jedoch die Fahrtrichtung kalkuliert werden kann, muss die Soll-Position auf das körperfeste Koordinatensystem des Fahrzeugs umgerechnet werden (vgl. Kapitel 6.2.3). Aus der Transformation der Koordinatensysteme ergibt sich der Vektor für die Soll-Positionen bezogen auf das Fahrzeug zu

$$\begin{pmatrix} \Delta s_x \\ \Delta s_y \\ 0 \end{pmatrix} = \begin{pmatrix} -\cos(\alpha) * x_{soll} - \sin(\alpha) * y_{soll} + x_{FTF} * \cos(\alpha) + y_{FTF} * \sin(\alpha) \\ -\sin(\alpha) * x_{soll} + \cos(\alpha) * y_{soll} + x_{FTF} * \sin(\alpha) - y_{FTF} * \cos(\alpha) \\ 0 \end{pmatrix}$$

Dieser Vektor wird im Skript aufgegriffen, wobei übersichtshalber die Form geändert wird. Der Vektor wird so umgestellt, dass der direkte Zusammenhang zwischen der Soll-Position und der Ist-Position ersichtlich wird (s. Programmcode 6-5).

```
31     #Ziel bezogen auf FTS
32     dx=-math.cos(alpha)*(x_soll-x_FTS)-math.sin(alpha)*(y_soll-y_FTS)
33     dy=-math.sin(alpha)*(x_soll-x_FTS)+math.cos(alpha)*(y_soll-y_FTS)
```

Programmcode 6-5: Lageregelung „Umrechnung der Soll-Position in Bezug auf das Fahrzeug“

Im nächsten Schritt wird die Geschwindigkeit für die Regelung berechnet. Zunächst werden die jeweiligen Geschwindigkeiten in x-Richtung und in y-Richtung benötigt. Diese ergeben sich aus den Wegdifferenzen und dem Proportionalitätsfaktor  $k_p$  (s. Programmcode 6-6). Die gesamte Geschwindigkeit berechnet sich anschließend aus der Quadratwurzel der Summe der einzelnen quadrierten Geschwindigkeiten. Nachdem diese ermittelt ist, kann sie verschiedene Werte annehmen u.a. einen Wert von 300mm/s. Da die maximale Geschwindigkeit des Fahrzeugs bei 255mm/s liegt, würde die Grenze überschritten werden. Daher ist eine Stellgrößenbegrenzung erforderlich. Diese wird mit einer *if*-Abfrage realisiert. Liegt der Wert über 255mm/s, so wird dieser begrenzt auf 255mm/s, ansonsten bleibt der Wert wie gehabt. Letztlich ist noch die Fahrtrichtung der translatorischen Bewegung zu bestimmen. Dies erfolgt über die Funktion *arctan2*.

```

26 #Berechnung der Geschwindigkeit
27 v_x=kp*dx
28 v_y=kp*dy
29
30 v_ges=math.sqrt(v_x**2+v_y**2)
31
32 if (v_ges>255):
33     v_ges=255
34 else:
35     v_ges
36
37 #Berechnung der Fahrtrichtung
38 rad = math.atan2(dy,dx)
```

Programmcode 6-6: Lageregelung „Berechnung der Fahrgeschwindigkeit und der Fahrtrichtung“

Als nächstes steht die Rotationsbewegung im Fokus. Zunächst wird hierfür die Winkelabweichung zwischen der Soll-Orientierung und der Ist-Orientierung bestimmt. Für die Reduzierung der Winkelabweichung ist es notwendig, dass das Fahrzeug stets in die Richtung dreht, die zur Reduzierung des Regelfehlers führt.

Bei der Berechnung der Winkelabweichung kann es passieren, dass diese einen Wert größer als 180° oder kleiner als -180° annimmt. In Folge würde das Fahrzeug sich in eine Richtung drehen, die eine längere Zeit zum Ausgleichen der Regelabweichung benötigt. Wird z.B. eine Regelabweichung von -200° festgelegt, so dreht das Fahrzeug um 200° rechtsherum. Allerdings ist der Weg viel kürzer, wenn das Fahrzeug linksherum drehen würde. Bei Drehung linksherum beträgt die Abweichung lediglich 160°. Daher ist es erforderlich den Winkel in solchen Fällen zu korrigieren. Dies geschieht über eine *if*-Abfrage. Diese erfüllt genau die soeben beschriebene Funktion (s. Programmcode 6-7). Nachdem aus der *if*-Abfrage der Wert für die Winkelabweichung identifiziert ist, wird diese am Schluss mit dem Proportionalitätsfaktor  $k_{\phi}$  in eine Winkelgeschwindigkeit  $\omega$  umgerechnet.

```

40 #Berechnung der Winkeländerung und Winkelgeschwindigkeit omega
41 d_phi=(phi_soll-phi_FTS)
42
43 if (abs(d_phi)>math.pi):
44     if (d_phi>0):
45         d_phi=-2*math.pi+d_phi
46     else:
47         d_phi=2*math.pi+d_phi
48 else:
49     d_phi=d_phi
50
51 omega=d_phi*k_phi #[1/s]
```

Programmcode 6-7: Lageregelung „Berechnung der Winkelgeschwindigkeit“

Im letzten Teil der Funktion „Lageregelung“ werden die gewonnenen Informationen über Fahrgeschwindigkeit, Fahrrichtung und Drehgeschwindigkeit mit Hilfe des HC05-Moduls an den *Arduino* übermittelt. Dieser kann die Informationen in Fahrbefehle umwandeln, sodass das Fahrzeug seine Bewegung ausführt. Für die Übergabe der Daten werden diese in ein Kommando eingebettet. Hierbei werden die Nachkommastellen begrenzt, da ansonsten unnötig viele Bits bei der Datenübertragung verwendet werden und der zusendende Datensatz größer als notwendig ist. Daher werden die *Float*-Zahlen auf vier Nachkommastellen begrenzt. Die Fahrtrichtung wird vom *Arduino-Sketch* als *Integerzahl* verarbeitet, weshalb keine Nachkommastellen benötigt werden. Wie die genaue Datenübertragung funktioniert wird in Kapitel 5.3.3 thematisiert und deshalb an dieser Stelle nicht näher ausgeführt.

```

53  #Übergabe der Daten über serielle Schnittstelle
54  cmd="{:.0f}".format(v_ges)+":{:.5f}".format(rad)+":{:.5f}".format(omega)
55  cmd=cmd+'\n'
56  print(str(cmd))
57  arduinoData.write(cmd.encode())

```

Programmcode 6-8: Lageregelung „Übergabe der Fahrbefehle an *Arduino*“

Die Funktion der Legeregelung ist nun vollständig definiert. Zum Ausführen dieser Funktion muss diese aufgerufen werden. Dies geschieht in einer endlosen *while*-Schleife. Hier werden zudem die Parameter für die Position  $x_{soll}$  und  $y_{soll}$  sowie die Soll-Orientierung  $\phi_{soll}$  angegeben. Mit der Funktion *time.sleep()* wird ein Zeitintervall definiert, welches angibt, nach welchem Zeitraum die Lageregelung erneut durchgeführt werden soll. An dieser Stelle wird zunächst ein Wert von 0,5s angenommen.<sup>6</sup>

```

61  while True:
62      Lageregelung(-200,0,0)
63      time.sleep(0.5)

```

Programmcode 6-9: Lageregelung „Aufrufen der Lageregelung“

---

<sup>6</sup> Der vollständige Programmcode für die Lageregelung ist in Anhang E zu finden.

### 6.3.3 Programmcode für Trajektorienplanung in Python

Nachdem der *Arduino-Sketch* zum Ausführen der Fahrbewegung und die Lageregelung zur Korrektur von Lageabweichung implementiert sind, wird nun der Programmcode für die Trajektorienplanung zwischen zwei Punkten begutachtet. Dafür werden die hergeleiteten Formeln aus Kapitel 6.2.2 verwendet. Im Folgenden wird statt der Bezeichnung „Trajektorienplanung“ der Begriff „Bahnplanung“ genutzt. Dies ist auf eine anfängliche Gleichsetzung der beiden Begrifflichkeiten zurückzuführen. In der Literatur werden Bahn und Trajektorie fälschlicherweise oftmals gleichgesetzt. Daher wurde für den Programmcode die Bezeichnung „Bahnplanung“ eingeführt. Allerdings werden nicht nur Position und Orientierung interpoliert, sondern auch die zugehörigen Geschwindigkeiten. Daher erfolgt in diesem Kapitel eine Trajektorienplanung. Lediglich der Programmcode wird als Bahnplanung bezeichnet.

Im ersten Schritt des Bahnplanungsalgorithmus wird eine Zielposition und eine Zielorientierung vorgegeben. Aus diesen werden die Wegdifferenzen bzw. die Winkeldifferenz zu der Lage berechnet, bei der sich das Fahrzeug befindet. Aus den Wegdifferenzen in x-Richtung und y-Richtung ergibt sich weiterhin die gesamt zufahrende Strecke.

```

84  #####Bahnplanung#####
85  #####
86 def Bahnplanung(x_ziel, y_ziel, phi_ziel):
87     r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
88     obj = json.loads(r.text)
89     x_start = float(obj["x"])
90     y_start = float(obj["y"])
91     phi_start = float(obj["phi"])
92
93     print("Koordinaten FTS: " + str(x_start), str(y_start), str(phi_start))
94
95     s_x_gesamt=x_ziel-x_start
96     s_y_gesamt=y_ziel-y_start
97     s_gesamt=math.sqrt(s_x_gesamt**2+s_y_gesamt**2)
98     phi_gesamt=phi_ziel-phi_start

```

Programmcode 6-10: Bahnplanung „Streckenberechnung“

Als nächstes wird der Geschwindigkeitsverlauf der Trajektorienplanung festgelegt. Es soll ein Geschwindigkeitsprofil mit Rampenprofil programmiert werden (s. Programmcode 6-10). Dafür werden zunächst die Werte für Beschleunigung und Geschwindigkeit festgelegt. Dabei ist wichtig, dass diese keine maximalen Werte überschreiten, sprich für die translatorische Geschwindigkeit ist z.B. die Grenze für eine max. Geschwindigkeit von 255mm/s einzuhalten und diese darf nicht überschritten werden. Aus den bisherigen Werten ist die Zeit der einzelnen Bewegungen zu ermitteln, wobei die Zeit in x-Richtung und y-Richtung in eine gesamte Zeit für die translatorische Bewegung zusammengefasst wird. Für die synchrone Bahnsteuerung ist es wichtig, dass das Fahrzeug zur selben Zeit die Position und Orientierung erreicht. Daher wird verglichen, welche von den berechneten Zeiten, die maximale Zeit ist. Diese wird als gesamte Zeit für die Bahn festgelegt (s. Programmcode 6-11).

```

117 ...
118 v
119 |
120 |
121 | /| | \ \
122 | / | | \ \
123 | / | | \ \ t
124 | t_a t_b
125 ...
126
127 v_trans=200 #mm/s
128 a_trans=a_x=a_y=150 #mm/s^2
129 teta=2.5 #rad/s^2
130 omega=3 #rad/s
131
132 t_ges_trans=s_gesamt/v_trans+v_trans/a_trans
133 t_ges_phi=abs(phi_gesamt)/omega +omega/teta
134
135 #für synchrone Bewegung müssen alle Positionen gleichzeitig erreicht werden
136 #bestimmen, welche Komponente am längsten braucht und diese als neue Zeit festlegen
137 t_ges=max(t_ges_phi,t_ges_trans)

```

Programmcode 6-11: Bahnplanung „Berechnung der maximalen Zeit“

Sobald die neue Zeit festgelegt ist, werden daraus die neuen Geschwindigkeiten ermittelt. Dieses Mal wird zwischen den Geschwindigkeiten in x-Richtung und y-Richtung differenziert. Dies ist notwendig, um später die jeweiligen x-Koordinaten bzw. y-Koordinaten der interpolierten Punkte zu bestimmen. Des Weiteren werden die die Beschleunigungszeiten für jede Komponente sowie jene Zeit bestimmt, ab der das Abbremsen der Bewegung beginnt, um beim Ziel eine Geschwindigkeit von Null zu erzielen (s. Programmcode 6-12).

```

153 #Berechnen der neuen Geschwindigkeiten bezogen auf die neue Zeit
154 v_x_neu=((a_x*t_ges)/2-math.sqrt((a_x**2*t_ges**2)/4-abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt
155 v_y_neu=((a_y*t_ges)/2-math.sqrt((a_y**2*t_ges**2/4)-abs(s_y_gesamt)*a_y))*abs(s_y_gesamt)/s_y_gesamt
156 omega_neu=((teta*t_ges)/2-math.sqrt((teta**2*t_ges**2/4)-abs(phi_gesamt)*teta))*abs(phi_gesamt)/phi_gesamt
157
158 #Beschleunigungszeiten
159 t_ax=abs(v_x_neu)/a_x
160 t_ay=abs(v_y_neu)/a_y
161 t_a_phi=abs(omega_neu)/teta
162
163 > ....
171
172 #Zeit ab der abgebremst werden soll
173 t_bx=t_ges-t_ax
174 t_by=t_ges-t_ay
175 t_b_phi=t_ges-t_a_phi

```

Programmcode 6-12: Bahnplanung „Berechnung von Geschwindigkeiten und Zeiten“

Nachfolgend werden die letzten Vorbereitungen für die Interpolation getätigt. Dafür werden leere Listen aufgestellt. Zum einen werden Listen für die jeweiligen Werte der Koordinaten und des Winkels bereitgestellt, zum anderen Listen für die Geschwindigkeitsanteile. In dem Programmcode 6-13 sind die beiden leeren Listen für den x-Anteil der Soll-Positionen und die Geschwindigkeiten in x-Richtung bei den einzelnen Soll-Positionen beschrieben. Neben den Listen wird eine Zeit „*delta\_t*“ festgelegt. Diese Zeit gibt an, nach wie vielen Sekunden der nächste Interpolationspunkt ermittelt werden soll. Zudem wird der Ist-Position ein Zeitwert von 0s zugewiesen. Aus der Intervallzeit „*delta\_t*“ berechnet sich wiederum die Anzahl an interpolierten Punkten, welche durchfahren werden sollen. Dabei ergibt sich zumeist keine gerade Zahl. Daher muss diese gerundet werden. Das Problem, dass dabei entsteht ist, dass zwischen dem letzten interpolierten Punkt und dem Zielpunkt eine geringere oder größere Zeitspanne liegt als zwischen den übrigen Punkten. Dieses Problem wird gelöst, indem das Zeitintervall „*delta\_t*“ korrigiert wird. Die Zeit „*delta\_t*“ ergibt sich nun aus der gesamten Fahrzeit dividiert durch die

Anzahl der zu interpolierenden Punkte. Der Vorteil dieser Doppelten Wertzuweisung liegt darin, dass das Zeitintervall immer einen ähnlichen Wert aufweist, hier bspw. um 0,1s herum. Das ist nützlich für die Lageregelung, welche in die Bahnplanung integriert werden soll, denn die Proportionalitätsparameter sind am besten auf ein definiertes Zeitintervall einzustellen. Würde nun von vornherein eine Anzahl an interpolierten Punkten festgelegt werden ohne die Länge der gesamten Strecke zu berücksichtigen, würde sich ein Zeitintervall abhängig von der Streckenlänge ergeben. Dadurch können die Interpolationszeiten stark variieren.

```

176     x_soll=[]
177     vx_soll=[]
178 >     """
179     ...
180     t_=[]
181
182     delta_t=0.1 #Intervallzeit
183     t=0 #Startzeitpunkt
184
185     n=int(round((t_ges/delta_t),0))
186     delta_t=t_ges/n

```

Programmcode 6-13: Bahnplanung „Listen für Interpolation“

Da bereits alle Vorbereitungen für die Interpolation erledigt sind, schließt sich die Interpolation der einzelnen Punkte an. Für die Interpolation wird eine neue Funktion „Interpolation“ eingeführt (s. Programmcode 6-14). Diese bekommt die Zeit übergeben, welche den Interpolationszeitpunkt angibt. Zudem werden Beschleunigungszeit, Bremszeit, Beschleunigung und Geschwindigkeit der Funktion zugewiesen. Innerhalb der Funktion wird zunächst überprüft, ob es sich um eine negative Geschwindigkeit handelt. In diesem Fall erhält die Beschleunigung nämlich ebenfalls ein negatives Vorzeichen. Im Anschluss werden die Gleichungen für die Interpolation der Position und der Geschwindigkeit definiert. Diese unterscheiden sich abhängig von der Phase (vgl. Kapitel 6.2.2). Zuletzt erfolgt eine Rückgabe der berechneten Position und Geschwindigkeit.

```

71     def Interpolation(t,t_a,t_b,a,v):
72         if v<0:
73             a=-a
74         else:
75             a
76
77         if(t <=t_a):
78             x=0.5*a*(t)**2
79             v_neu=a*t
80
81         if(t_a<t<=t_b):
82             x=v*t-0.5*v**2/a
83             v_neu=v
84
85         if(t>t_b):
86             x=0.5*a*(v/a)**2+v*(t_b-t_a)+v*(t-t_b)-0.5*a*(t-t_b)**2
87             v_neu=v-a*(t-t_b)
88
89         return [x,v_neu]

```

Programmcode 6-14: Bahnplanung "Funktion für Interpolation"

Die Funktion „Interpolation“ wird nun in eine for-Schleife integriert (s. Programmcode 6-15). Innerhalb der for-Schleife werden die x-Koordinaten, y-Koordinaten und die Orientierung interpoliert. Dafür wird jeweils die Funktion „Interpolation“ aufgerufen und die Rückgabewerte der Funktion einer zurückgelegten Strecke und einer Geschwindigkeit zugeordnet. Am Beispiel für die x-Koordinate sind das die Variablen `x_t` und `vx_soll_neu`. Die x-Koordinate zum betrachteten Zeitpunkt `x_soll_neu` ergibt sich dann wiederum aus der zurückgelegten Strecke `x_t` plus der Startkoordinate. Zuletzt werden die ermittelten Werte in die Listen für die interpolierten Koordinaten bzw. Orientierung und Geschwindigkeiten hinzugefügt. Zudem wird die betrachtete Zeit um „`delta_t`“ erhöht. Die for-Schleife wird solange durchlaufen bis die Variable `z` erreicht ist. Diese Variable ergibt sich aus der Anzahl der interpolierten Punkte `n` plus 1. Damit werden nicht nur die interpolierten Punkte selbst der Liste hinzugefügt, sondern ebenfalls der Start- und Zielpunkt.

```

195     z=n+1
196
197     for w in range(z):
198         [x_t, vx_soll_neu]=Interpolation(t,t_ax,t_bx,a_x,v_x_neu)
199         x_soll_neu=x_t+x_start
200         x_soll.append(x_soll_neu)
201         vx_soll.append(vx_soll_neu)
202
203         [y_t, vy_soll_neu]=Interpolation(t,t_ay,t_by,a_y,v_y_neu)
204         y_soll_neu=y_t+y_start
205         y_soll.append(y_soll_neu)
206         vy_soll.append(vy_soll_neu)
207
208         [phi_t, omega_soll_neu]=Interpolation(t,t_a_phi,t_b_phi,teta,omega_neu)
209         phi_soll_neu=phi_t+phi_start
210         phi_soll.append(phi_soll_neu)
211         omega_soll.append(omega_soll_neu)
212
213         #t_.append(t)
214         t+=delta_t

```

Programmcode 6-15: Bahnplanung "Bestimmen der interpolierten Punkte und Geschwindigkeiten"

Zur Verdeutlichung wird das Durchspielen eines Durchlaufs durch die `for`-Schleife anhand der x-Koordinate beschrieben. Zunächst hat die Zeit  $t$  den Wert Null. Das bedeutet innerhalb der Funktion „Interpolation“ werden in der ersten `if`-Schleifen Geschwindigkeit und Strecke zum Zeitpunkt  $t=0$  ermittelt. Diese Werte werden zurückgegeben und der ermittelte Wert für die Strecke in x-Richtung wird mit dem Startwert des Fahrzeuges addiert. Der Startwert liegt immer bei der anfangs getrackten Position des Fahrzeuges. Dadurch ergibt sich für den ersten Wert, der in die Liste hinzugefügt werden soll, der Startwert des Fahrzeuges. Im Anschluss wird die Zeit  $t$  um die Intervallzeit „`delta_t`“ erhöht und der nächste Durchlauf durch die `for`-Schleife beginnt.

