Simulation von autonomen Einparken mit dem Kleinroboter von Lego

Studienarbeit

5. und 6. Semester

des Studiengangs Angewandte Informatik
an der dualen Hochschule Baden-Württemberg in Karlsruhe
von

Simon Franke
18.04.2016

Bearbeitungszeitraum:	TBD
Matrikelnummer:	5484393
Kurs:	TINF13B2
Ausbildungsfirma:	FILIADATA GmbH
Betreuer:	Prof. H.-J. Haubner

Erklärung

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

---------- ----------

Ort *Datum* *Unterschrift*

Zusammenfassung:

Diese vorliegende Arbeit hat das Ziel, einen Algorithmus zu entwickeln, mit dem ein beliebiges Auto in eine passende Parklücke autonom einparken kann. Zu Beginn werden zunächst Grundlagen in theoretischer und technischer Hinsicht vermittelt. Anschließend wird mithilfe mathematischer Berechnungen ein Einparkweg ermittelt und simuliert. Zum Schluss wird der Algorithmus zum Einparken und die Ergebnisse des Testdurchlaufs vorgestellt.

Abstract:

This thesis aims to develop an algorithm that allows any car to park autonomously in a suitable parking space. At the beginning basics are first taught from a theoretical and technical viewpoint. With help of mathematics an ideal parking path is then identified and simulated. Finally, the algorithm for parking and the results of the test run will be presented.

I. Inhalt

II. Tabellenverzeichnis

III. Abbildungsverzeichnis

1. Einführung

In der heutigen Zeit wird das Autofahren immer mehr von technischen Hilfsmitteln unterstützt. Diese sogenannten Fahrassistenzsysteme greifen automatisch oder nach Aktivierung auf Antrieb, Steuerung oder Signaleinrichtungen ein, um entweder die Sicherheit oder den Komfort für Insassen und Straßenteilnehmer zu erhöhen. Dabei bewerten die Systeme die Umgebung und Motor- und Fahrwerkszustand durch verschiedene Arten von Sensoren. [1]

Volles autonomes Fahren ist eine Zukunftsvision vieler namhafter Automobilhersteller, jedoch ist es bis heute lediglich das, eine Vision [2]. Sowohl rechtliche, als auch soziale oder infrastrukturelle Einflüsse verhindern eine Marktdurchdringung des autonomen Fahrens. [3] Eine autonome Handlung des Autos jedoch bereits akzeptiert und implementiert, nämlich das Einparken. Durch sogenannte Einparkhilfen oder Einparkassistenten kann der Fahrer entweder unterstützt werden, oder er kann die Lenkung und Parklückenfindung ganz dem Fahrzeug überlassen. Aber auch hier ist es noch so, dass der Fahrer selbst Gas geben und Bremsen muss. [4]

Im Rahmen einer Studienarbeit an der DHBW Karlsruhe soll ein Algorithmus erarbeitet werden, anhand diesem ein Fahrzeug vollständig autonom Einparken kann. Dieser Algorithmus soll mithilfe eines Roboters, gebaut mit dem Lego Mindstorms System, realisiert und getestet werden.

2. Grundlagen

In diesem Kapitel wird Grundlagenwissen vermittelt, das benötigt wird, um die Inhalte der gesamten Arbeit verstehen zu können. Zunächst wird auf theoretische Grundlagen eingegangen zum Thema autonomes Einparken. Anschließend werden technische Hintergrundinformationen vermittelt, zu der verwendeten Hard- und Software.

2.1 Theoretische Grundlagen

In den Theoretischen Grundlagen geht es vor allem um Auto-spezifische Themen, wie die Einparkhilfe und wie es um den Stand der Technik und die Gesetzeslage bestellt ist.

2.1.1 Einparkhilfe

Die Einparkhilfe ist heute eines der bekanntesten der technischen Hilfsmittel. Bereits seit 1950 werden Automobile mit Parkunterstützungen versehen. Waren es zu Beginn noch passive Systeme, wie Peilstangen, die an sensiblen Teilen des Autos angebracht waren, um den Autofahrer vor einem drohenden Unfall zu warnen, sind heute eher die aktiven Systeme verbreitet. Hier gibt es zwei Ausprägungen, die rein akustischen oder die visuellen und akustischen Systeme. Im ersten Fall wird der Fahrer über piepende Geräusche über den Abstand zu einem anderen Gegenstand hingewiesen. Im zweiten Fall wird zusätzlich zu den Geräuschen auch noch in einem Display der Abstand zu einem Hindernis dargestellt. [5]

Die höchste Ausbaustufe der Einparkhilfe sind die selbst lenkenden Systeme, bei denen der Fahrer nur noch Gas geben, bremsen und den Gang wechseln muss. Der Lenkvorgang wird vom Auto übernommen. Voraussetzungen hierfür sind die aktiven Einparkhilfen. Die Parklücke wird von quer zur Fahrtrichtung ausgerichteten Messsensoren bewertet. Ebenfalls befinden sich an der vorderen und der hinteren Stoßstange diese Sensoren. Meist werden hier Ultraschallsensoren verwendet. Diese senden und empfangen Ultraschallsignale und übermitteln die gewonnenen Daten an das Steuergerät, welches nun aus der Ultraschallsignallaufzeit die Distanz vom Sensor zum Hindernis errechnet. Hier gilt, je höher die Anzahl der verbauten Sensoren, desto genauer ist das Messergebnis. Außerdem muss das