Nachdem die interpolierten Positionen und Orientierungen bekannt sind, sollen diese nun abgefahren werden. Dies wird durch eine `while`-Schleife inklusive `if`-Abfrage realisiert. Zu Beginn wird eine Laufvariable `k` festgelegt. Diese soll die Listen mit den Interpolationen durchlaufen. Durch die `if`-Abfrage wird sichergestellt, dass nur solang neue Interpolationen aus den Listen ausgelesen werden, wie die Liste lang ist. Innerhalb der `if`-Abfrage wird zuerst die Lageregelung abgerufen. Die Lageregelung bekommt die Werte für Soll-Position und Soll-Orientierung gegeben. Sie soll daraus eine Regelabweichung bestimmen und Werte für die zusätzlichen Geschwindigkeiten zurückgeben.

```

251     k=0
252     while True:
253         if (k<(z-1)):
254
255             d_v=Lageregelung(x_soll[k],y_soll[k],phi_soll[k])
256             print("Werte aus Lageregelung: " +str(d_v))
257             d_v_x=d_v[0]
258             d_v_y=d_v[1]
259             d_omega=d_v[2]

```

Programmcode 6-16: Bahnplanung "Durchlaufen der Lageregelung"

Dafür ist es erforderlich die Lageregelung, die in Kapitel 6.3.2 entworfen wurde nochmals geringfügig umzugestalten. Im Kapitel 6.3.2 ist die Lageregelung so gestaltet, dass die Weg- und Winkeldifferenzen berechnet wurden. Hieraus wurden dann die Geschwindigkeiten und Fahrtrichtung berechnet und diese über eine serielle Schnittstelle an das Fahrzeug übermittelt. Nun sollen die Informationen anstatt sie über eine Schnittstelle weiterzuleiten als Rückgabe „*return*“ der Funktion zurückgegeben werden. Es werden Geschwindigkeit in x-Richtung, Geschwindigkeit in y-Richtung und Winkelgeschwindigkeit als Werte zurückgegeben. Diese Rückgabe wird als eine Liste ausgegeben und kann innerhalb der Bahnplanung verwendet werden. In dem Programmcode 6-16 wird die Rückgabe der Variablen „*d\_v*“ zugeordnet und anschließend die einzelnen Elemente der Liste, sprich die Geschwindigkeit in x-Richtung, die Geschwindigkeit in y-Richtung und die Winkelgeschwindigkeit, neuen Variablen zugewiesen.

Die Werte für die Geschwindigkeiten aus der Lageregelung werden mit den interpolierten Geschwindigkeiten addiert, sodass bei Ermittlung einer Regelabweichung diese im besten Fall direkt behoben werden kann. Für die translatorische Geschwindigkeit ist eine Stellgrößenbegrenzung auf 255mm/s notwendig (s. Programmcode 6-17).

```

270     v_x_korr=vx_soll[k]+d_v_x
271     v_y_korr=vy_soll[k]+d_v_y
272     omega_korr=omega_soll[k]+d_omega
273
274     v_ges_korr=math.sqrt(v_x_korr**2+v_y_korr**2)
275
276     if (v_ges_korr>255):
277         v_ges_korr=255
278     else:
279         v_ges_korr

```

Programmcode 6-17: Bahnplanung "Korrektur der Geschwindigkeitswerte"

Im letzten Schritt der *if*-Abfrage wird aus den korrigierten Geschwindigkeiten die Fahrtrichtung zum Interpolationspunkt bezogen auf das Fahrzeukoordinatensystem berechnet (s. Programmcode 6-18). Es liegen daher alle Informationen vor, um eine Bewegung bis zum Interpolationspunkt durchzuführen. Zudem wird die Variable *k* um den Wert eins erhöht, damit im nächsten Durchlauf der nächste Interpolationspunkt angefahren werden. Im Anschluss an die *if*-Abfrage werden die Daten über die serielle Schnittstelle an das *Arduino-Board* des Fahrzeuges übergeben. Darauf folgt eine *time.sleep(delta\_t)*-Funktion, wodurch das System um die Intervallzeit „*delta\_t*“ pausiert, bevor der nächste Durchlauf gestartet wird. Dadurch hat das Fahrzeug die entsprechende Zeit, die Fahrbewegung durchzuführen.

```

282     rad = math.atan2(v_y_korr,v_x_korr)
283     k=k+1

```

Programmcode 6-18: Bahnplanung "Berechnung der Fahrtrichtung"

Nachdem alle interpolierten Punkte der Listen durchlaufen sind, sollte sich das Fahrzeug an der Zielposition mit der Zielorientierung befinden. Damit das Fahrzeug weiterhin seine Position hält, wurde zu Beginn eine `while`-Schleife eingeführt. Diese besteht aus der bereits betrachteten `if`-Schleife und zusätzlich aus einem „`else-statement`“. Da die `if`-Abfrage vollständig durchlaufen ist, wird nun bei jedem weiteren Durchlauf das „`else-statement`“ betrachtet. In dieses ist nochmals die Lageregelung integriert (s. Programmcode 6-19), sodass das Fahrzeug seine Position halten kann. Erst bei Abbruch des Vorgangs wird die Lageregelung ausgeschaltet.<sup>7</sup>

```
287     else:  
288         d_v=Lageregelung(x_ziel,y_ziel,phi_ziel)  
289         print("Werte aus Lageregelung: " +str(d_v))  
290         d_v_x=d_v[0]  
291         d_v_y=d_v[1]  
292         omega_korr=d_v[2]  
293         rad = math.atan2(d_v_y,d_v_x)  
294         v_ges_korr=math.sqrt(d_v_x**2+d_v_y**2)
```

Programmcode 6-19: Bahnplanung "Halten der Zielposition und Zielorientierung"

---

<sup>7</sup> Der vollständige Programmcode für die Bahnplanung bzw. Trajektorienplanung ist in Anhang F zu finden.

## 7 Autonome Pfadplanung

Bei der Pfadplanung bzw. Bahnplanung wird ein Pfad von einem Startpunkt bis zu einem Zielpunkt ermittelt. Zuvor wurde der Pfad als direkte Verbindung zwischen Start und Ziel ausgeführt, nun soll der Pfad über sogenannte Via-Punkte durchlaufen. Diese geben den Pfad vor und müssen durchfahren werden. Der Begriff Autonomie ist in der Literatur nicht klar definiert. Für diese Arbeit wird er definiert als Selbstständigkeit. Das bedeutet im Zusammenhang mit der Pfadplanung, dass ein Pfad zwischen Start und Zielpunkt nicht einfach durch einen Anwender vorgegeben wird. Mit Hilfe eines Algorithmus wird stattdessen ein geeigneter Pfad ermittelt, sodass das Fahrzeug sich „selbstständig“ seinen Weg zum Ziel sucht. Dieses Kapitel beschäftigt sich mit der autonomen Pfadplanung. Hierfür werden zunächst in einem Sprint Backlog die Anforderungen an den Sprint gelistet und beschrieben. Anschließend werden verschiedene Pfadplanungsalgorithmen betrachtet und ein geeigneter Algorithmus ausgewählt. Dieser wird letztlich implementiert. Zudem wird das Skript der Bahnplanung bzw. Trajektorienplanung aus Kapitel 6 erweitert, sodass eine Fahrbewegung für eine Mehrpunkt-Bewegung ausgeführt werden kann. Zum Schluss werden die gesamten Teilprogramme dieser Arbeit zusammengefügt, sodass eine autonome Bewegungssteuerung erzielt wird.

### 7.1 Sprint Backlog #3

Die Tabelle 7-1 enthält die Anforderungen für den dritten *Sprint*. Die ersten beiden Anforderungen behandeln das Fahren entlang eines vorgegebenen Pfades sowie das autonome Suchen eines Pfades zu einem vorgegebenen Punkt. Daher ist ein Pfadplanungsalgorithmus zu implementieren, der selbstständig einen Pfad in einer Umgebung bis hin zu einem definierten Ziel sucht. Anschließend muss das Fahrzeug diesen Weg abfahren können. Weil in dem *Sprint* zuvor ein stetiger Geschwindigkeitsverlauf gefordert wurde, ist dieser auch hier für die Fahrbewegung einzuführen. Eine weitere Anforderung ist das Fahren in einer statischen Umgebung. Es werden bei Hindernissen drei Arten unterschieden. Zum einen gibt es die statischen Hindernisse, zum anderen die semistatischen Hindernisse und zuletzt die dynamischen Hindernisse. Statische Hindernisse sind ortsfest und unbeweglich, semi-statische Hindernisse verweilen für eine gewisse Zeitspanne am gleichen Ort, werden dann allerdings an einen anderen Ort verschoben und dynamische Hindernisse bewegen sich frei in der Umgebung (vgl. Künemund, 2017: S.24). In dieser Arbeit sollen ausschließlich ortfeste also statische Hindernisse betrachtet werden.

Tabelle 7-1: Sprint Backlog #3

Sprint Backlog #3	
Nr.	Anforderung
1	Fahren entlang eines vorgegebenen Pfades über mehrere Punkte
2	Autonomes Suchen eines Pfades zu einem vorgegebenen Punkt
3	Fahren in einer statischen Umgebung
4	Kollisionsfreies Fahren im freien Konfigurationsraum
5	Definierbarer Arbeits- und Konfigurationsraum
6	Kompensation von Bewegungsungenauigkeiten
7	Erstellen von virtuellen Hindernissen

Die nächsten beiden Anforderungen thematisieren den Arbeits- und Konfigurationsraum. Es soll eine kollisionsfreies Fahren in einem freien Konfigurationsraum möglich sein und der Arbeits- und Konfigurationsraum sollen klar und einfach definiert werden können. Daher stellt sich zunächst die Frage, was genau ein Arbeits- und Konfigurationsraum sind. Der Arbeitsraum beschreibt physikalisch betrachtet, jenen Raum in der sich das FTF bewegt (vgl. Künemund, 2017: S.25). In dieser Arbeit ist das der gesamte Raum, welcher vom Trackingsystem erfassbar ist. Der Konfigurationsraum wird in der Robotik als die Menge aller einnehmbaren Konfigurationen eines Roboters definiert. Da sich das Fahrzeug ohne Einschränkungen in der Ebene bewegen kann, entspricht in diesem Fall der Konfigurationsraum dem Arbeitsraum. Allerdings wird beim Konfigurationsraum des Weiteren zwischen freiem und belegtem Konfigurationsraum unterteilt. Der freie Konfigurationsraum stellt die Menge aller Konfigurationen dar, welche das Fahrzeug einnehmen darf. Der belegte Konfigurationsraum ist die Menge alle Konfigurationen, welche durch ein Hindernis belegt sind (vgl. Hinzmann, 2011: S.39f). Die Anforderung eines kollisionsfreien Fahrens erfordert das Fahren im freien Konfigurationsraum. Zudem ist eine Abfrage beim Suchen eines Pfades erforderlich, um sicher zu gehen dass der geplante Pfad nicht durch den belegten Konfigurationsraum läuft. Dafür ist eine klare Definition des freien Konfigurationsraums notwendig. In dieser Arbeit sollen dabei allerdings lediglich virtuelle Hindernisse betrachtet werden. Das bedeutet diese müssen durch den Anwender selbst erzeugt werden.

Die letzte Anforderung beschäftigt sich mit den Bewegungsungenauigkeiten. Es müssen die Bewegungsungenauigkeiten kompensiert werden. Bei der Ausführung von geplanten Bewegungen kommt es oftmals zu Abweichungen von der geplanten Trajektorie z.B. durch Schlupf der Räder. Diese müssen zum Erreichen des Ziels korrigiert werden (vgl. Künemund, 2017: S.24). In dem zweiten *Sprint* wurde dazu bereits die Lageregelung eingeführt. Diese soll auch in die Pfadplanung integriert werden.

## 7.2 Pfadplanungsalgorithmen

Für die Pfadplanung existieren diverse Algorithmen. Sie unterscheiden sich in der Art, wie sie nach einem Pfad suchen und einen geeigneten Pfad auswählen. Die Vielzahl an verschiedenen Pfadplanungsalgorithmen kann in diverse Klassen aufgeteilt werden. Es sollen daher drei dieser Klassen betrachtet werden. Die erste Klasse sind die zellbasierten Verfahren, die zweite Klasse die Potentialfeldverfahren und die dritte Klasse die Sampling-basierten Verfahren. Im Folgenden soll kurz auf die einzelnen Verfahren eingegangen und Beispiele erläutert. Anschließend wird ein geeignetes Verfahren ausgewählt.

### Zellbasierte Verfahren

Bei den zellbasierten Verfahren wird der Arbeitsraum bzw. der freie Konfigurationsraum in Zellen aufgeteilt. Die Größe und Geometrie der Zellen variieren abhängig vom jeweiligen Algorithmus. Benachbarte Zellen werden in einen Graphen eingetragen, wobei die Knoten der Graphen den Zellen entsprechen. Innerhalb des Graphen werden einzelne Knoten verbunden. Dabei können lediglich benachbarte Knoten mit einer Kante verbunden werden. Nachdem der Graph aufgebaut ist, wird festgelegt, in welcher Zelle Anfangs- und Zielpunkt liegen. Anschließend wird ein Graph zwischen diesen gesucht. ( Hinzmann, 2011: S.42)

Eines der bekanntesten und gängigsten zellbasierten Verfahren ist der A\*-Algorithmus (s. Abbildung 7-1). Hier wird der Arbeitsraum wie beschrieben in Zellen eingeteilt. Die Genaugigkeit des Algorithmus ist abhängig von der Zellgröße, wobei auch unterschiedliche Zellgrößen definiert werden können. Zunächst wird der Start- und der Zielknoten festgelegt und die Hindernisse definiert. Einer der wichtigsten Aspekte ist dann die Kostenfunktion. Das bedeutet, für jede Zelle werden die Kosten ermittelt. Die Kosten bestehen dabei aus dem zurückgelegten Weg und den voraussichtlichen Kosten für den Weg bis zum Ziel. Wird z.B. wie in Abbildung 7-1 der vierte Knoten bzw. die vierte Zelle betrachtet, so hat diese eine Distanz von 4 Zellen von Startpunkt zurückgelegt. Die bisherigen Kosten belaufen sich daher auf 4. Die künftigen Kosten zum Ziel werden anschließend geschätzt. Von Zelle 4 bis zum Ziel 19 sind es in der Diagonalen über das Hindernis ca. 7. Zusammen belaufen sich die Kosten auf 11. So werden die Kosten von benachbarten Zellen immer gegeneinander abgewogen und der Weg mit den kürzesten Kosten ausgewählt. (vgl. Swift, 2017)

Vorteil dieses Algorithmus ist vor allem, dass dieser bei gleicher Zellteilung einfach zu implementieren ist. Zudem wird immer der optimale bzw. kürzeste Weg zum Ziel gefunden. Nachteilig ist jedoch das gerade bei größeren Arbeitsräumen und möglichst exakter Abbildung des Konfigurationsraum der Rechenaufwand steigt und das Verfahren ungeeignet wird für Systeme, welche eine schnelle Pfadplanung benötigen.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18		20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Abbildung 7-1: Zellaufteilung bei einem A-Stern Algorithmus  
( Swift, 2017)

### Potentialfeldverfahren

Die Ideen von Potentialfeldern ist einer der ersten Ansätze zur Pfadplanung. Die Idee des Potentialverfahrens basiert darauf, dass sich Teilchen unter Einfluss eines Potentialfeldes im Raum bewegen. Daher wird bei diesem Verfahren jedes Objekt als Hindernis mit einem künstlichen Kraftfeld betrachtet. Das Fahrzeug ist dabei eine Punktmasse, welches sich unter dem Einfluss der Potentialfelder im Raum bewegt. Der Start stellt dabei ein abstoßendes Potential dar, wohingegen das Ziel in einer Potentialsenke liegt und daher anziehend wirkt. Hindernisse sind ein Potentialgebirge und erzeugen damit eine abstoßende Kraft. In jeder Konfiguration wirkt eine Gesamtkraft auf das Fahrzeug, daraus wird zu jedem Zeitpunkt die Drehmomente der Motoren bestimmt. Die Abbildung 7-2 verdeutlicht das Verfahren nochmal. Der blaue Bereich stellt hier eine Potentialsenke dar. In diesem Bereich befindet sich das Ziel. Der Start und die Hindernisse sind Potentialgebirge. Durch Überlagerung der einzelnen Potentialfelder ergibt sich das Gesamtpotential des Raumes. (vgl. Koren & Borenstein, 1991: S.1f)

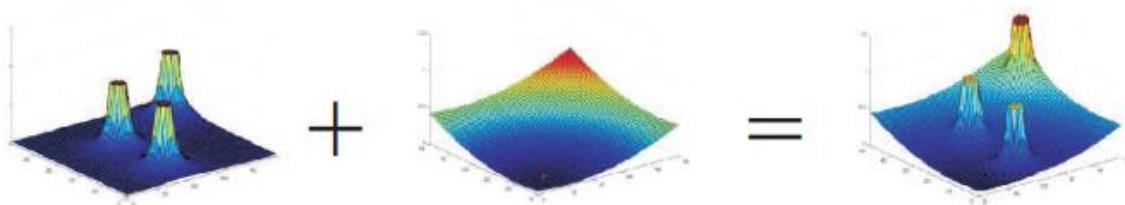


Abbildung 7-2: Gesamtpotential  
( Kavraki & LaValle, 2016: S.145)

Bei Umgebungen mit wenigen bzw. keinen Hindernissen können so schnelle Lösungen gefunden werden. Allerdings besteht die Gefahr, dass lokale Minima nicht überwunden werden können. Zudem ist die Abbildung der Potentiale in einem kartesischen Raum sehr kompliziert und mit zunehmender Anzahl an Hindernissen und somit engen Passagen kann es zum Oszillieren des Fahrzeuges kommen. Um dies zu vermeiden, muss ein großer Aufwand bei Erstellen der Potentialfelder betrieben werden. (vgl. Koren & Borenstein, 1991: S.7f)

## Sampling-basierte Verfahren

Die sampling-basierten Verfahren sind eine weitere Unterklasse der Pfadplanungsalgorithmen. Hier werden aus dem Konfigurationsraum zufällig Konfigurationen gesucht. Die schnelle Ausbreitung und Erkundung des kompletten Konfigurationsraum ist dabei im Vordergrund. Dafür gibt es verschiedene Samplingstrategien. Diese sind darauf ausgelegt den Konfigurationsraum nach bestimmten Kriterien abzudecken. Das Optimumkriterium ist hierbei möglichst schnell einen Pfad zu finden. Durch bestimmte Heuristiken kann diese Pfadsuche beschleunigt werden. Die reine Planung für ein einfaches Problem kann dadurch innerhalb von Sekunden erfolgen. Allerdings benötigt eine solcher Pfad oftmals Optimierungsschritte zum Verschleifen oder um Umwege zu entfernen. (vgl. Leidner, 2011: S.8)

Im Folgenden sollen die beiden Verfahren RRT und PRM näher dargestellt werden, um die sampling-basierten Verfahren besser nachvollziehen zu können.

## Probabilistic Roadmaps PRM

Die *Probabilistic Roadmaps* sind eine Variante der Sampling-basierten Verfahren. Beim PRM wird der Pfad im Konfigurationsraum in zwei Schritten bestimmt. Im ersten Schritt wird eine sogenannte *Roadmap* erstellt. Diese stellt eine Art Straßenkarte dar. Dieser Schritt wird auch als Konstruktionsphase bezeichnet. Es werden hierbei zufällig kollisionsfreie Konfigurationen erstellt und diese anschließend durch einen Planer miteinander verknüpft. Dabei werden jeweils die nächsten Nachbarn untereinander verknüpft, wodurch ein dichtes Netz entsteht. Dieses Netz bildet eine Karte ab, in der der Konfigurationsraum auf verschiedenen Wegen kollisionsfrei durchlaufen werden kann (s. Abbildung 7-3). Im zweiten Schritt wird versucht eine Startkonfiguration mit einer Zielkonfiguration mittels Graphensuche miteinander zu verbinden. (vgl. Choset, et al., 2005: S. 203-208)

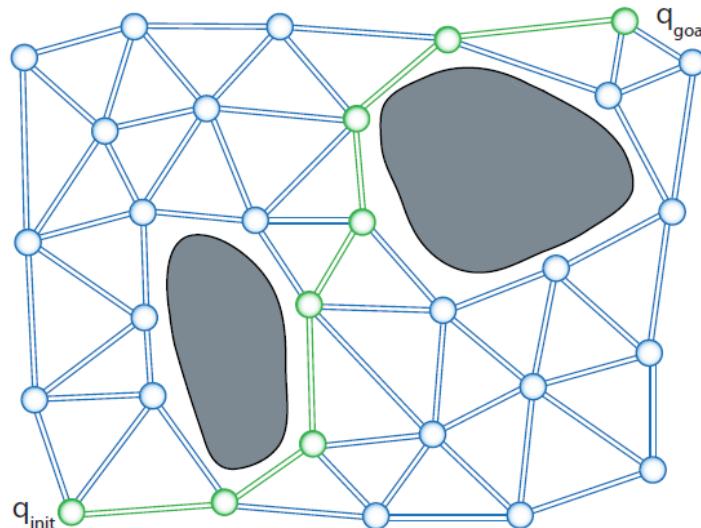


Abbildung 7-3: Erstellen einer *Probabilistic Roadmap*  
(Leidner, 2011)

Solange sich die Szene nicht ändert, können einmal erstellte *Roadmap* immer wieder zur Pfadplanung verwendet werden. Sollte jedoch in der Erkundungsphase mehrmals kein Ergebnis zwischen Start und Ziel gefunden werden, ist es möglich, die Roadmaps zu erweitern, um so z.B. durch weitere Erkundung des Konfigurationsraum einen geeigneten Pfad zu finden. Verändert sich die Umgebung, so kann der Algorithmus zudem um eine erneute Kollisionsabfrage zur Laufzeit erweitert werden. Die *Probabilistic Roadmaps* eignen sich vor allem für Szenarien, bei denen die gleiche Aufgabe immer wieder durchgeführt wird. Der detaillierte Aufbau der Karte nimmt zwar viel Zeit in Anspruch, dafür können aber im Nachhinein umso schneller realisierbare Bahnen gefunden werden. Für sich ständig verändernden Umgebungen ist das Verfahren hingegen ungeeignet, da die gesamte Karte ständig überprüft werden müsste. (vgl. Bohlin & Kavraki, 2000: S.527f)

### Rapid Exploring Random Tree

Die Klasse der *Rapid-Exploring Random Trees (RRT)* ist eine weitere Algorithmusfamilie zur zufallsbasierten Erkundung des Konfigurationsraums. Anders als bei PRMs wird bei RRTs eine direkte Verbindung zwischen einzelnen Knoten hergestellt, wodurch jeder Knoten einen Vorgänger hat, ausgenommen des Startknotens (s. Abbildung 7-4). Die Idee bei RRTs ist es, den Konfigurationsraum schnell mit einem Suchbaum abzudecken. Dabei werden solange Äste am Suchbaum erzeugt, bis eine Verbindung zwischen Start- und Zielkonfiguration vorliegt. Bekannt sind hierbei zunächst der Start- und Zielpunkt sowie die Hindernisse auf der Karte. In dem Arbeitsraum werden zufällig Knoten ausgewählt und überprüft, ob diese im freien Konfigurationsraum liegen. Befindet sich der Knoten im freien Raum, so wird er mit dem dichtesten Knoten oder Kante des Baumes verbunden. Wird die Kante wiederrum akzeptiert, da sie mit keinem Hindernis kollidiert, so entsteht ein neuer Ast am Baum. Der Algorithmus wird wiederholt durchgeführt, bis der Baum Start- und Zielknoten beinhaltet. Im Anschluss wird geschaut, über welche der Baumverzweigungen ein geeigneter Pfad zwischen Start und Ziel verläuft. (vgl. Kuffner James & LaValle, 2000: S.2f)

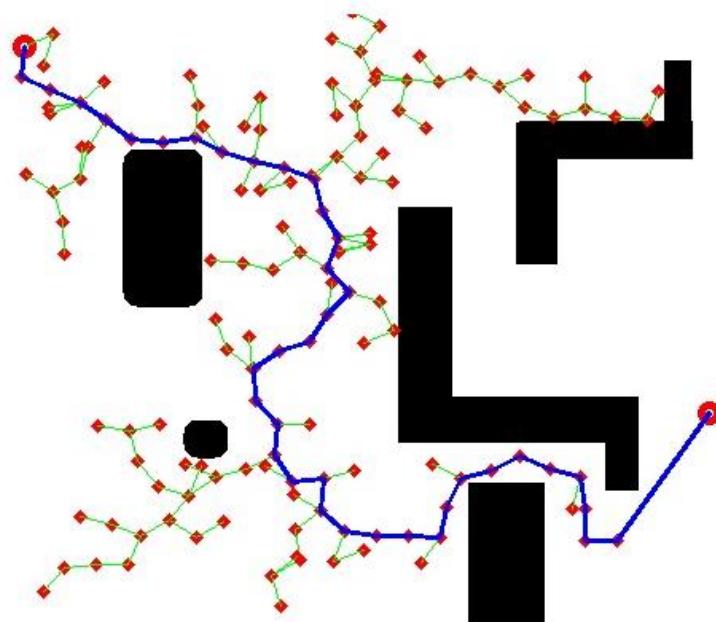


Abbildung 7-4: Rapid Exploring Random Trees  
( Yadav, 22)

Die RRTs sind besonders gut geeignet für Anfragen, bei denen sich die Umgebung anschließend ändert. Ein unerkundeter Raum wird schnell exploriert und direkt zielführend ein realisierbarer Pfad gefunden. Im Gegensatz zu PRMs liegt nach einer Änderung des Konfigurationsraumes kein ungültiges Netz vor, da dieses immer wieder neu erzeugt wird. Dadurch ist das Verfahren jedoch auch ein wenig langsamer als die PRMs. (vgl. Leidner, 2011: S.11)

## Auswahl eines geeigneten Verfahrens

In den vorherigen Abschnitten wurden verschiedene Algorithmen für die Pfadplanung betrachtet. Nun soll ein geeigneter Algorithmus für diese Arbeit ausgewählt werden. Auf einige Vorteile und Nachteile der einzelnen Algorithmen wurden bereits kurz eingegangen. Die Tabelle 7-2 stellt diese nochmals gegenüber und fügt weitere Vor- und Nachteile hinzu.

Tabelle 7-2: Vergleich der Pfadplanungsverfahren

Vergleich der Verfahren	Vorteile	Nachteile
A-Stern	Einfach zu implementieren	hoher Rechenaufwand
	Reduktion auf einfache Graphensuche	hohe Kartenauflösung führt zu hohem Speicherbedarf
	Kürzester Weg zwischen Start und Ziel	Fahrzeug muss kleiner als Zellgröße sein
Potentialfeld	Anschauliche Idee	hoher Implementierungsaufwand
	Geeignet für keine bis wenig Hindernisse	schwierig geeignete Potentialfunktion zu bilden
	für statische und dynamische Umgebungen	ggfs. Ziel unerreichbar
	glatte Pfade werden erzeugt	Oszillationsgefahr
PRM	sehr effizient und schnell	Optimierungsschritte notwendig
	für komplexer Probleme geeignet	Abstand zu Hindernissen nicht maximal
	schnelle Suche in Echtzeit	ungeeignet für veränderliche Umgebung
	relativ leicht zu implementieren	-
RRT	sehr effizient und schnell	Optimierungsschritte notwendig
	auf dynamische Umgebungen erweiterbar	Abstand zu Hindernissen nicht maximal
	für komplexe Probleme geeignet	-
	relativ leicht zu implementieren	-

Bei dem zellbasierten Verfahren A-Stern reduziert sich das Problem der Pfadplanung auf eine einfache Graphensuche. Diese ist einfach zu implementieren. Der ermittelte Pfad ist stets der kürzeste Weg zwischen Start- und Zielkonfiguration. Allerdings muss das Kartenmaterial zunächst diskretisiert werden. Dafür muss die Fahrzeuggöße kleiner als die Zellgröße sein, da ansonsten die Position nicht genau definiert ist. Je besser die Kartenauflösung ist desto höher ist der Speicherbedarf. Zudem nimmt der Rechenaufwand mit höherer Kartenauflösung und höherer Anzahl an Hindernissen zu und ist damit nicht mehr so effizient.

Das Potentialfeldverfahren besticht durch seine anschauliche Idee und ist zudem für Umgebungen mit keinen bzw. wenig Hindernissen geeignet. Außerdem werden direkt glatte Pfade erzeugt, sodass im Anschluss keine weiteren Optimierungsschritte notwendig sind. Im Gegensatz dazu ist der Implementierungsaufwand hoch, vor allem da es schwierig ist, eine geeignete Potentialfunktion zu bilden. Des Weiteren kann es passieren, dass lokale Minima auftreten und das Ziel somit ggfs. nicht erreichbar ist. Bei vielen Hindernissen besteht darüber hinaus eine Oszillationsgefahr des Fahrzeugs.

Das PRM ist besonders für komplexe Probleme geeignet. Nachdem in der Konstruktionsphase eine Karte erstellt ist, arbeitet das Verfahren sehr schnell und effizient, sodass sogar eine Graphensuche in Echtzeit erfolgen kann. Außerdem ist das Verfahren relativ einfach zu implementieren, wobei sich das Wort relativ auf den Vergleich zum A-Stern-Algorithmus bezieht, der noch einfacher zu implementieren ist. Nachteilig bei diesem Verfahren ist hier, dass weitere Optimierungsschritte zum Glätten bzw. Verschleifen des Pfades erforderlich sind. Zudem ist er nicht in veränderlichen und dynamischen Umgebungen geeignet. Gleichzeitig wird nicht der Pfad mit einem maximalen Abstand zu den Hindernissen gesucht.

Die Vorteile und Nachteile des RRT ähneln denen des PRMs, da es sich bei beiden Verfahren um sampling-basierte Ansätze handelt. Das RRT arbeitet ebenfalls sehr effizient und schnell. Zudem ist es für komplexe Probleme geeignet und relativ leicht zu implementieren. Im Vergleich zum PRM läuft die Graphensuche jedoch nicht in Echtzeit. Es sind ebenfalls weitere Optimierungsschritte nach der Pfadfindung notwendig. Die Abstände zu den Hindernissen sind nicht maximal. Die Dauer der Pfadsuche ist abhängig von der Komplexität der Umgebung und kann daher variieren. Vorteilhaft ist, dass das Verfahren auf dynamische Umgebungen erweiterbar ist.

Aus dem Vorteil-Nachteil-Vergleich zeigt sich, dass die Eignung eines Verfahrens abhängig von der Aufgabenstellung und der Umgebung ist. Für diese Arbeit eignen sich im Besonderen die sampling-basierten Ansätze. Für den hier betrachteten Arbeitsraum sind PRM und RRT gleichermaßen anwendbar. Allerdings ist der spätere Einsatzzweck des FTF nicht bekannt. Es kann sich somit um einen Einsatz in einer dynamischen oder statischen Umgebung handeln, wobei das System voraussichtlich für dynamischen Umgebung erweitert werden soll. Daher ist in diesem Fall das RRT-Verfahren dem PRM-Verfahren vorzuziehen und soll dementsprechend umgesetzt werden.

### 7.3 Implementieren des Pfadplanungsalgorithmus

Nachdem im Kapitel 7.2 zur Pfadplanung ein RRT-Algorithmus ausgewählt wurde, soll dieser implementiert werden. Jedoch existieren selbst bei den RRT-Algorithmen wieder verschiedene Arten der Pfadsuche abhängig von den gewählten Heuristiken. Für den RRT-Algorithmus in dieser Arbeit werden zwei Heuristiken definiert. Zum einen erfolgt keine gerichtete Pfadsuche in Richtung des Ziels, d.h. dass die Knoten im Raum zufällig ausgewählt werden. Es wird keine Ausbreitungsrichtung des Baumes vorgegeben. Dadurch wird die Komplexität des Algorithmus minimiert. Um dennoch eine möglichst effiziente Pfadsuche zu erzielen, soll zum anderen eine Pfadkontrolle Richtung Ziel durchgeführt werden. Nachdem ein neuer Knoten dem Baum hinzugefügt wurde, soll überprüft werden, ob ein kollisionsfreier Weg vom letzten Knoten, also zum Ziel, möglich ist. Ist dies der Fall ist die Pfadsuche bereits abgeschlossen, ansonsten wird sie fortgesetzt. Die genaue Funktionsweise und Vorgehensweise für den RRT-Algorithmus wird anhand des Programmcodes erläutert. Dabei werden allerdings nur Ausschnitte des Codes begutachtet.

Zunächst werden zwei neue Funktionen definiert. Diese beiden Funktionen haben die Aufgabe zu kontrollieren, ob sich ein neuer Knoten bzw. auch der Zielknoten im freien Konfigurationsraum befindet (s. Programmcode 7-1). Die erste Funktion „inHindernis“ bekommt dementsprechend einen Punkt übergeben und berechnet für den Punkt den Abstand zu jedem Hindernis, welcher sich im Raum befindet. Dazu wird eine neue Liste geschaffen. Diese speichert, ob sich der Punkt in einen Hindernis befindet mit „True“ oder ob er im freien Raum liegt. Ist dies der Fall, wird in der Liste der Boolesche Operator „False“ eingetragen. Als Rückgabewert wird die Liste mit den Boolschen Operatoren „False“ und „True“ zurückgegeben. Die Funktion „inArbeitsraum“ arbeitet hingegen mit einem Status „AR\_OK“. Ist dieser Status auf „True“ gesetzt, so befindet sich der Punkt im Arbeitsraum. Zunächst wird der Status auf „True“ gesetzt. Anschließend wird durch *if*-Abfragen überprüft, ob er tatsächlich zwischen den angegebenen Grenzen des Arbeitsraums liegt. Sobald eine x-Koordinate oder y-Koordinate außerhalb der Grenzen ist, wird der Status auf „False“ gesetzt. Zurückgegeben wird abschließend der Status.

```

592     def inHindernis(p_check):
593         in_Hindernis=[]
594         for a in range (H):
595             distanz=berechne_distanz(Hindernisse[a][0],p_check)
596
597             if distanz <= Hindernisse[a][1]:
598                 in_Hindernis.append(True)
599
600             else:
601                 in_Hindernis.append(False)
602
603     return in_Hindernis
604
605 def inArbeitsraum(p1):
606     print ("Funktion AR")
607     AR_OK=True
608     print("p_x: "+ str(p1[0]))
609     print("p_y: "+ str(p1[1]))
610     if AR_x[1] <p1[0] or p1[0] < AR_x[0]:
611         AR_OK=False
612     if AR_y[1] <p1[1] or p1[1] < AR_y[0]:
613         AR_OK=False
614
615     return AR_OK

```

Programmcode 7-1: Pfadplanung "Funktionen für AR-Kontrolle und Hinderniskontrolle"

Als nächstes sollen zwei Funktionen eingeführt werden, welche die Lage eines Punktes bezogen auf zwei weitere Punkte bzw. auf den Vektor zwischen ihnen beschreibt. Dies ist erforderlich, um zu überprüfen, ob eine neue Kante zwischen zwei Knoten durch ein Hindernis läuft. Dazu ist jedoch zunächst Hintergrundwissen erforderlich.

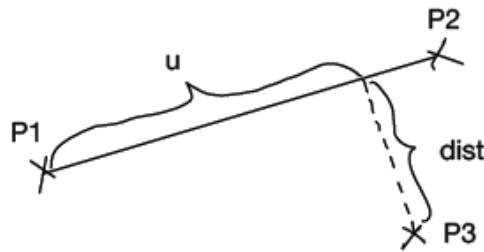


Abbildung 7-5: Lage dreier Punkte im Raum nach Borke, 1988

Die Abbildung 7-5 stellt die Aufgabenstellung dar. Gegeben sind drei Punkte P1, P2 und P3. Punkte P1 und P2 sollen hierbei die Knoten im RRT-Baum abbilden, während Punkt P3 als Hindernis fungiert. Zwischen den beiden Punkten P1 und P2 wird ein Linie gezogen. Dieser berechnet sich über die Gleichung 7-1. (vgl. Borke, 1988)

$$P = P1 + u(P2 - P1) \quad (7-1)$$

In dieser Gleichung ist  $u$  die unbekannte Größe. Diese gibt an, an welcher Position auf der Linie sich ein bestimmter Punkt befindet. In diesem Fall ist von Interesse, wo sich der Punkt P3 bezogen auf P1 und P2 befindet. Dafür wird zunächst die Formel zur Berechnung der Größe  $u$  herangezogen und anschließend können die x-Koordinate und y-Koordinate des Schnittpunktes berechnet werden. Die Größe  $u$  nimmt hier einen Wert zwisch 0 und 1 an.

$$u = \frac{(x_3 - x_1)(x_2 - x_1) + (y_3 - y_1)(y_2 - y_1)}{(p_1 - p_2)^2} \quad (7-2)$$

$$x = x_1 + u * (x_2 - x_1) \quad (7-3)$$

$$y = y_1 + u * (y_2 - y_1) \quad (7-4)$$

Die Funktion, welche diese Berechnung im Code beschreibt, wird als „Punkt\_zu\_Linie“ definiert. Da die Funktionsweise bereits erläutert wurde, wird an dieser Stelle nicht näher auf die Funktion selbst eingegangen. Sie wird allerdings für eine weitere Funktion „Kollision\_Pfad\_Hindernis“ aufgegriffen (s. Programmcode 7-2). Bei dieser Funktion wird erneut die Liste aller Hindernisse durchlaufen und geschaut, ob ein Hindernis zu nah am neuen Pfad liegt. Dieser Pfad wird aus zwei Punkten des RRT-Baumes generiert. Zudem soll überprüft werden, ob sich ein Hindernis zwischen den Punkten des neuen Pfades befindet oder außerhalb. Dazu werden zwei Stati „zu\_nah“ und „dazwischen“ definiert. Diese haben zunächst den Wert „False“, was bedeutet es gibt keine Kollision zwischen Pfad und Hindernis. Zudem wird eine Liste „Linienkollision“ eingeführt, welche den Status für jedes Hindernis speichert. Anschließend wird für jedes Hindernis die Größe u, und die Distanz vom Hindernis zum Pfad mit der Funktion „Punkt\_zu\_Linie“ bestimmt. Danach wird überprüft, ob u einen Wert zwischen 0 und 1 annimmt. Ist dem so, so wird der Status „dazwischen“ auf „True“ gesetzt. Anschließend wird geprüft, ob die Distanz zwischen Pfad und Hindernis kleiner als der Radius des Hindernisses ist. Wenn die Distanz kleiner als der Radius des Hindernisses ist, so wird der Status „zu\_nah“ auf „True“ gesetzt. Nur wenn beide Stati auf „True“ sind, so befindet sich der Pfad im Hindernis. Somit wird der Liste „Linienkollision“ der Status „True“ hinzugefügt, ansonsten der Status „False“. Sobald der Pfad nur ein Hindernis kreuzt, wird der neue Pfad verworfen. Dies wird aber zu einem späteren Zeitpunkt erläutert. Zurückgegeben wird von der Funktion die Liste mit den Stati.

```

633     def Kollision_Pfad_Hindernis(p1,p2):
634         LinienKollision=[]
635         for s in range(H):
636             zu_nah=False
637             dazwischen=False
638
639             u, dist =Punkt_zu_Linie(p1,p2,Hindernisse[s][0])
640
641             if 0<=u <=1:
642                 dazwischen=True
643
644             if dist <= Hindernisse[s][1]:
645                 zu_nah=True
646
647             if zu_nah and dazwischen:
648                 LinienKollision.append(True)
649             else:
650                 LinienKollision.append(False)
651
652     return LinienKollision

```

Programmcode 7-2: Pfadplanung "Funktion für Kollisionskontrolle"

Im nächsten Schritt wird der Arbeitsraum definiert und Hindernisse erzeugt. Der Arbeitsraum ist in dieser Arbeit beschränkt auf die Größe der Kalibriervorlage, da die Testumgebung recht klein ist. In den Sprint-Anforderungen wurde definiert, dass virtuelle Hindernisse erzeugt werden. Es werden in dieser Arbeit also zunächst keine Hindernisse von der Trackingkamera erfasst. Stattdessen gibt der Anwender die Position und Größe des Hindernisses ein (s. Programmcode 7-3). Dafür wird zunächst gefragt, wie viele Hindernisse der Anwender in seiner Umgebung haben möchte. Die Anzahl an Hindernissen wird der Variablen H zugeordnet. Im Anschluss wird eine leere Liste für die Hindernisse erzeugt. Im Folgenden wird diese gefüllt. Hierfür gibt der Anwender für jedes Hindernis die jeweilige Position und den Radius des Hindernis an. Diese Parameter werden für jedes Hindernis bestimmt und dann der Liste „Hindernisse“ hinzugefügt.

```

657 AR_X=(-525,525)
658 AR_y=(-320,320)
659
660 #Hindernisse erzeugen
661 H=int(input('Bitte gib Anzahl an Hindernissen ein: '))
662 Hindernisse=[]
663 R_FTS=550/2
664 for i in range (H):
665     x_Hindernis=float(input("x_Koordinate des Hindernis "+str(i+1)+":"))
666     y_Hindernis=float(input("y_Koordinate des Hindernis "+str(i+1)+": "))
667     MP_Hindernis=(x_Hindernis,y_Hindernis)
668     R_Hindernis=float(input("Radius des Hindernis "+str(i+1)+": "))+R_FTS
669     HindernisPara=[MP_Hindernis,R_Hindernis]
670     Hindernisse.append(HindernisPara)
671     print("Hinderniss: "+str(Hindernisse))

```

Programmcode 7-3: Pfadplanung "Definition des Konfigurationraums"

Bevor der Radius der Parameterliste der Hindernisse hinzugefügt werden kann, wird dieser mit dem Radius des Fahrzeuges addiert. Die Begründung hierzu wird in Abbildung 7-6 verdeutlicht. Bei der Überprüfung, ob sich ein Punkt im Hindernis befindet, muss berücksichtigt werden, dass dieser Punkt später von Fahrzeug abgefahren werden könnte. Von dem Fahrzeug wird allerdings nur die Koordinate des Mittelpunktes betrachtet. Wenn der Mittelpunkt des Fahrzeugs außerhalb des Hindernisses liegt, bedeutet dies allerdings nicht, dass das komplette Fahrzeug außerhalb des Hindernisses liegt. Daher wird der belegte Konfigurationsraum direkt so angepasst, dass er den Radius des Fahrzeuges inkludiert. Durch diese Erweiterung ist es ausreichend nur den Fahrzeugmittelpunkt zu betrachten. Der Radius ergibt sich dabei aus der halben Diagonalen des Chassis.

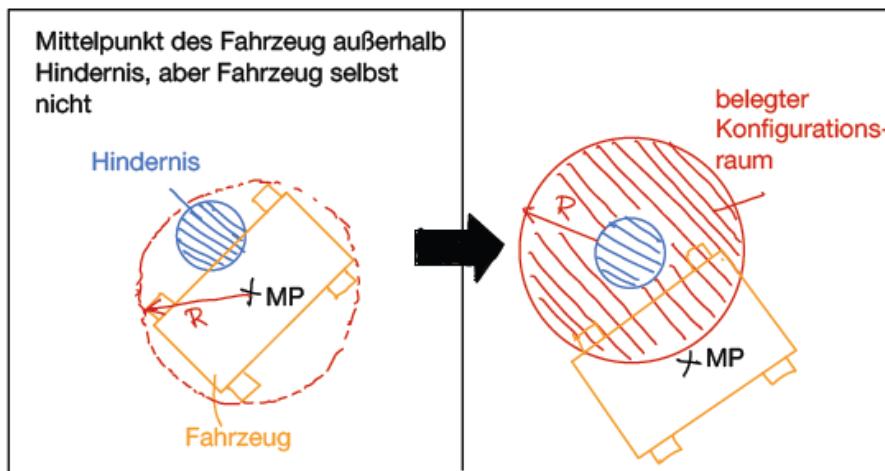


Abbildung 7-6: Anpassen des belegten Konfigurationraums

Als nächstes wird der Startpunkt ermittelt und der Endpunkt definiert (s. Programmcode 7-4). Die Koordinaten des Startpunktes wird, wie in den vorherigen Kapiteln, mittels http-Request angefragt und ausgegeben. Die Position wird nun dem Punkt „Start“ hinzugefügt. Dieser erhält den Namen „k0“. Anschließend wird eine neue Liste eingeführt, die alle neuen Punkte speichert. Für jeden Punkt werden die Informationen Name, Koordinaten und *Parent* gespeichert. Unter dem Ausdruck *Parent* verbirgt sich immer der Vorgängerpunkt also der „Eltern“-Punkt. Da der Startpunkt der erste Punkt in der Liste und im RRT-Baum ist, besitzt dieser keinen *Parent*. Der Parent wird daher mit *None* angegeben.

Nachdem der Startpunkt ermittelt und in die Liste eingetragen ist, soll nun der Zielpunkt definiert werden. Dafür soll der Anwender den Zielpunkt so lange eingeben, bis er sich im freien Konfigurationsfeld befindet. Nach jeder Eingabe wird dafür mittels den Funktionen „inHindernis“ und „inArbeitsraum“ überprüft, ob dies der Fall ist. Sobald der eingebene Punkt im freien Konfigurationsraum liegt, wird das Ziel akzeptiert und erhält den Status „True“. Im Anschluss wird eruiert, ob ein direkter Pfad zwischen Start und Ziel ohne das Kreuzen eines Hindernisses abzubilden ist. Dafür wird der Status „Weg\_zum\_Ziel“ eingeführt, welcher zunächst den Wert „False“ aufweist. Dann wird mit der Funktion „Kollision\_Pfad\_Hindernis“ geprüft, ob ein Hindernis zwischen Start und Ziel liegt. Wenn ein direkter Pfad gefunden wird, so wird der Status „Weg\_zum\_Ziel“ auf „True“ gesetzt und die Pfadsuche ist bereits abgeschlossen. Ansonsten bleibt der Status unverändert. Die Pfadsuche wird fortgesetzt.

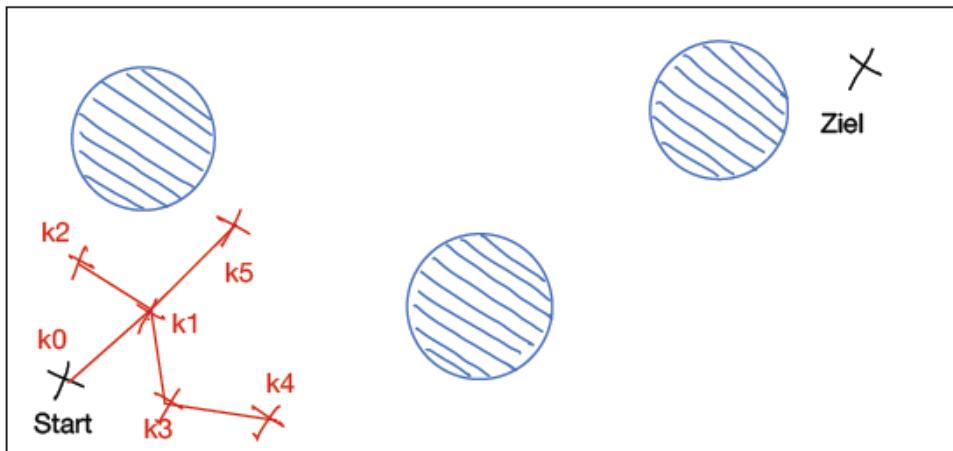
```

680 Start=(x_start,y_start)
681 print("Start: "+str(Start))
682 Start_Name='k0'
683 Punkte=[[Start_Name,Start,'None']]
684
685 #Endpunkt angeben
686 Ziel_OK=False
687 while not Ziel_OK:
688     x_Ziel=float(input("x-Koordinate des Zielpunktes: "))
689     y_Ziel=float(input("y-Koordinate des Zielpunktes: "))
690     phi_Ziel=float(input("Orientierung des Zielpunktes: "))
691     Ziel=(x_Ziel,y_Ziel)
692
693 Kollisionskontrolle=inHindernis(Ziel)
694 print("Kollision: "+str(Kollisionskontrolle))
695 AR_Kontrolle=inArbeitsraum(Ziel)
696 print("AR_Kontrolle: " +str(AR_Kontrolle))
697
698 if any (Kollisionskontrolle) or (AR_Kontrolle !=True):
699     print("ERROR: Ziel liegt nicht im Konfigurationsraum!")
700     pass
701 else:
702     Ziel_OK=True
703 print("Ziel_OK: "+str(Ziel_OK))
704
705 Weg_zum_Ziel=False
706
707 #Prüfen ob direkter Weg zu Ziel möglich
708 if any (Kollision_Pfad_Hindernis(Start,Ziel)):
709     print("Kein direkter Pfad zu Ziel möglich")
710
711 else:
712     print("Direkter Pfad zu Ziel gefunden!")
713     Weg_zum_Ziel=True
714 print("Weg zum Ziel: "+str(Weg_zum_Ziel))

```

Programmcode 7-4: Pfadplanung "Definieren des Zielpunktes"

Nun beginnt die tatsächliche Pfadsuche. Da der Programmcode dafür umfangreich ist, soll dieser nicht im Detail erläutert werden. Anstelle dessen wird die RRT-Suche mittels Flussdiagramm veranschaulicht und die einzelnen Schritte ggfs. durch eine Verbildlichung der Situation unterstützt. Der betrachtete Ausgangspunkt zur Beschreibung des Ablaufs ist in Abbildung 7-7 dargestellt. In dem RRT-Baum befinden sich bereits einige neue Punkte, jedoch führt noch keiner dieser Punkte zum Ziel. Die Punkte sind in der Liste „Punkte“ gespeichert und es soll ein neuer Punkt hinzugefügt werden.



**Punkte= [[k0,Start,None], [k1,(x,y),k0], [k2, (x,y), k1],[k3,(x,y),k1], [ k4, (x,y), k3],  
[k5,(x,y), k1]]**

Abbildung 7-7: Ausgangssituation des betrachteten RRT-Algorithmus

Das Flussdiagramm bzw. die Pfadsuche beginnt mit einer Statusabfrage (s. Abbildung 7-8). Diese überprüft, ob bereits ein Weg zum Ziel führt. Ist der Status auf „True“, so existiert bereits ein Weg, ansonsten ist der Status auf „False“ und die Pfadsuche beginnt. Dafür wird ein neuer Status für einen Punkt „Punkt\_OK“ hinzugefügt. Zu Anfang ist dieser Status immer auf „False“ gestetzt. Bei einer Kontrolle des Status ergibt sich im ersten Durchlauf ein Status „False“. Daher wird ein beliebiger Punkt im Raum erzeugt und die Distanz zwischen dem Punkt und jedem weiteren Punkt im RRT-Baum verglichen. Der Punkt im RRT-Baum, zu welchem der neue Punkt die geringste Distanz aufweist, wird als *Parent* angesehen. Im Anschluss wird zwischen dem *Parent* und *Child* der Vektor und daraus der Einheitsvektor berechnet. Da der Einheitsvektor stets die Größe eins aufweist, kann dieser mit einem beliebigen Faktor multipliziert werden. Dieser Faktor gibt an, in welchem Abstand zum Parent der neue Knoten bzw. Punkt hinzugefügt werden soll. Das bedeutet, der zuvor erzeugte beliebige Punkt dient nur zur Richtungsvorgabe für den neuen Punkt bzw. Knoten im RRT-Baum und wird anschließend nicht mehr benötigt. Dies hat den Vorteil, dass alle Punkte im RRT-Baum den selben Abstand, bis auf den letzten Punkt und dem Ziel, aufweisen.

Sobald der neue Punkt generiert wurde, erfolgt eine Kollisions- und Arbeitsraumkontrolle. Dabei werden die folgenden zwei Fragen beantwortet:

1. Liegt der neue Punkt in einem Hindernis bzw. im belegten Konfigurationsraum?
2. Befindet sich der Punkt im vorgegebenen Arbeitsraum?

Wenn die erste Frage mit „True“ beantwortet wird und bzw. oder die zweite Frage mit „False“, so wird der neue Punkt verworfen. Der Status „Punkt\_Ok“ bleibt auf „False“. Die Punktesuche beginnt von vorne. Ansonsten ist der Punkt in Ordnung und der Status „Punkt\_Ok“ wird auf „True“ gesetzt. Der neue Punkt wird der Liste für die Punkte im RRT-Baum mit den Angaben Name, Koordinaten und *Parent* angefügt. Zudem wird kontrolliert, ob vom neuen Punkt zum Ziel ein Pfad vorhanden ist, der nicht über ein Hindernis verläuft. Existiert ein solcher Pfad, wird der Status „Weg\_zum\_Ziel“ auf „True“ gesetzt, ansonsten bleibt er auf „False“. Dann beginnt die Schleife von vorne und es erfolgt wieder die

Statuskontrolle „Weg\_zum\_Ziel“. Ist diese Statuskontrolle dieses Mal positiv, dann ist ein kompletter Weg vom Start zum Ziel vorhanden und die Pfadsuche kann abgebrochen werden. Bei negativer Antwort auf die Statusabfrage beginnt die Prozedur von vorne und es wird erneut ein neuer beliebiger Punkt erzeugt.

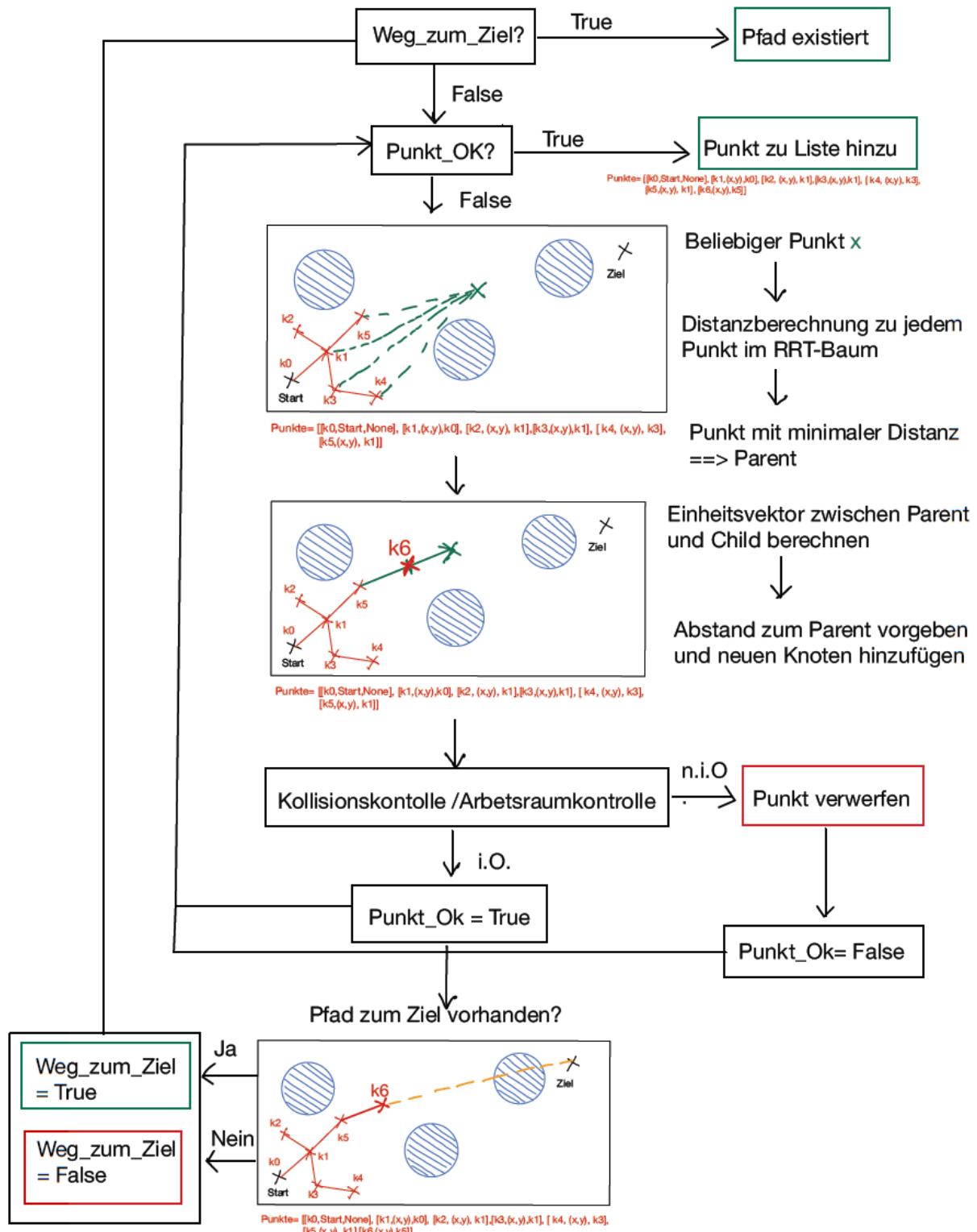


Abbildung 7-8: Flussdiagramm zur Pfadsuche

Nun liegt eine Liste mit vielen Punkten vor, jedoch ist noch unbekannt, über welchen der Punkte das Fahrzeug fahren muss, um zum Ziel zu gelangen. Dies wird durch eine Rückwärtssuche ermittelt (s. Programmcode 7-5). Dafür wird zunächst eine neue Liste für den Pfad erzeugt. Dieser Liste wird direkt der Zielpunkt zugewiesen. Zudem wird ein neuer Status „am\_Start“ eingeführt und dieser mit dem Wert „False“ belegt. Anschließend wird der Zielpunkt als aktueller Punkt vorgegeben und sein Vorgängerpunkt als *Parent*. Im Anschluss wird eine while-Schleife gestartet. Diese überprüft vor jedem Durchlauf den Status „am\_Start“ und inkludiert eine for-Schleife mit der Laufvariablen l. Die for-Schleife wird so lange durchlaufen, wie es Anzahl an Punkte in der Liste „Punkte“ gibt. Innerhalb der for-Schleife wird für jeden Punkt geprüft, ob dieser dem Parent des aktuellen Punkt entspricht. Sprich hat der Zielpunkt beispielsweise die folgenden Parameter, dann ist k8 der Name des Zielpunktes und k3 sein *Parent*.

[k8, (-300,300), k3]

Sobald die for-Schleife den Punkt mit dem Namen „k3“ erreicht hat, wird dieser an der ersten Position in die Liste des Pfades eingefügt und als aktuellen Punkt festgelegt sowie sein *Parent* als aktuellen *Parent* angegeben. Danach wird die for-Schleife weiter durchlaufen. Erst wenn die for-Schleife den Startpunkt erreicht, wird die übergeordnete while-Schleife beendet. Diese wird beendet, indem der aktuelle *Parent* mit dem Wert „None“ verglichen wird. Beim Einführen der Punktes „Start“ wurde diesem als *Parent* den Wert „None“ zugewiesen. Daher wird der Status „am\_Start“ auf „True“ gesetzt. Die Bedingung der while-Schleife ist erfüllt, sodass letztlich eine Liste „Pfad“ vorliegt. Diese Liste umfasst alle Punkt, die vom Start zum Ziel führen.

```

789 #Weg zurück vom Ziel zum Start
790 Pfad=[Punkte[-1][1]]
791 am_Start=False
792
793 aktueller_Punkt=Punkte[-1][1]
794 aktueller_Parent=Punkte[-1][2]
795
796 while not am_Start:
797     for l in range(len(Punkte)):
798         if Punkte[l][0]==aktueller_Parent:
799             Pfad.insert(0,Punkte[l][1])
800             aktueller_Punkt=Punkte[l][1]
801             aktueller_Parent=Punkte[l][2]
802         if aktueller_Parent=='None':
803             am_Start=True
804
805 Pfad.append(Ziel)

```

Programmcode 7-5: Pfadplanung "Auswahl der Punkte für Pfad"

#### 7.4 Generierung einer Fahrbewegung entlang eines Pfades

In den vorherigen Kapiteln wurde bereits ein Skript für die Pfadsuche sowie ein Skript für das Abfahren einer Punkt-zu-Punkt-Bahn erarbeitet. Durch Kombination der beiden Skripte soll nun eine Fahrbewegung entlang des gesamten Pfades über die Via-Punkte erfolgen. Da es sich hierbei nicht mehr um eine reine Punkt-zu-Punkt-Bewegung handelt, muss das Skript für die Bahnplanung bzw. Trajektorienplanung überarbeitet werden.

In der Abbildung 7-9 ist zu erkennen, dass bei einer Punkt-zu-Punkt-Bewegung die Anfangsbedingungen Geschwindigkeit am Start und am Ziel gleich Null gegeben ist. Für eine Mehrpunktbewegung ergibt sich daraus beim Durchfahren jedes Via-Punktes eine Brems- und Beschleunigungsbewegung, sodass am jeweiligen Punkt die Geschwindigkeit Null ist. Um dies zu umgehen, soll die Trajektorienplanung so überarbeitet werden, dass ausschließlich am Start beschleunigt und am Ziel abgebremst wird. Dazwischen soll das Fahrzeug mit konstanter Geschwindigkeit fahren.

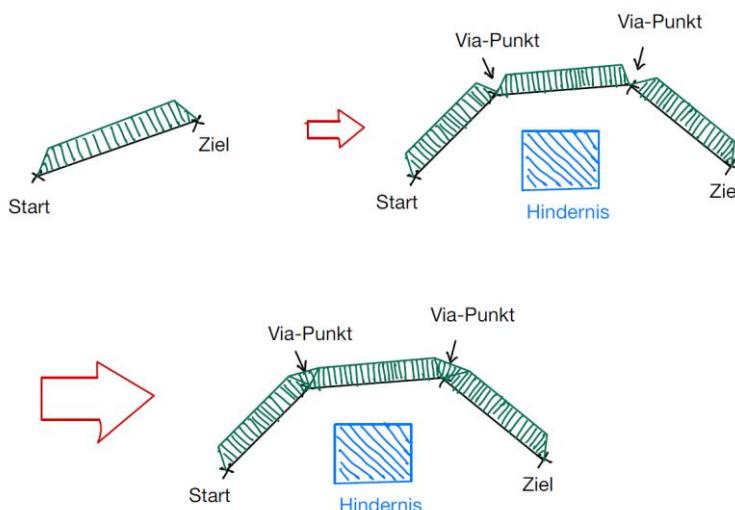


Abbildung 7-9: Geschwindigkeitsverlauf an Via-Punkten

Dazu benötigt das Skript der Bahnplanung eine Information, an welchem Punkt er sich zum aktuellen Zeitpunkt befindet und wie viele Punkte in Summe abzufahren sind. Je nachdem erfolgt die Interpolation der Punkte für eine Strecke zwischen zwei Punkten auf verschiedene Art und Weise. Es ergeben sich vier verschiedene Geschwindigkeitsverläufe (s. Abbildung 7-10). Entsprechend des Geschwindigkeitsverlaufs a ist zu interpolieren, wenn der Pfad weiterhin nur aus Start- und Zielpunkt besteht. Die Interpolation erfolgt nach b, wenn mehr als zwei Punkte im Pfad inkludiert sind und der betrachtete Punkt der Startpunkt ist. Sobald mehr als drei Punkte im Pfad enthalten sind und der Streckenabschnitt zwischen zwei Via-Punkten aufgespannt wird, sollte die Interpolation nach den Geschwindigkeitsverlauf c erfolgen. Bei mehr als zwei Punkte sowie dem letzten Streckenabschnitt ist eine Interpolation nach dem Geschwindigkeitsverlauf d durchzuführen.

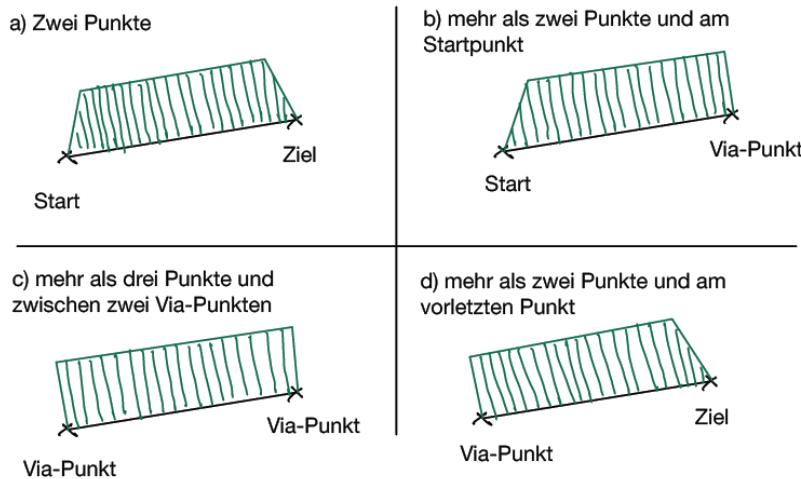


Abbildung 7-10: Geschwindigkeitsverläufe der Streckenabschnitte

In dem Skript der Bahnplanung wurde bereits eine Funktion für die Interpolation a eingeführt. Dementsprechend sind Funktionen zur Interpolation für die restlichen Geschwindigkeitsverläufe einzuführen. Die Gleichungen für die einzelnen Phasen werden analog zur ersten Interpolation ermittelt. Auf eine Herleitung wird an dieser Stelle verzichtet. Für die einzelnen Interpolationen sowie der Gesamtzeit und der neuen Geschwindigkeit ergeben sich folgende Gleichungen:

#### Geschwindigkeitsverlauf b:

$$\text{Gesamtzeit} \quad t_e = \frac{s_e}{v_m} + 0,5 * \frac{v_m}{a_m} \quad (7-5)$$

$$\text{Geschwindigkeit} \quad v_{m,i} = a_{m,i} * t_e - \sqrt{a_{m,i}^2 * t_e^2 - 2 * s_{e,i} * a_{m,i}} \quad (7-6)$$

$$s(t) = \begin{cases} \frac{1}{2} * a_m * t^2 & 0 \leq t \leq t_a \\ v_{m,i} * (t - \frac{1}{2} * \frac{v_{m,i}}{a_m}) & t_a \leq t \leq t_e \end{cases} \quad (7-7)$$

$$v(t) = \begin{cases} a_m * t & 0 \leq t \leq t_a \\ v_m & t_a \leq t \leq t_e \end{cases} \quad (7-8)$$

#### Geschwindigkeitsverlauf c:

$$\text{Gesamtzeit} \quad t_e = \frac{v_m}{s_e} \quad (7-9)$$

$$\text{Geschwindigkeit} \quad v_{m,i} = \frac{s_e}{t_e} \quad (7-10)$$

$$s(t) = v_{m,i} * t \quad (7-11)$$

$$v(t) = v_{m,i} \quad (7-12)$$

**Geschwindigkeitsverlauf d:**

Gesamtzeit

$$t_e = \frac{s_e}{v_m} + 0,5 * \frac{v_m}{a_m} \quad (7-13)$$

Geschwindigkeit

$$v_{m,i} = a_{m,i} * t_e - \sqrt{a_{m,i}^2 * t_e^2 - 2 * s_{e,i} * a_{m,i}} \quad (7-14)$$

$$s(t) = \begin{cases} v_{m,i} * t & 0 \leq t \leq t_b \\ t_b * v_{m,i} + v_{m,i}(t - t_b) - 0,5 * a_m * (t - t_b)^2 & t_b \leq t \leq t_e \end{cases} \quad (7-15)$$

$$v(t) = \begin{cases} v_{m,i} & 0 \leq t \leq t_b \\ v_{m,i} - a_m * (t - t_b) & t_b \leq t \leq t_e \end{cases} \quad (7-16)$$

Durch if-Abfragen wird ermittelt, an welchem Punkt des Pfades sich das Fahrzeug aktuell befindet (s. Programmcode 7-6). Für den jeweiligen Streckenabschnitt werden erneut die gesamte Zeit, die neuen Geschwindigkeiten und die Beschleunigungszeit sowie Bremszeitpunkt berechnet. Im Anschluss daran erfolgt die Interpolation und das Abfahren des Streckenabschnittes.

```

239     if len(Pfad)>2 and p==1:
240
241     ...
242     v
243     |
244     |
245     /|_____
246     | |
247     | /|_____|_____
248     p0   t_a           p1
249     ...

```

Programmcode 7-6: Pfadplanung "Geschwindigkeitsverlauf eines Streckenabschnitts"

Um die Interpolation und das Abfahren der Strecke durchführen zu können, muss die Funktion „Bahnplanung“ zunächst aufgerufen werden. Zudem wurden in der Pfadplanung lediglich die x-Koordinaten und y-Koordinaten der einzelnen Via-Punkte festgelegt, jedoch nicht die Orientierung. Daher wird neben der Liste für die Punkte im Pfad eine weitere Liste für die Orientierung phi erzeugt (s. Programmcode 7-7). Es wird festgelegt, dass mit jedem Streckenabschnitt ein n-ter-Anteil der Gesamtdrehung zwischen Start- und Zielorientierung zurückgelegt werden soll. Mit Hilfe einer for-Schleife werden die entsprechenden Orientierungen an den Via-Punkten errechnet.

```

686 #Abfahren des Pfades
687 delta_phi=(phi_Ziel-phi_start)/(len(Pfad)-1)
688 phi_Pfad=[]
689 for z in range(len(Pfad)):
690     phi=phi_start+z*delta_phi
691     phi_Pfad.append(phi)
692 print("Phi entlang Pfad: "+str(phi_Pfad))
693
694 p=1
695 print("Länge Pfad: "+str(len(Pfad)))
696 v=[150]
697 omega=[3]
698
699 while p < len(Pfad):
700     [v_neu,omega_neu]=Bahnplanung(Pfad[p][0],Pfad[p][1],phi_Pfad[p],v[p-1],omega[p-1])
701     v.append(v_neu)
702     omega.append(omega_neu)
703     print("Am Punkt: "+str(p))
704     p=p+1
705
706 print("Am Ziel!!!")

```

Programmcode 7-7: Pfadplanung "Abfahren des Pfades"

Das Abfahren der einzelnen Punkte erfolgt mittels *while*-Schleife. Die *while*-Schleife wird so lange ausgeführt, bis der Zielpunkt erreicht ist. Dafür wird eine Laufvariable *p* sowie Listen mit den Geschwindigkeiten bzw. Drehgeschwindigkeiten an jedem Punkt aufgestellt. Die Laufvariable *p* erhält den Wert eins. Auf die Funktionsweise der Geschwindigkeitslisten wird später eingegangen.

Die Laufvariable *p* wird in der *while*-Scheife genutzt, um zu bestimmen, welche Position und Orientierung an die Bahnplanung übergeben werden soll. Die Bahnplanung wird ausgeführt und das Fahrzeug fährt den Streckenabschnitt zwischen dem aktuellem Punkt und dem übergebenen Punkt ab. Zudem werden in der Funktion „Bahnplanung“ nun Rückgabewerte eingeführt. Es werden die Geschwindigkeit sowie Drehgeschwindigkeit am letzten interpolierten Punkt zurückgegeben. Diese Geschwindigkeiten werden den zuvor eingeführten Listen hinzugefügt und beim nächsten Durchlaufen der *while*-Schleife mit an die Funktion „Bahnplanung“ übergeben. Dadurch ist für Ausgangsgeschwindigkeit für den nächsten Streckenabschnitt bekannt.

Die Abbildung 7-11 zeigt die Notwendigkeit hierfür. Wenn sich das Fahrzeug an einem Via-Punkt befindet und die vorherige Geschwindigkeit nicht übernommen wird, berechnet sich das System eine neue Geschwindigkeit die vollkommen unabhängig von der vorherige Geschwindigkeit ist. Dadurch kann es zu einem Sprung im Geschwindigkeitsverlauf kommen, sodass dieser nicht mehr stetig ist. Die Anforderung „Steitiger Geschwindigkeitsverlauf“ ist somit nicht mehr erfüllt. Zudem muss das Fahrzeug unerwartet beschleunigen oder abbremsen. Diese Vorgänge sind jedoch bei der Interpolation nicht vorgesehen.

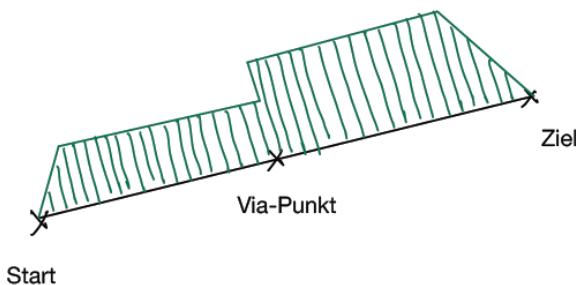


Abbildung 7-11: Geschwindigkeitsdifferenz an Via-Punkt

Nachdem sich das Fahrzeug an Zielposition und in Zielorientierung befindet, soll dieses weiterhin seine Lage halten. Dazu wird wieder eine Lageregelung durchgeführt, welche jedoch nicht näher beschrieben wird. Der komplette Programmcode für die Pfadsuche sowie das Abfahren des Pfades befindet sich in Anhang G.

## 8 Schlussbetrachtung

Dieses Kapitel enthält eine Zusammenfassung der Masterarbeit. Dabei wird auf zentralen Inhalte und Ergebnisse eingegangen. Zusätzlich erfolgt eine kritische Betrachtung und Diskussion der Ergebnisse. Den Abschluss bildet ein Ausblick auf mögliche Weiterentwicklungen samt Folgearbeiten der erarbeiteten Ergebnisse.

### 8.1 Zusammenfassung

Die vorliegende Ausarbeitung behandelt die Entwicklung einer Bewegungssteuerung für eine Fahrerloses Transportsystem. Das Ergebnis dieser Arbeit sind Programmcodes der Bewegungssteuerung. Die Zielerreichung und Ergebniserstellung erfolgt in mehreren Kapitel bzw. Abschnitten.

Im ersten Kapitel wird die Problemstellung, die Zielstellung und der grobe Arbeitsablauf dargestellt. Die Bewegungssteuerung eines Roboters hat die Aufgabe Programmcodes entgegenzunehmen und abzuarbeiten, um daraus Bewegungsbefehle zu generieren. Sie gewannen daher im Zuge der Industriealisierung an Bedeutung. Das vorliegende fahrerlose Transportsystem untergliedert sich im ein fahrerloses Transportfahrzeug sowie ein Trackingsystem, welche von der Hochschule zu Verfügung gestellt werden. Für diese ist eine Bewegungssteuerung zu implementieren.

Im zweiten Kapitel wird die Vorgehensweise in dieser Arbeit betrachtet. Die Umsetzung der Arbeit basiert auf der agilen Projektmanagementmethode *Scrum*. In der *Scrum*-Methode wird ein *Product Backlog* mit Anforderungen an eine Bewegungssteuerung beschrieben. Diese Anforderungen werden iterativ in kleinen *Sprints* umgesetzt. Für jeden *Sprint* werden die Anforderungen in einem *Sprint Backlog* zusammengefasst. Für die Bewegungssteuerung ergeben sich in Summe vier *Sprints*. Von diesen sind drei *Sprints* Bestandteil dieser Arbeit. Im ersten *Sprint* werden die Schnittstellen analysiert, der zweite *Sprint* beschäftigt sich mit einer Punkt-zu-Punkt-Bewegung und der letzte *Sprint* beinhaltet die Bewegungssteuerung einer Mehrpunktbewegung.

Das dritte Kapitel beschreibt die Ausgangssituation der Arbeit. Die Ausarbeitung der Bewegungssteuerung wird anhand eines FTFs von Nexus Robotik und einer Trackingkamera durchgeführt. Das Fahrzeug inkludiert Mecanum Räder sowie Ultraschallsensoren. Die Ansteuerung und Funktionsweise dieser Komponenten werden daher, unterstützt durch theoretische Grundlagen, erläutert. Zudem werden von Nexus diverse Bibliotheken und ein Demoskript bereitgestellt, welche ebenfalls erklärt werden. Neben der Begutachtung des FTF wird des Weiteren kurz auf die Funktionsweise der Trackingkamera eingegangen.

Auf Basis der analysierten Ausgangssituation wird im vierten Kapitel ein *Product Backlog* erstellt. Die einzelnen Anforderungen werden hier jedoch nicht näher umschrieben. Vielmehr werden sie priorisiert und somit nach den verschiedenen *Sprints* geordnet. Auf die Anforderungen selbst wird in den jeweiligen Kapiteln zur Umsetzung der *Sprints* eingegangen.

Das fünfte Kapitel umfasst den ersten *Sprint*. In dem dazugehörigen *Sprint Backlog* werden die Anforderungen beschrieben. Es werden die vorhandenen Schnittstellen des Fahrzeugs und der Trackingkamera analysiert. Die Trackingkamera arbeitet über REST API. Das Fahrzeug bindet verschiedene Möglichkeiten zur Implementierung einer Schnittstelle ein, u.a. für ein Bluetooth-Modul. Für die Datenübertragung zum Fahrzeug wird das Bluetooth-Modul HC-05 ausgewählt. Über ein Python-Skript werden mittels HTTP-Befehle die Fahrzeugkoordinaten angefragt und ausgegeben. Die Erteilung der Bewegungsbefehle erfolgt über selbiges Skript. Zur Umsetzung der Bewegungssteuerung wird

zunächst die Hardware implementiert und das Trackingsystem kalibriert. Der im Anschluss entwickelt Programmcode übernimmt die soeben beschrieben Aufgaben der Bewegungssteuerung.

Im sechsten Kapitel wird der zweite *Sprint* mit seinen Anforderungen thematisiert. Die Anforderungen beinhalten die Entwicklung einer Trajektorienplanung einer Punkt-zu-Punkt-Bewegung sowie die Einführung einer Lageregelung. Dafür werden zunächst die theoretischen Grundlagen Kaskadenregelung, allgemeine Trajektorienplanung, Trajektorienplanung einer Punkt-zu-Punkt-Bewegung sowie die Transformation von Koordinatensystemen beschrieben. Zudem werden jeweils Formeln für die Interpolation und Transformation hergeleitet. Diese Formeln finden Verwendung in den Programmcodes der Bewegungssteuerung. Zunächst wird ein Programmcode für die Lageregelung entwickelt. Anschließend ein Arduino-Sketch für die Entgegennahme der Bewegungsbefehle weiterentwickelt und zuletzt wird der Programmcode für die Trajektorienplanung implementiert. Dieser beinhaltet die Lageregelung sowie eine Schnittstelle zur Datenübertragung.

Das siebte und letzte Kapitel beleuchtet die autonome Pfadsuche. Dazu werden verschiedene Pfadplanungsalgorithmen betrachtet. Diese Algorithmen haben das Ziel einen geeigneten Weg zwischen einem Start- und Endpunkt aufzubauen. Nach einem Vergleich der Algorithmen wird für diese Arbeit die Umsetzung eines RRT-Algorithmus ausgewählt und entsprechende Heuristiken festgelegt. Bei der Pfadplanung ergibt sich eine Pfad über Via-Punkte, sodass eine Bewegungssteuerung für eine Mehrpunkt-Bewegung benötigt wird. Zur Umsetzung dieser Bewegung wird das bisherige Skript der Bahnplanung erweitert. Die Mehrpunkt-Bewegung erfolgt in einer statischen Umgebung mit zeitlich unveränderlichen Hindernissen. Diese Hindernisse sind zunächst virtuell und werden vom Anwender vorgegeben.

## 8.2 Kritische Betrachtung

Die im Zuge dieser Arbeit entwickelten Programmcodes realisieren eine Bewegungssteuerung. Diese Bewegungssteuerung erfüllt die gestellten Anforderungen im *Product Backlog* bzw. der einzelnen *Sprint Backlog* mit einer Ausnahme. Die Ausnahme hierbei bildet die Anforderung „Stetiger Geschwindigkeitsverlauf“. Es werden bereits einige Maßnahmen getroffen, um diese Anforderung im Allgemeinen zu erfüllen, jedoch werden nicht alle Randbedingungen berücksichtigt.

Es wird im Laufe der Arbeit der Pfadplanungsalgorithmus RRT entwickelt. Dieser generiert einen Pfad oftmals über mehrerer Via-Punkte. Allerdings sind beim RRT-Algorithmus weitere Optimierungsmaßnahmen erforderlich, um eine stetige Bewegung zu generieren. Eine dieser Maßnahmen ist das Verschleifen von Punkten, um eine stetigen Übergang zwischen den Punkten und einen gleichmäßigen Verlauf zu erhalten (s. Abbildung 8-1 oben). Zudem kann es passieren, dass die Anforderung aufgrund von zu hoher Geschwindigkeit bei Kurvenfahrt nicht erfüllt ist. Das Fahrzeug kommt von der Bahn ab, wenn zu hohe Zentrifugalkräfte auf das Fahrzeug wirken. Die Zentrifugalkräfte können durch Vergrößerung des Kurvenradius oder Reduzierung der Geschwindigkeit reduziert werden (s. Abbildung 8-1 unten). All solche Randbedingungen bzw. Optimierungen werden in dieser Arbeit nicht thematisiert, haben jedoch Einfluss auf eine stetige Bewegung.

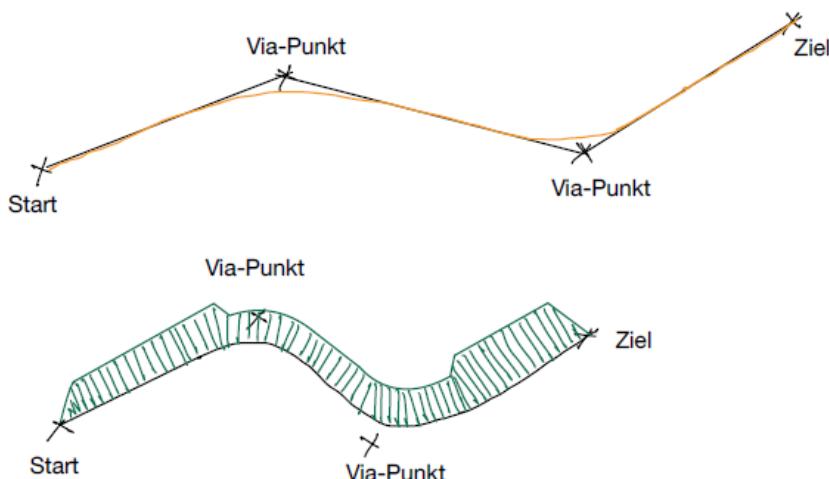


Abbildung 8-1: Optimierungsmaßnahmen an Via-Punkten

Innerhalb des *Product Backlogs* werden keine näheren technischen Anforderungen aufgestellt. Technische Anforderungen beinhalten z.B. die Genauigkeit des Systems. Daher gibt es keine genauen Vorgaben für das entwickelte System. Allerdings wurde mit der Lageregelung bereits der erste Schritt getätigkt, um eine hohe Genauigkeit zu erzielen. Jedoch spielen noch viele weitere Faktoren eine wichtige Rolle, um eine hohe Genauigkeit zu erzielen. Diese sind jedoch nicht Bestandteil dieser Arbeit.

Um zu überprüfen, ob die Anforderungen erfüllt werden, wurde die Bewegungssteuerung getestet. Diese Test erfolgen innerhalb des zur Verfügung stehenden Arbeitsraums. Sie dienen zur Weiterentwicklung und Korrektur des Programmcodes und zielen auf die Überprüfung der einzelnen Funktionen der Bewegungssteuerung ab. Die Bewegungssteuerung funktioniert grundsätzlich ohne Beeinträchtigung. Allerdings ist die gegebene Testumgebung sehr klein im Verhältniss zur Fahrzeuggröße und wird durch die Verwendung von Stativen noch minimiert. Daher ist es sinnvoll weitere systematische und gezielte Tests in einer größeren Umgebung durchzuführen.

Während der Test ist auffällig, dass das Fahrzeug aufgrund seiner Beschichtung dazu neigt, an beliebigen Stellen zu reflektieren. Dies führt dazu, dass die Trackingkamera Marker identifiziert, welche nicht vorhanden sind. Dadurch kann das System kurzfristig durcheinander kommen. Daher ist es erforderlich, dass Fahrzeug mit einem anderen Material zu beschichten, um diesen Fehler zu eliminieren.

Der entwickelte Programmcode der Bewegungssteuerung inkludiert diverse Funktionen wie z.B. die Lageregelung oder die Interpolation. Diese werden im Skript als eigenständige Funktionen definiert und können daher auch unabhängig vom übergeordneten Programmcode abgerufen werden. Dadurch können sie im einzelnen getestet werden, jedoch auch in übergeordneten Programmcodes eingebunden werden. Selbiges gilt für Programmteile, welche immer wieder benötigt werden, wie beispielsweise die Berechnung einer Distanz. Auch für solche Aufgaben werden Funktionen eingeführt, sodass nicht der gleiche Code immer wieder formuliert werden muss, sondern lediglich die Funktion aufgerufen werden kann. Dies ermöglicht einen strukturierten und geordneten Programmablauf mit wenig redundanten Beschreibungen.

Final kann festgestellt werden, dass ein funktionierender Programmcode für die Bewegungssteuerung entwickelt wurde, welcher nicht nur eine Punkt-zu-Punkt-Bewegung, sondern sogar eine Mehrpunkt-bewegung beinhaltet. Das Ziel der Aufgabenstellung „Entwicklung einer Bewegungssteuerung für eine Punkt-zu-Punkt-Bewegung“ wurde vollumfänglich erreicht. Darüber hinaus ist das Ziel der Arbeit die Entwicklung einer Bewegungssteuerung für eine Mehrpunkt-Bewegung. Hierfür bedarf der Programmcode noch ein wenig Optimierungspotential. Dafür gewinnt das Fahrzeug aufgrund der selbstständigen Pfadsuche an Autonomität. Der Aufbau des Programmcodes ist gut strukturiert und einfach verständlich. Er bildet eine gute Grundlage und Struktur für eine Weiterentwicklung und erfüllt alle Kriterien für eine Bewegungssteuerung.

### 8.3 Ausblick

Wie in der kritischen Betrachtung beschrieben, bestehen noch einige Verbesserungspotentiale hinsichtlich der entwickelten Bewegungssteuerung. Unter anderem werden die Funktionen der Bewegungssteuerung durch Tests validiert. Allerdings ist es erforderlich, weitere Test durchzuführen, um die Fehleranfälligkeit der Steuerung zu überprüfen. Zudem bestehen noch einige Optimierungspotentiale hinsichtlich der Fahrzeugbewegung selbst. Diese gilt es, in weiteren Arbeiten durchzuführen, bevor dann die Bewegungssteuerung weiterentwickelt werden kann.

Für die Weiterentwicklung der Steuerung sind im *Product Backlog* noch einige Anforderungen enthalten. Diese eignen sich für einen oder mehrere weitere *Sprints*. Die Anforderungen beschreiben zum einen das Fahren in dynamischen Umgebungen mit realen Hindernissen, zum anderen die Einführung eines User Interface zur Darstellung und Simulation der Bewegung sowie zum Täglichen von Eingaben durch den Anwender. Diese Anforderungen sind in Tabelle 8-1 gelistet.

Tabelle 8-1: Offene Anforderungen

Offene Anforderungen des Product Backlogs	
Nr.	Anforderung
1	Fahren von Kurven
2	Kollisionsfreies Fahren in einer dynamischen Umgebung
3	Dynamische Hindernisse detektieren
4	User Interface
5	Visualisierung/Simulation des Fahrweges des FTS im Konfigurationsraum
6	Verschleifen von Punkten entlang des Fahrweges
7	Stetiger und Ruckfreier Geschwindigkeitsverlauf

In diesen Anforderungen sind nochmals das Verschleifen von Punkten und der stetige sowie ruckfreie Geschwindigkeitsverlauf sowie das Fahren von Kurven aufgeführt. Für einen ruckfreien Geschwindigkeitsverlauf ist nicht nur ein Rampenprofil erforderlich. Das bedeutet die Interpolation muss nochmal überarbeitet werden und zum Beispiel durch ein Sinoidenprofil ausgetauscht werden.

Für das Fahren in dynamischen Umgebungen ist der Einsatz der Ultraschallsensoren relevant. Den Ultraschallsensoren selbst werden in dieser Arbeit wenig Beachtung geschenkt, da sie für eine statische Umgebung nicht erforderlich sind. Daher wurde ihre Kommunikationsschnittstelle RS485 ausgeschaltet und die Pins für die Schnittstelle des Bluetooth-Moduls genutzt. Für Verwendung der Ultraschallsensoren zum Detektieren von dynamischen Hindernissen sollte folglich eine geeignete Lösung gefunden werden.

## 9 Literaturverzeichnis

- 4D SAS. (2023). *Overview of JSON commands*. Von <https://doc.4d.com/4Dv19/4D/19.6/Overview-of-JSON-commands.300-6270054.en.html> abgerufen am 31.08.2023
- Arduino. (2023). *Libaries*. Von <https://www.arduino.cc/reference/en/libraries/> abgerufen am 31.08.2023
- Atmel. (2015). *ATmega328P*. Von [https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P\\_Datasheet.pdf](https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf) abgerufen am 31.08.2023
- AZ-Delivery. (2023). *HC-05 HC-06*. Von <https://www.az-delivery.de/products/hc-05-6-pin> abgerufen am 31.08.2023
- Barr, M. (09. 01 2001). *Introduction to Pulse Width Modulation (PWM)*. Von <https://barrgroup.com/embedded-systems/how-to/pwm-pulse-width-modulation> abgerufen am 31.08.2023
- Bartmann, E. (2012). *Die elektronische Welt mit Arduino entdecken*. Köln: O'Reilly.
- Baumgarten, M. (1990). *Grundsatzanalyse des Mecanum Rades*. Von <https://www.innorad.de/Daten/Interna/Literatur/RaederRollen/Grundsatzanalyse%20des%20Mecanum-Rades%20-%20Markus%20Baumgarten,%20Braunschweig.pdf> abgerufen am 31.08.2023
- Bies, L. (2017). *RS485 serial information*. Von <https://www.lammertbies.nl/comm/info/rs-485#intr> abgerufen am 31.08.2023
- Bohlin, R., & Kavraki, L. (2000). Path Planning Using Lazy PRM. *International Conference on Robotics and Automation (ICRA)* (S. 521-528). San Francisco: IEEE .
- Borke, P. (10 1988). *Points, lines, and planes*. Von <https://paulbourke.net/geometry/pointlineplane/> abgerufen am 31.08.2023
- Bräunl, T. (2022). *Embedded Robotics (Fourth Edition)*. Singapore: Springer Nature .
- Bruhlmann, T. (2012). *Arduino Praxiseinstieg behandelt Arduino 1.0*. Heidelberg: Verlagsgruppe Hüthig Jehle Rehm.
- Carrasco, D. (24. 03 2021). *PCINT on arduino*. Von <https://www.electrosoftcloud.com/en/pcint-interrupts-on-arduino/> abgerufen am 31.08.2023
- Choset, H., Lynch, K., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L., & Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations*. London: The MIT press.
- Components101. (16. 07 2021). *HC-05 - Bluetooth Module*. Von <https://components101.com/wireless/hc-05-bluetooth-module> abgerufen am 31.08.2023
- DroneBot-Workshop. (10. 05 2022). *Vwendung von Arduino-Interrupt*. Von <https://dronebotworkshop.com/interrupts/> abgerufen am 31.08.2023
- Ericson, E. (04. 04 2022). *Detecting movement direction with two ultrasonic distance sensors*. Von <https://www.sensingthecity.com/detecting-movement-direction-with-two-ultrasonic-distance-sensors/> abgerufen am 31.08.2023
- FH Bielefeld. (2017). *REST*. Von <https://fh-bielefeld-mif-sw-engineering.gitbooks.io/script/content/embedded-computing/IOT-Protokolle/Rest.html> abgerufen am 31.08.2023

fonial GmbH. (2023). *World Wide Web*. Von <https://www.fonial.de/wissen/begriff/www/> abgerufen am 31.08.2023

Gammon, N. (08 2015). *Atmega328P Timer bit-pattern charts for download*. Von <https://forum.arduino.cc/t/atmega328p-timer-bit-pattern-charts-for-download/328377> abgerufen am 31.08.2023

Generation Robotics. (2023). *Rechts Mecanum Rad*. Von <https://www.generationrobots.com/de/403135-rechtes-mecanum-rad-100-mm.html> abgerufen am 31.08.2023

Hinzmann, T. (2011). *Pfadplanung in Innenräumen und Integration in eine Simulationsumgebung für autonome Bodenfahrzeuge*. Karlsruhe: KIT.

Hirzel, T. (03. 08 2023). *Basics of PWM (Pulse Width Modulation)*. Von <https://docs.arduino.cc/learn/microcontrollers/analog-output> abgerufen am 31.08.2023

Hofschulte, J. (17. 09 2022). Robotik Grundlagen. *Vorlesungsskript*. Hannover: HSH.

Hofschulte, J., & Waldt, N. (10. 11 2021). Workshop Erweiterte Steuerungstechnik/Cyberphysische Systeme. *Skript*. Hannover: HSH.

Intellinet. (2023). *Ein Leitfaden für Power over Eternet (PoE) Switches*. Von <https://intellinetnetwork.de/pages/poe-switches-technologie> abgerufen am 31.08.2023

Jäisch, R., & Donges, J. (2017). *Mach was mit Arduino*. München: Hanser.

Kanjanawanishkul, K. (02 2015). Omnidirectionals wheeled mobile robots: Wheel types and practical applications. *International Journal of Advanced Mechatronic Systems*, S. 289-302.

Kavraki, L. E., & LaValle, S. (2016). Motion Planning. In B. Siciliano, & O. Khatib, *Springer Handbook of Robotics* (S. 139-162). Heidelberg: Springer .

Kollmorgen. (2023). *Servomotoren*. Von [https://www.kollmorgen.com/de-de/products/motors/servo/servomotoren?utm\\_source=google&utm\\_medium=cpc&utm\\_campaign=DE-servo-motors-nonbrand&utm\\_term=servomotoren&matchtype=e&utm\\_content=&gad=1&gclid=EAIaIQobChMI3fGMyL3AgAMVQoIQBh354AliEAAYAiAAEgL1L\\_D\\_BwE](https://www.kollmorgen.com/de-de/products/motors/servo/servomotoren?utm_source=google&utm_medium=cpc&utm_campaign=DE-servo-motors-nonbrand&utm_term=servomotoren&matchtype=e&utm_content=&gad=1&gclid=EAIaIQobChMI3fGMyL3AgAMVQoIQBh354AliEAAYAiAAEgL1L_D_BwE) abgerufen am 31.08.2023

Koren, Y., & Borenstein, J. (1991). *Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation*. Sacramento: IEEE Conference on Robotics and Automation.

Kuffner James, & LaValle, S. (2000). RRT-Connect: An Efficient Approach to Single-Query Path Planning. *International Conference on Robotics and Automation (ICRA)* (S. 1-8). San Francisco: IEEE.

Kunbus. (2023). *Baudraten Grundlagen*. Von <https://www.kunbus.de/baudraten-grundlagen#:~:text=Die%20Baudrate%20ist%20eine%20Einheit,Datenendeinrichtung%20und%20der%20Daten%C3%BCbertragungseinrichtung%20aufbaut>. abgerufen am 31.08.2023

Künemund, F. (2017). *Navigationssystem für holonome Fahrerlose Transportfahrzeuge mit multikritieller Optimierung*. Düsseldorf: VDI Verlag.

Lackes, R. (19. 02 2018). *World Wide Web*. Von <https://wirtschaftslexikon.gabler.de/definition/world-wide-web-www-49260> abgerufen am 31.08.2023

Leidner, D. (2011). *Pfadplanung für Aufgaben in der realen Welt ausgeführt durch reale Robotersysteme*. Mannheim: DLR.

- Linde. (2023). *Automatisch unterwegs*. Von <https://www.linde-mh.de/de/Loesungen/Intralogistik-Automatisierung/Fahrerlose-Transportsysteme/> abgerufen am 31.08.2023
- Mareczek, J. (2020). *Grundlagen der Roboter-Manipulatoren – Band 2*. Berlin: Springer Vieweg.
- Margolis, M. (2012). *Arduino Kochbuch*. Köln: O'Reilly.
- mwwalk. (11. 08 2014). *Arduino Pin Change Interrupts*. Von <https://thewanderingengineer.com/2014/08/11/arduino-pin-change-interrupts/> abgerufen am 31.08.2023
- Natrual Point. (2010). *S250e Quick Start Guide*. Von <https://d111srqycjesc9.cloudfront.net/S250e%20Quick%20Start%20Guide.pdf> abgerufen am 31.08.2023
- NatrualPoint Inc. (2012). *Data Sheet S250e*. Von <https://optitrack.com/support/hardware/s250e.html> abgerufen am 31.08.2023
- NatrualPoint Inc. (2023). *Allgemeine FAQs*. Von <https://optitrack.com/support/faq/general.html> abgerufen am 31.08.2023
- Nexus Robot. (2012). *Robot Kits User's Manual*. Von <https://www.nexusrobot.com/product/4wd-mecanum-wheel-mobile-arduino-robotics-car-10011.html> abgerufen am 31.08.2023
- Odendahl, M., Finn, J., & Wenger, A. (2010). *Arduino-Physical Computing für Bastler*. Köln: O'Reilly.
- Parmar, N. (03. 06 2022). *API, REST API and RESTful API*. Von <https://wirtschaftslexikon.gabler.de/definition/world-wide-web-www-49260> abgerufen am 31.08.2023
- Preußig, J. (2020). *Agiles Projektmanagement*. Freiburg: Haufe Verlag.
- Probst, U. (2022). *Servoantriebe in der Automatisierungstechnik (3.Auflage)*. Wiesbaden: Springer Vieweg.
- RN-Wissen. (23. 06 2023). *Regelungstechnik*. Von <https://rn-wissen.de/wiki/index.php/Regelungstechnik> abgerufen am 31.08.2023
- Sauter, M. (2022). *Grundkurs Mobile Kommunikationssysteme (8.Auflage)*. Wiesbaden: Springer Vieweg.
- Seobility. (2023). *REST-API*. Von [https://www.seobility.net/en/wiki/REST\\_API](https://www.seobility.net/en/wiki/REST_API) abgerufen am 31.08.2023
- Swift, N. (28. 02 2017). *Easy A\* (star) Pathfinding*. Von <https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2> abgerufen am 31.08.2023
- Tobola, A. (01. 09 2014). *Regelungstechnik*. Von <https://tnotes.de/ControlPage> abgerufen
- Weis, O. (21. 10 2021). *RS485-Kommunikationsanleitung*. Von <https://www.eltima.com/de/article/rs485-communication-guide/> abgerufen am 31.08.2023
- Wenz, M. (2008). *Automatische Konfiguration der Bewegungssteuerung von Industrierobotern*. Karlsruhe: TH Karlsruhe.
- Winkelmann, P. (2000). *Definition des CRM*. Deutscher Direktmarketing Verband in Kooperation mit der Fachhochschule Landshut.

Winkler, A. (2016). *Sensorgeführte Bewegungen stationärer Roboter*. Chemnitz: Universitätsverlag Chemnitz.

Wirdemann, R., & Mainusch, J. (2017). *Scrum mit User Stories*. München: Hanser Verlag.

Yadav, A. (14. 11 22). *RRT*. Von <https://github.com/nimRobotics/RRT> abgerufen am 31.08.2023

## 10 Anhang

Anhang A: Datenblatt FTF

Anhang B: Demoskript

Anhang C: Arduino-Skript für Fahrbewegung Sprint #1

Anhang D: Arduino-Skript für Fahrbewegung Sprint #2

Anhang E: Skript Lagerregelung

Anhang F: Skript Bahnplanung für Punkt-zu-Punkt-Bewegung

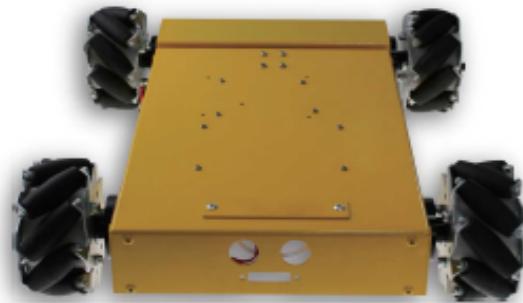
Anhang G: Skript Pfadplanung und Fahrbewegung entlang Pfad

## Anhang A: Datenblatt FTF



### 4WD 100mm Mecanum wheel robot kit 10011

This robot kit is based on a 4 wheels driving mecanum wheels chassis. the vehicle is stable and can be made to move in any direction and turn by varying the direction and speed of each wheel. Moving all four wheels in the same direction causes forward/backward movement, running left/right sides in opposite directions causes rotation, and running front and rear in opposite directions causes sideways movement. Combined motion is also possible. Its rear wheels are mounted in a particular way, this simple suspension structure ensures its four wheels can adhere to the ground even when the ground is uneven. This is very important for an mecanum wheels system. It also includes a microcontroller and motors with encoders.



Aluminum alloy frame



4 wheels drive



Mecanum wheel



Ultrasonic sensor



IR sensor



Encoder



Programmable



Easily expand

#### Features:

- 4 wheels drive
- Mecanum wheels
- Aluminum alloy body
- Suspension ensure 4 wheel adhere to ground
- Mounts reserve for Ultrasonic and Infrared sensor
- DC motors with encoders
- Includes Microcontroller and IO expansion board
- Capable of moving forward, backward, sideway, rotation

#### Parts included:

- 4 100mm Aluminum Mecanum Wheels
- Faulhaber 12V DC Coreless Motor
- Arduino 328 Controller
- Arduino IO Expansion V1.1
- 2 12V Ni-Mh Battery
- 12V Charger
- 4 100mm Aluminum Mecanum Wheels
- Faulhaber 12V DC Coreless Motor
- Arduino 328 Controller
- Arduino IO Expansion V1.1
- 2 12V Ni-Mh Battery
- 12V Charger

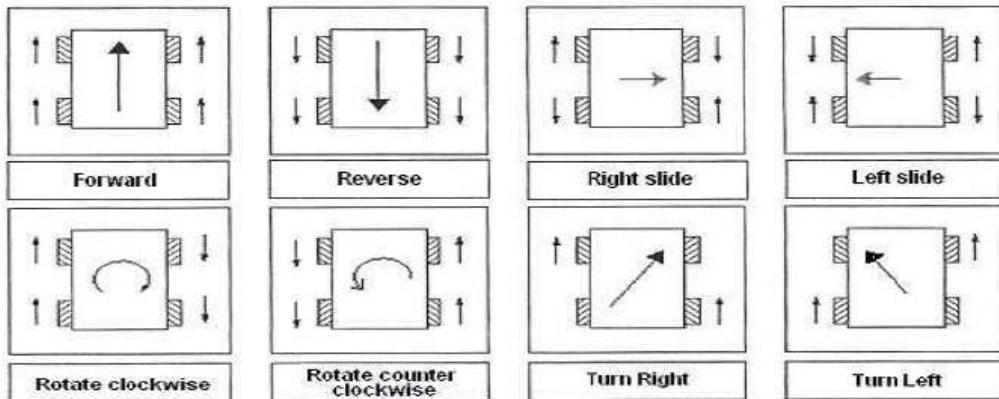




## 4WD 100mm Mecanum wheel robot kit 10011



### Co-effect of 4 Mecanum wheels:



### Specifications:

<b>Chassis</b>		
Appearance	Rectangle	
Length	400mm	
Max Width	360mm	
Height	100mm	
Chassis Height	22mm	
Wheel Base	300mm	
Wheel span	300mm	
Material	Aluminium	
Max Speed	0,6m/s	
Drive Mode	4 wheel drive	
Climbing Capacity	20 degree	
Load Capacity	20kg	
min-ITX Compatible	Yes	
<b>Wheel</b>		
Type	45 degree Mecanum Wheel	
Diameter	100mm	
Thickness	50mm	
Material	Aluminum Alloy and Rubber	
Load Capacity	15kg	
Roller Material	Rubber	
Diameter of Roller	29mm	
Length of Roller	47mm	
Coupled Mode	2 x 684ZZ bearings	

<b>Motor</b>		
Type	Faulhaber 12V DC	Brushless Motor
Power	17W	
RPM	120rpm	
Diameter	30mm	
Length	42mm	
Total Length	85mm	
Diameter of Shaft	6mm	
Length of Shaft	35mm	
No Load Current	75mA	
Load Current	1400mA	
Gearbox Ratio	64:1	

<b>Encoder</b>		
Type	Optical	
Encoder Phase	AB	
Encoder Resolution	12CPR	

<b>Battery and Charger</b>		
Battery	12V Ni-Mh	
Slow Charger	100-240V~24-12V Out	
Duration of Charge	2 hours	
Running Time	0,5 hour	

<b>Microcontroller Specification</b>		
Atmega 328		
14 Channels Digital I/O		
6 PWM Channels (Pin11,Pin10,Pin9,Pin6,Pin5,Pin3)		
8 Channels 10-bit Analog I/O		
USB interface		
Auto sensing/switching power input		
JTAG header for direct program download		
Serial Interface TTL Level		
Support AREF		
Support Male and Female Pin Header		
Integrated sockets for APC220 RF Module		
Five I2C Interface Pin Sets		
Two way Motor Drive with 2A maximum current		
7 key inputs		
DC Supply:USB Powered or External 7V-12V DC		
DC Output:5V /3,3V DC and External Power Output		
Dimension:90x80mm		

<b>IO expansion board</b>		
To support RS485 interface or drive 4 motors		

## Anhang B: Demoskript

```
#include <EEPROM.h>
#include <MotorWheel.h>
#include <Omni4WD.h>
#include <PID_Beta6.h>
#include <PinChangeInt.h>
#include <PinChangeIntConfig.h>
#include <SONAR.h>

/*
*****
Sonar:0x12
-----
|          |
M3          |          M2
|          |
|          |
|          |
|          |
Sonar:0x13 |          | Sonar:0x11
|          |
|          |
|          |
|          |
-----|          |
|          |
M4          |          M1
|          |
|          |
|          |
|          |
-----|          |
|          |
Sonar:0x14
*****
****

*/
irqISR(irq1,isr1);
MotorWheel wheel1(3,2,4,5,&irq1);

irqISR(irq2,isr2);
MotorWheel wheel2(11,12,14,15,&irq2);

irqISR(irq3,isr3);
MotorWheel wheel3(9,8,16,17,&irq3);

irqISR(irq4,isr4);
MotorWheel wheel4(10,7,18,19,&irq4);

Omni4WD Omni(&wheel1,&wheel2,&wheel3,&wheel4); //Alle Motoren inbegriffen

SONAR sonar11(0x11),sonar12(0x12),sonar13(0x13),sonar14(0x14);
```

```
unsigned short distBuf[4];

unsigned char sonarsUpdate() {
    static unsigned char sonarCurr = 1;
    if(sonarCurr==4) sonarCurr=1;
    else ++sonarCurr;
    if(sonarCurr==1) {
        distBuf[1]=sonar12.getDist();
        sonar12.showDat();
        sonar12.trigger();
    } else if(sonarCurr==2) {
        distBuf[2]=sonar13.getDist();
        sonar13.showDat();
        sonar13.trigger();
    } else if(sonarCurr==3){
        distBuf[3]=sonar14.getDist();
        sonar14.showDat();
        sonar14.trigger();
    } else {
        distBuf[0]=sonar11.getDist();
        sonar11.showDat();
        sonar11.trigger();
    }
    return sonarCurr;
}

void goAhead(unsigned int speedMMPS){
    if(Omni.getCarStat() != Omni4WD::STAT_ADVANCE) Omni.setCarSlow2Stop(300);
    Omni.setCarAdvance(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void turnLeft(unsigned int speedMMPS){
    if(Omni.getCarStat() != Omni4WD::STAT_LEFT) Omni.setCarSlow2Stop(300);
    Omni.setCarLeft(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void turnRight(unsigned int speedMMPS){
    if(Omni.getCarStat() != Omni4WD::STAT_RIGHT) Omni.setCarSlow2Stop(300);
    Omni.setCarRight(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void rotateRight(unsigned int speedMMPS){
    if(Omni.getCarStat() != Omni4WD::STAT_ROTATERIGHT) Omni.setCarSlow2Stop(300);
    Omni.setCarRotateRight(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
```

```
}

void rotateLeft(unsigned int speedMMPS){
    if(Omni.getCarStat() != Omni4WD::STAT_ROTATELEFT) Omni.setCarSlow2Stop(300);
    Omni.setCarRotateLeft(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void allStop(unsigned int speedMMPS){
    if(Omni.getCarStat() != Omni4WD::STAT_STOP) Omni.setCarSlow2Stop(300);
    Omni.setCarStop();
}

void backOff(unsigned int speedMMPS){
    if(Omni.getCarStat() != Omni4WD::STAT_BACKOFF) Omni.setCarSlow2Stop(300);
    Omni.setCarBackoff(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void(*motion[16])(unsigned int speedMMPS) = {goAhead, turnRight, goAhead,
turnRight,
turnLeft, goAhead, turnLeft, goAhead,
rotateRight, rotateRight, turnRight, turnRight,
rotateLeft, backOff, turnLeft, allStop};

unsigned long currMillis=0;
void demoWithSensors(unsigned int speedMMPS,unsigned int distance) {
    unsigned char sonarcurrent = 0;
    if(millis()-currMillis>SONAR::duration + 20) {
        currMillis=millis();
        sonarcurrent = sonarsUpdate();
    }

    if(sonarcurrent == 4){
        unsigned char bitmap = (distBuf[0] < distance); //right
        bitmap |= (distBuf[1] < distance) << 1; // back
        bitmap |= (distBuf[2] < distance) << 2; // left
        bitmap |= (distBuf[3] < distance) << 3; // front

        (*motion[bitmap])(speedMMPS);
        Serial.println(bitmap);
    }
}

Omni.PIDRegulate();
}

void setup() {
    delay(2000);
    TCCR1B=TCCR1B&0xf8|0x01;      // Pin9,Pin10 PWM 31250Hz
    TCCR2B=TCCR2B&0xf8|0x01;      // Pin3,Pin11 PWM 31250Hz
```

```
SONAR::init(13);
Omni.PIDEnable(0.35,0.02,0,10);
}

void loop() {
    demoWithSensors(160,20);
}
```

Anhang C: Arduino-Skript für Fahrbewegung Sprint #1

```
    Omni.setCarAdvance(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void turnLeft(unsigned int speedMMPS){
    if(Omni.getCarStat()!=Omni4WD::STAT_LEFT) Omni.setCarSlow2Stop(300);
    Omni.setCarLeft(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void turnRight(unsigned int speedMMPS){
    if(Omni.getCarStat()!=Omni4WD::STAT_RIGHT) Omni.setCarSlow2Stop(300);
    Omni.setCarRight(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void rotateRight(unsigned int speedMMPS){
    if(Omni.getCarStat()!=Omni4WD::STAT_ROTATERIGHT)
Omni.setCarSlow2Stop(300);
    Omni.setCarRotateRight(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void rotateLeft(unsigned int speedMMPS){
    if(Omni.getCarStat()!=Omni4WD::STAT_ROTATELEFT) Omni.setCarSlow2Stop(300);
    Omni.setCarRotateLeft(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void allStop(unsigned int speedMMPS){
    if(Omni.getCarStat()!=Omni4WD::STAT_STOP) Omni.setCarSlow2Stop(300);
    Omni.setCarStop();
}

void backOff(unsigned int speedMMPS){
    if(Omni.getCarStat()!=Omni4WD::STAT_BACKOFF) Omni.setCarSlow2Stop(300);
    Omni.setCarBackoff(0);
    Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void(*motion[7])(unsigned int speedMMPS) =
{goAhead,turnLeft,turnRight,backOff,rotateLeft,rotateRight,allStop};

void driveCar(unsigned speedMMPS){
    unsigned int direction;
    while(Serial.available()==0){

    }

    direction=Serial.readStringUntil('\n').toInt();
    (*motion[direction])(speedMMPS);
```

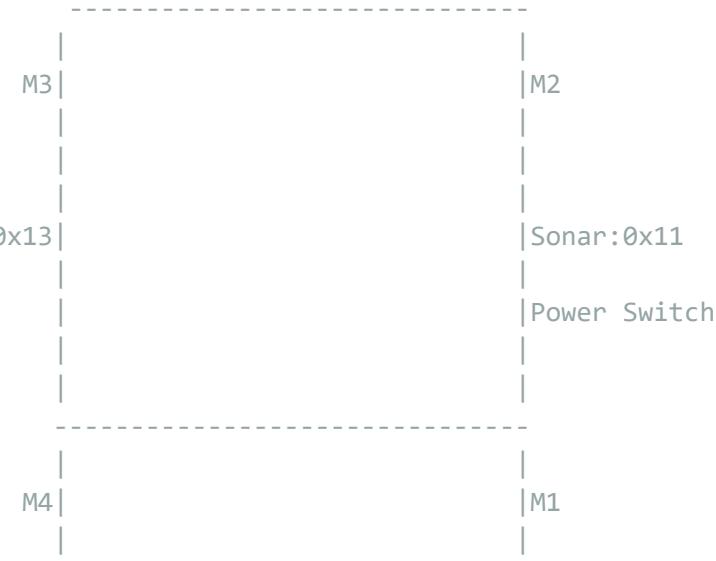
```
Omni.PIDRegulate();  
}  
  
void setup() {  
Serial.begin(19200);  
  
delay(2000);  
TCCR1B=TCCR1B&0xf8|0x01;      // Pin9,Pin10 PWM 31250Hz  
TCCR2B=TCCR2B&0xf8|0x01;      // Pin3,Pin11 PWM 31250Hz  
  
Omni.PIDEable(0.35,0.02,0,10);  
}  
  
void loop() {  
  
driveCar(100);  
}
```

## Anhang D: Arduino-Skript für Fahrbewegung Sprint#2

```
#include <EEPROM.h>
#include <MotorWheel.h>
#include <Omni4WD.h>
#include <PID_Beta6.h>
#include <PinChangeInt.h>
#include <PinChangeIntConfig.h>
```

```
/*
*****
*****
```

Sonar:0x12



Sonar:0x14

```
*****
*****
```

\*/

```
unsigned int speed;
unsigned int distance;
float direction;
float rotation;
int time;
String cmd;
int i,j,k;

irqISR(irq1,isr1);
MotorWheel wheel1(3,2,4,5,&irq1);

irqISR(irq2,isr2);
MotorWheel wheel2(11,12,14,15,&irq2);

irqISR(irq3,isr3);
MotorWheel wheel3(9,8,16,17,&irq3);
```

```
irqISR(irq4,isr4);
MotorWheel wheel4(10,7,18,19,&irq4);

Omni4WD Omni(&wheel1,&wheel2,&wheel3,&wheel4);

int count=0;
ISR(TIMER0_COMPA_vect){
    count++;
    if (count==100) {
        digitalWrite(13,!digitalRead(13));
        count=0;
    }
    Omni.PIDRegulate();
}

void setup() {
pinMode(13,OUTPUT);
Serial.begin(57600);
delay(2000);
TCCR1B=TCCR1B&0xf8|0x01;      // Pin9,Pin10 PWM 31250Hz
TCCR2B=TCCR2B&0xf8|0x01;      // Pin3,Pin11 PWM 31250Hz
OCR0A=100;
TIMSK0|=2;
Omni.PIDEable(0.35,0.02,0,10);
}

void loop() {
while(Serial.available()==0){
}
cmd=Serial.readStringUntil('\n');

i=cmd.indexOf(':');
speed=cmd.substring(0,i).toInt();
j=cmd.indexOf(':',i+1);
direction=cmd.substring(i+1,j).toFloat();
k=cmd.indexOf('\n');
rotation=cmd.substring(j+1,k).toFloat();

Omni.setCarMove(speed,direction,rotation);
}
```

## Anhang E: Skript Lageregelung

```
import requests
import serial
import time
import math
import json

def Lageregelung(x_soll,y_soll,phi_soll):
    kp=1.2
    k_phi=1

    r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
    obj = json.loads(r.text)
    x_FTS = float(obj["x"])
    y_FTS = float(obj["y"])
    phi_FTS = float(obj["phi"])

    alpha=phi_FTS-math.pi/2

    #Ziel bezogen auf FTS
    dx=-math.cos(alpha)*(x_soll-x_FTS)-math.sin(alpha)*(y_soll-y_FTS)
    dy=-math.sin(alpha)*(x_soll-x_FTS)+math.cos(alpha)*(y_soll-y_FTS)

    #Berechnung der Geschwindigkeit
    v_x=kp*dx
    v_y=kp*dy

    v_ges=math.sqrt(v_x**2+v_y**2)

    if (v_ges>255):
        v_ges=255
    else:
        v_ges

    #Berechnung der Fahrrichtung
    rad = math.atan2(dy,dx)

    #Berechnung der Winkeländerung und Winkelgeschwindigkeit omega
    d_phi=(phi_soll-phi_FTS)

    if (abs(d_phi)>math.pi):
        if (d_phi>0):
            d_phi=-2*math.pi+d_phi
        else:
            d_phi=2*math.pi+d_phi
    else:
        d_phi=d_phi
```

```
omega=d_phi*k_phi #[1/s]

#Übergabe der Daten über serielle Schnittstelle
cmd="{:.0f} {:.5f} {:.5f} {:.5f}\n".format(v_ges)+":{:.5f} {:.5f} {:.5f} {:.5f}\n".format(rad)+":{:.5f} {:.5f} {:.5f} {:.5f}\n".format(omega)
cmd=cmd+'\n'
print(str(cmd))
arduinoData.write(cmd.encode())

arduinoData=serial.Serial('COM5', 56700)
while True:
    Lageregelung(-200,0,0)
    time.sleep(0.5)
```

**Anhang F: Skript Bahnplanung für Punkt-zu-Punkt-Bewegung**

```
import requests
import serial
import time
import math
import json
import matplotlib.pyplot as plt

#=====
#=====Lageregelung=====
#=====

def Lageregelung(x_soll,y_soll,phi_soll):
    kp=1.2
    k_phi=0.7

    print("x_soll: "+str(x_soll))
    print("y_soll: "+str(y_soll))
    print("phi_soll: "+str(phi_soll))

    r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
    obj = json.loads(r.text)
    x_FTS = float(obj["x"])
    y_FTS = float(obj["y"])
    phi_FTS = float(obj["phi"])

    print("Koordinaten FTS: " + str(x_FTS), str(y_FTS), str(phi_FTS))

    alpha=phi_FTS-math.pi/2 #*180/math.pi
    print("Alpha: "+str(alpha))

    #Ziel bezogen auf FTS
    dx=-math.cos(alpha)*(x_soll-x_FTS)-math.sin(alpha)*(y_soll-y_FTS)
    dy=-math.sin(alpha)*(x_soll-x_FTS)+math.cos(alpha)*(y_soll-y_FTS)

    dist = math.sqrt(dx**2 + dy**2)
    print("Distanz: "+str(dist))

    #Berechnung der Geschwindigkeit
    v_x=kp*dx
    v_y=kp*dy
```

```
#Ziel bezogen auf FTS
d_phi=(phi_soll-phi_FTS)

if (abs(d_phi)>math.pi):
    if (d_phi>0):
        d_phi=-2*math.pi+d_phi
    else:
        d_phi=2*math.pi+d_phi

print("Rotationswinkel: "+str(d_phi))
d_omega=k_phi*d_phi # [1/s]

print("Kommando Winkel: "+str(d_omega))
print("  ")

return v_x,v_y,d_omega

#=====
#=====Funktion für Bahnplanung=====
#=====

def Interpolation(t,t_a,t_b,a,v):

    if v<0:
        a=-a
    else:
        a

    if(t <=t_a):
        x=0.5*a*(t)**2
        v_neu=a*t

    if(t_a<t<=t_b):
        x=v*t-0.5*v**2/a
        v_neu=v

    if(t>t_b):
        x=0.5*a*(v/a)**2+v*(t_b-t_a)+v*(t-t_b)-0.5*a*(t-t_b)**2
        v_neu=v-a*(t-t_b)

    return [x,v_neu]
```

```

#=====
#=====Bahnplanung=====
#=====

def Bahnplanung(x_ziel, y_ziel, phi_ziel):
    r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
    obj = json.loads(r.text)
    x_start = float(obj["x"])
    y_start= float(obj["y"])
    phi_start= float(obj["phi"])

    print("Koordinaten FTS: " + str(x_start), str(y_start),
          str(phi_start))

    s_x_gesamt=x_ziel-x_start
    s_y_gesamt=y_ziel-y_start
    s_gesamt=math.sqrt(s_x_gesamt**2+s_y_gesamt**2)
    phi_gesamt=phi_ziel-phi_start

    if (abs(phi_gesamt)>math.pi):
        if (phi_gesamt>0):
            phi_gesamt=-2*math.pi+phi_gesamt
        else:
            phi_gesamt=2*math.pi+phi_gesamt
    else:
        phi_gesamt

    print("Winkel_neu: "+str(phi_gesamt))

    '''

    v
    |
    |
    | _____
    | /| | \ \
    | / | | \ \
    |/__| _____|_\ \____t
    t_a      t_b
    '''

v_trans=200 #mm/s
a_trans=a_x=a_y=100 #mm/s^2
teta=2.5 #rad/s^2
omega=3 #rad/s

#t_ges_trans=s_gesamt/v_trans+v_trans/a_trans
t_ges_x=abs(s_x_gesamt)/v_trans+v_trans/a_trans
t_ges_y=abs(s_y_gesamt)/v_trans+v_trans/a_trans
t_ges_phi=abs(phi_gesamt)/omega +omega/teta

```

```

#für synchrone Bewegung müssen alle Positionen gleichzeitig erreicht
werden
#bestimmen, welche Komponente am längsten braucht und diese als neue
Zeit festlegen
t_ges=max(t_ges_phi,t_ges_x,t_ges_y)

print("x Zeit: "+str(t_ges_x))
print("y Zeit: "+str(t_ges_y))
print("phi Zeit: "+str(t_ges_phi))
print("Neue Zeit: "+str(t_ges))

#Berechnen der neuen Geschwindigkeiten bezogen auf die neue Zeit
v_x_neu=((a_x*t_ges)/2-math.sqrt((a_x**2*t_ges**2)/4-
abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt

v_y_neu=((a_y*t_ges)/2-math.sqrt((a_y**2*t_ges**2/4)-
abs(s_y_gesamt)*a_y))*abs(s_y_gesamt)/s_y_gesamt

omega_neu=((teta*t_ges)/2-math.sqrt((teta**2*t_ges**2/4)-
abs(phi_gesamt)*teta))*abs(phi_gesamt)/phi_gesamt

#Beschleunigungszeiten
t_ax=abs(v_x_neu)/a_x
t_ay=abs(v_y_neu)/a_y
t_a_phi=abs(omega_neu)/teta

#Zeit ab der abgebremst werden soll
t_bx=t_ges-t_ax
t_by=t_ges-t_ay
t_b_phi=t_ges-t_a_phi

print("Zeit zum Abbremsen in x: "+str(t_bx))
print("Zeit zum Abbremsen in y: "+str(t_by))
print("Zeit zum Abbremsen in phi: "+str(t_b_phi))

#leere Listen, in welche die Koordinaten der interpolierten Punkte
und die Sollgeschwindigkeiten geschrieben werden
x_soll=[]
vx_soll=[]

y_soll=[]
vy_soll=[]

phi_soll=[]
omega_soll=[]

t_=[]

```

```
delta_t=0.1 #Intervallzeit
t=0 #Startzeitpunkt

n=int(round((t_ges/delta_t),0))
delta_t=t_ges/n

print("delta t: "+str(delta_t))

z=n+1

for w in range(z):
    [x_t, vx_soll_neu]=Interpolation(t,t_ax,t_bx,a_x,v_x_neu)
    x_soll_neu=x_t+x_start
    x_soll.append(x_soll_neu)
    vx_soll.append(vx_soll_neu)

    [y_t, vy_soll_neu]=Interpolation(t,t_ay,t_by,a_y,v_y_neu)
    y_soll_neu=y_t+y_start
    y_soll.append(y_soll_neu)
    vy_soll.append(vy_soll_neu)

    [phi_t, omega_soll_neu]= Interpolation(t,t_a_phi,t_b_phi,teta,omega_neu)
    phi_soll_neu=phi_t+phi_start
    phi_soll.append(phi_soll_neu)
    omega_soll.append(omega_soll_neu)

    t_.append(t)
    t+=delta_t

print("x_soll: "+str(x_soll))
print("vx_soll: "+str(vx_soll))
print("vy_soll: "+str(vy_soll))
print("y_soll: "+str(y_soll))
print("phi_soll: "+str(phi_soll))

plt.plot(t_,x_soll, color='red')
plt.plot(t_,y_soll, color='green')
plt.plot(t_,phi_soll, color='blue')
plt.show()

plt.plot(t_,vx_soll, color='red')
plt.plot(t_,vy_soll, color='green')
plt.plot(t_,omega_soll, color='blue')
plt.show()

plt.plot(x_soll, y_soll)
plt.show()

k=0
while True:
```

```

if (k<(n)):

    d_v=Lageregelung(x_soll[k],y_soll[k],phi_soll[k])
    print("Werte aus Lageregelung: " +str(d_v))
    d_v_x=d_v[0]
    d_v_y=d_v[1]
    d_omega=d_v[2]

    v_x_korr=vx_soll[k]+d_v_x
    v_y_korr=vy_soll[k]+d_v_y
    omega_korr=omega_soll[k]+d_omega

    v_ges_korr=math.sqrt(v_x_korr**2+v_y_korr**2)

    if (v_ges_korr>255):
        v_ges_korr=255
    else:
        v_ges_korr

    rad = math.atan2(v_y_korr,v_x_korr)
    k=k+1
else:
    d_v=Lageregelung(x_ziel,y_ziel,phi_ziel)
    print("Werte aus Lageregelung: " +str(d_v))
    d_v_x=d_v[0]
    d_v_y=d_v[1]
    omega_korr=d_v[2]
    rad = math.atan2(d_v_y,d_v_x)
    v_ges_korr=math.sqrt(d_v_x**2+d_v_y**2)

    print("Geschwindigkeit: "+str(v_ges_korr))
    print("Fahrrichtung: "+str(rad))
    print("Drehgeschwindigkeit: "+str(omega_korr))

#Übergabe der Daten über serielle Schnittstelle
cmd="{:0f}".format(v_ges_korr)+":{:5f}".format(rad)+":{:5f}".format(omega_korr)
cmd=cmd+'\n'
print(str(cmd))
arduinoData.write(cmd.encode())

time.sleep(delta_t)

=====
=====Aufruf von Programm=====
=====

arduinoData=serial.Serial('COM5',57600)
Bahnplanung(-200,300,2)

```

**Anhang G: Skript Pfadplanung und Fahrbewegung entlang Pfad**

```
import requests
import serial
import time
import math
import json
import numpy as np

#=====
#=====Lageregelung=====
#=====

def Lageregelung(x_soll,y_soll,phi_soll):
    kp=1.2
    k_phi=0.7

    print("x_soll: "+str(x_soll))
    print("y_soll: "+str(y_soll))
    print("phi_soll: "+str(phi_soll))

    r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
    obj = json.loads(r.text)
    x_FTS = float(obj["x"])
    y_FTS = float(obj["y"])
    phi_FTS = float(obj["phi"])
    print("Koordinaten FTS: " + str(x_FTS), str(y_FTS), str(phi_FTS))

    alpha=phi_FTS-math.pi/2
    print("Alpha: "+str(alpha))

#Ziel bezogen auf FTS
dx=-math.cos(alpha)*(x_soll-x_FTS)-math.sin(alpha)*(y_soll-y_FTS)
dy=-math.sin(alpha)*(x_soll-x_FTS)+math.cos(alpha)*(y_soll-y_FTS)

dist = math.sqrt(dx**2 + dy**2)
print("Distanz: "+str(dist))

#Berechnung der Geschwindigkeit
v_x=kp*dx
v_y=kp*dy

d_phi=(phi_soll-phi_FTS)

if (abs(d_phi)>math.pi):
    if (d_phi>0):
        d_phi=-2*math.pi+d_phi
    else:
        d_phi=2*math.pi+d_phi
```

```
print("Rotationswinkel: "+str(d_phi))
d_omega=d_phi*k_phi # [1/s]

print("Kommando Winkel: "+str(d_omega))
print("  ")

return v_x,v_y,d_omega

#=====
#=====Funktion für Bahnplanung=====
#=====

def Interpolation1(t,t_a,t_b,a,v):
    if v<0:
        a=-a
    else:
        a

    if(t <=t_a):
        x=0.5*a*(t)**2
        v_neu=a*t

    if(t_a<t<=t_b):
        x=v*t-0.5*v**2/a
        v_neu=v

    if(t>t_b):
        x=0.5*a*(v/a)**2+v*(t_b-t_a)+v*(t-t_b)-0.5*a*(t-t_b)**2
        v_neu=v-a*(t-t_b)

    return [x,v_neu]

def Interpolation2(t,t_a,a,v):
    if v<0:
        a=-a
    else:
        a

    if(t <=t_a):
        x=0.5*a*(t)*(t)
        v_neu=a*t

    if(t_a<t):
        x=v*t-0.5*v**2/a
        v_neu=v

    return [x,v_neu]
```

```

def Interpolation3(t,t_b,a,v):
    if v<0:
        a=-a
    else:
        a

    if (t<=t_b):
        x=v*t
        v_neu=v

    if (t>t_b):
        x=t_b*v+v*(t-t_b)-0.5*(t-t_b)**2*a
        v_neu=v-a*(t-t_b)

    return [x,v_neu]

def Interpolation4(t,v):
    x=v*t
    v_neu=v

    return [x,v_neu]
#=====
#=====Bahnplanung=====
#=====

def Bahnplanung(x_ziel, y_ziel, phi_ziel,v_alt,omega_alt):
    r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
    obj = json.loads(r.text)
    x_start = float(obj["x"])
    y_start= float(obj["y"])
    phi_start= float(obj["phi"])

    print("Koordinaten FTS: " + str(x_start), str(y_start), str(phi_start))

    s_x_gesamt=x_ziel-x_start
    s_y_gesamt=y_ziel-y_start
    s_gesamt=math.sqrt(s_x_gesamt**2+s_y_gesamt**2)
    phi_gesamt=phi_ziel-phi_start

    print("Strecke in x-Richtung: "+str(s_x_gesamt))
    print("Strecke in y-Richtung: "+str(s_y_gesamt))
    print("Strecke in translatorisch: "+str(s_gesamt))
    print("Winkel: "+str(phi_gesamt))

    if (abs(phi_gesamt)>math.pi):
        if (phi_gesamt>0):
            phi_gesamt=-2*math.pi+phi_gesamt
        else:
            phi_gesamt=2*math.pi+phi_gesamt

```

```

print("Winkel_neu: "+str(phi_gesamt))

v_trans=v_alt #mm/s
a_trans=a_x=a_y=150 #mm/s^2
teta=2.5 #rad/s^2
omega=omega_alt #rad/s

#leere Listen, in welche die Koordinaten der interpolierten Punkte
und die Sollgeschwindigkeiten geschrieben werden
x_soll=[]
vx_soll=[]

y_soll=[]
vy_soll=[]

phi_soll=[]
omega_soll=[]

delta_t=0.1 #Intervallzeit
t=0 #Startzeitpunkt

if len(Pfad)==2:
    print("In erster if-Schleife")

    """
    v
    |
    |
    | /| _____ | \
    | / | _____ | \
    |/____|_____|____\____ t
        t_a           t_b
    """

    #t_ges_trans=s_gesamt/v_trans+v_trans/a_trans
    t_ges_x=abs(s_x_gesamt)/v_trans+v_trans/a_trans
    t_ges_y=abs(s_y_gesamt)/v_trans+v_trans/a_trans
    t_ges_phi=abs(phi_gesamt)/omega +omega/teta

    #für synchrone Bewegung müssen alle Positionen gleichzeitig
    erreicht werden
    #bestimmen, welche Komponente am längsten braucht und diese als
    neue Zeit festlegen
    t_ges=max(t_ges_phi,t_ges_x,t_ges_y)

    print("Neue Zeit: "+str(t_ges))

```

```

#Berechnen der neue Geschwindigkeiten bezogen auf die neue Zeit
v_x_neu=((a_x*t_ges)/2-math.sqrt(((a_x**2)*(t_ges**2)/4)-
abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt

v_y_neu=((a_y*t_ges)/2-math.sqrt(((a_y**2)*(t_ges**2)/4)-
abs(s_y_gesamt)*a_y))*abs(s_y_gesamt)/s_y_gesamt

omega_neu=((teta*t_ges)/2-math.sqrt(((teta**2)*(t_ges**2)/4)-
abs(phi_gesamt)*teta))*abs(phi_gesamt)/phi_gesamt

#Beschleunigungszeiten
t_ax=abs(v_x_neu)/a_x
t_ay=abs(v_y_neu)/a_y
t_a_phi=abs(omega_neu)/teta

#Zeit ab der abgebremst werden soll
t_bx=t_ges-t_ax
t_by=t_ges-t_ay
t_b_phi=t_ges-t_a_phi

n=int(round((t_ges/delta_t),0))
delta_t=t_ges/n

z=n+1

for w in range(z):

    [x_t, vx_soll_neu]=Interpolation1(t,t_ax,t_bx,a_x,v_x_neu)
    x_soll_neu=x_t+x_start
    x_soll.append(x_soll_neu)
    vx_soll.append(vx_soll_neu)

    [y_t, vy_soll_neu]=Interpolation1(t,t_ay,t_by,a_y,v_y_neu)
    y_soll_neu=y_t+y_start
    y_soll.append(y_soll_neu)
    vy_soll.append(vy_soll_neu)

    [phi_t, omega_soll_neu]=Interpolation1(t,t_a_phi,t_b_phi,teta,omega_neu)
    phi_soll_neu=phi_t+phi_start
    phi_soll.append(phi_soll_neu)
    omega_soll.append(omega_soll_neu)

    t+=delta_t

```

```

if len(Pfad)>2 and p==1:
    print("In zweiter Schleife")
    '''
    v
    |
    |
    | /| _____ |
    | / |           |
    |/__|_____ |____ t
p0   t_a          p1
'''

t_ges_trans=s_gesamt/v_trans+0.5*v_trans/a_trans
t_ges_phi=abs(phi_gesamt)/omega +0.5*omega/teta

#für synchrone Bewegung müssen alle Positionen gleichzeitig
erreicht werden
#bestimmen, welche Komponente am längsten braucht und diese als
neue Zeit festlegen
t_ges=max(t_ges_phi,t_ges_trans)

print("Neue Zeit: "+str(t_ges))

#Berechnen der neuen Geschwindigkeiten bezogen auf die neue Zeit
v_x_neu=((a_x*t_ges)-math.sqrt(((a_x**2)*(t_ges**2))-2*abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt

v_y_neu=((a_y*t_ges)-math.sqrt(((a_y**2)*(t_ges**2))-2*abs(s_y_gesamt)*a_y))*abs(s_y_gesamt)/s_y_gesamt

omega_neu=((teta*t_ges)-math.sqrt(((teta**2)*(t_ges**2))-2*abs(phi_gesamt)*teta))*abs(phi_gesamt)/phi_gesamt

#Beschleunigungszeiten
t_ax=abs(v_x_neu)/a_x
t_ay=abs(v_y_neu)/a_y
t_a_phi=abs(omega_neu)/teta

n=int(round((t_ges/delta_t),0))
delta_t=t_ges/n

z=n+1

for w in range(z):
    [x_t, vx_soll_neu]=Interpolation2(t,t_ax,a_x,v_x_neu)
    x_soll_neu=x_t+x_start
    x_soll.append(x_soll_neu)
    vx_soll.append(vx_soll_neu)

```

```

[y_t, vy_soll_neu]=Interpolation2(t,t_ay,a_y,v_y_neu)
y_soll_neu=y_t+y_start
y_soll.append(y_soll_neu)
vy_soll.append(vy_soll_neu)

[phi_t, omega_soll_neu]=Interpolation2(t,t_a_phi,teta,omega_neu)
phi_soll_neu=phi_t+phi_start
phi_soll.append(phi_soll_neu) #/math.pi*180)
omega_soll.append(omega_soll_neu)

t+=delta_t

if len(Pfad)>2 and p==(len(Pfad)-1):
    print("In dritter if-Schleife")

'''  

v  

|  

|  

| _____ | \  

|       | \ \  

|       | \ \ t  

p0      t_b   p1  

'''  

t_ges_trans=s_gesamt/v_trans+0.5*v_trans/a_trans
t_ges_phi=abs(phi_gesamt)/omega +0.5*omega/teta

#für synchrone Bewegung müssen alle Positionen gleichzeitig
erreicht werden
#bestimmen, welche Komponente am längsten braucht und diese als
neue Zeit festlegen
t_ges=max(t_ges_phi,t_ges_trans)

print("Neue Zeit: "+str(t_ges))

#Berechnen der neuen Geschwindigkeiten bezogen auf die neue Zeit
v_x_neu=((a_x*t_ges)-math.sqrt(((a_x**2)*(t_ges**2))-  

2*abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt

v_y_neu=((a_y*t_ges)-math.sqrt(((a_y**2)*(t_ges**2))-  

2*abs(s_y_gesamt)*a_y))*abs(s_y_gesamt)/s_y_gesamt

omega_neu=((teta*t_ges)-math.sqrt(((teta**2)*(t_ges**2))-  

2*abs(phi_gesamt)*teta))*abs(phi_gesamt)/phi_gesamt

#Beschleunigungszeiten
t_ax=abs(v_x_neu)/a_x
t_ay=abs(v_y_neu)/a_y
t_a_phi=abs(omega_neu)/teta

```

```

#Zeit ab der abgebremst werden soll
t_bx=t_ges-t_ax
t_by=t_ges-t_ay
t_b_phi=t_ges-t_a_phi

n=int(round((t_ges/delta_t),0))
delta_t=t_ges/n

z=n+1

for w in range(z):
    [x_t, vx_soll_neu]=Interpolation3(t,t_bx,a_x,v_x_neu)
    x_soll_neu=x_t+x_start
    x_soll.append(x_soll_neu)
    vx_soll.append(vx_soll_neu)

    [y_t, vy_soll_neu]=Interpolation3(t,t_by,a_y,v_y_neu)
    y_soll_neu=y_t+y_start
    y_soll.append(y_soll_neu)
    vy_soll.append(vy_soll_neu)

    [phi_t, omega_soll_neu]=Interpolation3(t,t_b_phi,teta,omega_neu)
    phi_soll_neu=phi_t+phi_start
    phi_soll.append(phi_soll_neu)
    omega_soll.append(omega_soll_neu)

    t+=delta_t

if len(Pfad)>2 and p<(len(Pfad)-1) and p>1:
    print("In vierter if-Schleife")

'''

v
|
|_____
|       |
|_____|_____| t
p0      p1
'''


t_ges_trans=s_gesamt/v_trans
t_ges_phi=abs(phi_gesamt)/omega

#für synchrone Bewegung müssen alle Positionen gleichzeitig
erreicht werden
#bestimmen, welche Komponente am längsten braucht und diese als
neue Zeit festlegen
t_ges=max(t_ges_phi,t_ges_trans)

```

```
print("Neue Zeit: "+str(t_ges))

#Berechnen der neue Geschwindigkeiten bezogen auf die neue Zeit
v_x_neu=s_x_gesamt/t_ges
v_y_neu=s_y_gesamt/t_ges
omega_neu=phi_gesamt/t_ges

n=int(round((t_ges/delta_t),0))
delta_t=t_ges/n

z=n+1

for w in range(z):
    [x_t, vx_soll_neu]=Interpolation4(t,v_x_neu)
    x_soll_neu=x_t+x_start
    x_soll.append(x_soll_neu)
    vx_soll.append(vx_soll_neu)

    [y_t, vy_soll_neu]=Interpolation4(t,v_y_neu)
    y_soll_neu=y_t+y_start
    y_soll.append(y_soll_neu)
    vy_soll.append(vy_soll_neu)

    [phi_t, omega_soll_neu]=Interpolation4(t,omega_neu)
    phi_soll_neu=phi_t+phi_start
    phi_soll.append(phi_soll_neu)
    omega_soll.append(omega_soll_neu)

    t+=delta_t

for k in range(n):
    d_v=Lageregelung(x_soll[k],y_soll[k],phi_soll[k])
    print("Werte aus Lageregelung: " +str(d_v))
    d_v_x=d_v[0]
    d_v_y=d_v[1]
    d_omega=d_v[2]

    v_x_korr=vx_soll[k]+d_v_x
    v_y_korr=vy_soll[k]+d_v_y
    omega_korr=omega_soll[k]+d_omega

    v_ges_korr=math.sqrt(v_x_korr**2+v_y_korr**2)

    if (v_ges_korr>255):
        v_ges_korr=255
    else:
        v_ges_korr

    rad = math.atan2(v_y_korr,v_x_korr)
```

```
k=k+1

print("Geschwindigkeit: "+str(v_ges_korr))
print("Fahrrichtung: "+str(rad))
print("Drehgeschwindigkeit: "+str(omega_korr))

#Übergabe der Daten über serielle Schnittstelle
cmd=" {:.0f} {:.5f} {:.5f} ".format(v_ges_korr,rad,omega_korr)
cmd=cmd+'\n'
print(str(cmd))
arduinoData.write(cmd.encode())

time.sleep(delta_t)

return [v_ges_korr,omega_korr]

#=====
#=====Pfadplanung=====
#=====
arduinoData=serial.Serial('COM5', 57600)

#####Funktionen für Pfadplanungsalgorithmus#####
def berechne_distanz(p1,p2):
    dist=math.sqrt((p2[0]-p1[0])**2+(p2[1]-p1[1])**2)

    return dist

def inHindernis(p_check):
    in_Hindernis=[]
    for a in range (H):
        distanz=berechne_distanz(Hindernisse[a][0],p_check)

        if distanz <= Hindernisse[a][1]:
            in_Hindernis.append(True)
        else:
            in_Hindernis.append(False)
    return in_Hindernis

def inArbeitsraum(p1):
    print ("Funktion AR")
    AR_OK=True
    print("p_x: "+ str(p1[0]))
    print("p_y: "+ str(p1[1]))
    if AR_x[1] <p1[0] or p1[0] < AR_x[0]:
        AR_OK=False
    if AR_y[1] <p1[1] or p1[1] < AR_y[0]:
        AR_OK=False

    return AR_OK
```

```
def Punkt_zu_Linie(p1,p2,p3):
    Vektor=(p2[0]-p1[0],p2[1]-p1[1])
    VektorGröße=math.sqrt(Vektor[0]**2+Vektor[1]**2)

    u=((p3[0]-p1[0])*(p2[0]-p1[0])+(p3[1]-p1[1])*(p2[1]-
p1[1]))/(VektorGröße**2)

    #Koordinaten des Punktes, an dem sich der Vektor von p1 zu p2 mit
    #der Orthogonalen zu p3 schneidet
    px=p1[0]+u*(p2[0]-p1[0])
    py=p1[1]+u*(p2[1]-p1[1])

    distanz=berechne_distanz((px,py),p3)
    print("u,distanz: "+str(u)+", "+str(distanz))

    return u, distanz

def Kollision_Pfad_Hindernis(p1,p2):
    LinienKollision=[]
    for s in range(H):
        zu_nah=False
        dazwischen=False

        u, dist =Punkt_zu_Linie(p1,p2,Hindernisse[s][0])

        if 0<=u <=1:
            dazwischen=True

        if dist <= Hindernisse[s][1]:
            zu_nah=True

        if zu_nah and dazwischen:
            LinienKollision.append(True)
        else:
            LinienKollision.append(False)

    return LinienKollision
```

```
#####Pfadplanung#####
AR_x=(-450,450)
AR_y=(-320,320)

#Hindernisse erzeugen
H=int(input('Bitte gib Anzahl an Hindernissen ein: '))
Hindernisse=[]
R_FTS=550/2
for i in range (H):
    x_Hindernis=float(input("x_Koordinate des Hindernis " +str(i+1)+":"))
    y_Hindernis=float(input('y_Koordinate des Hindernis '+str(i+1)+': '))
    MP_Hindernis=(x_Hindernis,y_Hindernis)
    R_Hindernis=float(input("Radius des Hindernis "+str(i+1)+": ")) +R_FTS
    HindernisPara=[MP_Hindernis,R_Hindernis]
    Hindernisse.append(HindernisPara)
    print("Hinderniss: "+str(Hindernisse))

#Startpunkt einlesen
r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
obj = json.loads(r.text)
x_start = float(obj["x"])
y_start= float(obj["y"])
phi_start= float(obj["phi"])

Start=(x_start,y_start)
print("Start: "+str(Start))
Start_Name='k0'
Punkte=[[Start_Name,Start,'None']]

#Endpunkt angeben
Ziel_OK=False
while not Ziel_OK:
    x_Ziel=float(input("x-Koordinate des Zielpunktes: "))
    y_Ziel=float(input("y-Koordinate des Zielpunktes: "))
    phi_Ziel=float(input("Orientierung des Zielpunktes: "))
    Ziel=(x_Ziel,y_Ziel)

    Kollisionskontrolle=inHindernis(Ziel)
    print("Kollision: "+str(Kollisionskontrolle))
    AR_Kontrolle=inArbeitsraum(Ziel)
    print("AR_Kontrolle: " +str(AR_Kontrolle))

    if any (Kollisionskontrolle) or (AR_Kontrolle !=True):
        print("ERROR: Ziel liegt nicht im Konfigurationsraum!")
        pass
    else:
        Ziel_OK=True
print("Ziel_OK: "+str(Ziel_OK))
```

```
Weg_zum_Ziel=False

#Prüfen ob direkter Weg zu Ziel möglich
if any (Kollision_Pfad_Hindernis(Start,Ziel)):
    print("Kein direkter Pfad zu Ziel möglich")

else:
    print("Direkter Pfad zu Ziel gefunden!")
    Weg_zum_Ziel=True
print("Weg zum Ziel: "+str(Weg_zum_Ziel))

#Pfad finden

while not Weg_zum_Ziel:
    #Erstellen von neuen Punkten
    Punkt_OK=False
    while not Punkt_OK:
        Punkt_Name = "k{}".format(len(Punkte))
        x_Punkt=np.random.randint(AR_x[0],AR_x[1])
        y_Punkt=np.random.randint(AR_y[0],AR_y[1])
        print("Beliebiger Punkt: "+str(x_Punkt)+", "+str(y_Punkt))

        #Bestimmen welcher der dichteste Punkt zu beliebigen Punkt ist,
        #Distanz berechnen, Index bestimmen
        minDist=1e10 #nur als Anfangsdistanz
        for t in range(len(Punkte)):
            dist_zu_Punkt=berechne_distanz((Punkte[t][1]),(x_Punkt,y_Punkt))
            if dist_zu_Punkt<=minDist:
                minDist=dist_zu_Punkt
                ParentPunkt=t

        Parent="k{}".format(ParentPunkt)
        Koor_Parent=(Punkte[ParentPunkt][1])

        #Vektor von Parent zu beliebigem Punkt
        dx=x_Punkt-Punkte[ParentPunkt][1][0]
        dy=y_Punkt-Punkte[ParentPunkt][1][1]
        KürzesterVektor=[dx,dy]
        LängeVektor=math.sqrt((dx**2)+(dy**2))
        EinheitsVektor=[(dx/LängeVektor),(dy/LängeVektor)]

        #Neuen Punkt bestimmten
        Abs=150 #Abstand, in dem der neue Punkt gesetzt werden soll
        print("ParentPunkt_x: "+str(Punkte[ParentPunkt][1][0]))
        x_neu=Punkte[ParentPunkt][1][0]+Abs*EinheitsVektor[0]
        y_neu=Punkte[ParentPunkt][1][1]+Abs*EinheitsVektor[1]
        print("Neuer Punkt: "+str(x_neu)+", "+str(y_neu))
        Punkt_neu=(x_neu,y_neu)
```

```
Kollision = inHindernis(Punkt_neu)
ARcheck=inArbeitsraum(Punkt_neu)
print("ARcheck: "+str(ARcheck))
print("Kollision: "+str(Kollision))

if any(Kollision) or (ARcheck ==False):
    pass
else:
    Punkt_OK=True
print("Punkt_OK: "+str(Punkt_OK))

#neuer Punkt zu Liste hinzufügen
Punkte.append([Punkt_Name, Punkt_neu, Parent])
print("Punkte: "+str(Punkte))

#schauen, ob es vom letzten Punkt der Liste einen hindernisfreien
#Pfad zum Ziel gibt
LinienCheck=Kollision_Pfad_Hindernis(Punkte[-1][1],Ziel)

if any(LinienCheck):
    pass
else:
    Weg_zum_Ziel=True
print("Weg zum Ziel: "+str(Weg_zum_Ziel))

#Weg zurück vom Ziel zum Start
Pfad=[Punkte[-1][1]]
am_Start=False

aktueller_Punkt=Punkte[-1][1]
aktueller_Parent=Punkte[-1][2]

while not am_Start:
    for l in range(len(Punkte)):
        if Punkte[l][0]==aktueller_Parent:
            Pfad.insert(0,Punkte[l][1])
            aktueller_Punkt=Punkte[l][1]
            aktueller_Parent=Punkte[l][2]
        if aktueller_Parent=='None':
            am_Start=True

Pfad.append(Ziel)

print ("Pfad: "+str(Pfad))
print ("Anzahl an Punkten in Pfad: "+str(len(Pfad)))
```

```

#Abfahren des Pfades
delta_phi=(phi_Ziel-phi_start)/(len(Pfad)-1)
phi_Pfad=[]
for z in range(len(Pfad)):
    phi=phi_start+z*delta_phi
    phi_Pfad.append(phi)
print("Phi entlang Pfad: "+str(phi_Pfad))

p=1
print("Länge Pfad: "+str(len(Pfad)))
v=[200]
omega=[3]

while p < len(Pfad):
    [v_neu,omega_neu]=Bahnplanung(Pfad[p][0],Pfad[p][1],phi_Pfad[p],v[p-1],omega[p-1])
    v.append(v_neu)
    omega.append(omega_neu)
    print("Am Punkt: "+str(p))
    p=p+1

print("Am Ziel!!!")

while True:
    d_v=Lageregelung(Pfad[p-1][0],Pfad[p-1][1],phi_Pfad[p-1])
    print("Werte aus Lageregelung: " +str(d_v))
    d_v_x=d_v[0]
    d_v_y=d_v[1]
    omega_korr=d_v[2]
    rad = math.atan2(d_v_y,d_v_x)
    v_ges_korr=math.sqrt(d_v_x**2+d_v_y**2)

    print("Nur Lageregelung")
    print("Geschwindigkeit: "+str(v_ges_korr))
    print("Fahrrichtung: "+str(rad))
    print("Drehgeschwindigkeit: "+str(omega_korr))

#Übergabe der Daten über serielle Schnittstelle
cmd="{:.0f}".format(v_ges_korr)+":{:.5f}".format(rad)+":{:.5f}".format(omega_korr)
cmd=cmd+'\n'
print(str(cmd))
arduinoData.write(cmd.encode())

time.sleep(0.1)

```