Fahrzeug über eine elektro-mechanische Servolenkung, die von einem Elektromotor angetrieben werden kann, verfügen. [5]

Möchte der Fahrer einparken, aktiviert er den Parklenkassistenten per Knopfdruck oder verringert seine Geschwindigkeit unter einen vorgegebenen Wert. Daraufhin wird quer zur Fahrtrichtung vermessen, ob eine Parklücke die passende Größe hat. Falls dies zutrifft, wird dem Fahrer dies signalisiert. Anschließend muss der Fahrer in einem passenden Abstand zur Parklücke anhalten, den Rückwärtsgang einlegen, leicht Gas geben und anschließend passend bremsen. Steht der Wagen in der Parklücke, kann der Assistent durch Einlegen des Vorwärtsganges beendet werden. [4]

Modernere Assistenten unterstützen ebenfalls das Rangieren in einer Parklücke, bei dem durch mehrmaliges korrigieren innerhalb der Parklücke eingeparkt wird. Dadurch kann eine kleinere Parklücke ausreichen. [6]

2.1.2 Autonomes Einparken

Wie in 2.1.1 beschrieben, muss der Fahrzeugführer selbst den Gang einlegen, Gas geben und bremsen. Nur der Lenkvorgang wird durch die Einparkhilfe übernommen. Dies hat rechtliche Hintergründe, da der Fahrer jederzeit die Kontrolle über sein Fahrzeug haben muss. Dies wird im „Wiener Übereinkommen über den Straßenverkehr“ aus dem Jahr 1968, zuletzt überarbeitet im Jahr 2014, im Artikel 8, Absatz 5 beschrieben, als:

„Jeder Führer muss dauernd sein Fahrzeug beherrschen oder seine Tiere führen können.“

Diese Definition würde jedoch auch einige bereits verwendete Assistenzsysteme ausschließen, wie den Tempomaten oder den Abstandsregler. Somit wurde in der Überarbeitung von 2014 vermerkt, dass Assistenzsysteme zulässig sind, solange sie vom Fahrer überwacht und jederzeit überstimmt oder abgeschaltet werden können. Die Verantwortung für das Fahrzeug liegt somit beim Fahrzeugführer. [7]

Des Weiteren fehlt auch bei den Fahrern die Bereitschaft die vollständige Kontrolle an ein computerbasiertes System abzugeben, auch weil einige der Sensoren noch die Zuverlässigkeit fehlt. [8]

Somit gibt es das voll-autonome Einparken heutzutage noch in keinem Serienfahrzeug. Viele bekannte Autohersteller forschen jedoch in diese Richtung und stellen dabei vor allem die Vorzüge dieser Art des Einparkens heraus. So beschreibt Mikael Thor, ein Mitarbeiter von Volvo:

„Und während das Auto sich im autonomen Betrieb selbst seinen Stellplatz sucht, sitzt man schon im Restaurant und nippt am Aperitif.“

Damit beschreibt er das Szenario, dass der Fahrzeugführer am Eingang eines Parkhauses das Auto abstellt und dieses dann voll selbstständig sich eine Parklücke sucht und sich dort abstellt. Ist der Fahrer dann mit seinem Restaurantbesuch fertig, zahlt er sein Parkticket, worauf das Fahrzeug zur Ausfahrt gefahren wird. [9]
Dies als Zukunftsaussicht klingt interessant, jedoch müssen in dieser Vision einige Voraussetzungen gelten. So muss nicht-autonomer Verkehr im Parkhaus ebenso ausgeschlossen werden, wie auch Fußgänger. Ebenso müsste das Parkhaus darauf vorbereitet werden, durch einen digitalen Lageplan und elektronische Stellplatzinformationen. BMW ist der Meinung, dass solch eine Technik eventuell bereits 2020 zum Einsatz kommen könnte. [9]

2.2 Technische Grundlagen

In den technischen Grundlagen wird auf wichtige Aspekte der Software und Hardware eingegangen.

2.2.1 Hardware in LEGO Mindstorms

Das LEGO Mindstorms System ist der Versuch des dänischen Spielwarenherstellers LEGO in den Bereich der schulischen Bildung vorzudringen. Das System soll eine spielerische Annäherung an die Themen Robotik und Programmieren ermöglichen und das bereits in jungen Jahren. Es dient dabei als verständliches Beispiel für ein sogenanntes "embedded system", in dem ein Mikrocontroller mit Sensoren und Aktoren kommuniziert. Durch die Möglichkeiten von LEGO sind damit sehr viele Ausprägungsformen eines embedded system denkbar. Das LEGO Mindstorms System wird im Rahmen der Fraunhofer-Initiative "Roberta – Lernen mit Robotern" zumeist verwendet. [10]
Das Kernstück ist der sogenannte "Brick", der einem Legostein nachempfunden wurde. Abbildung 1 zeigt den Brick in der aktuellen Version, EV3. [11]

Abbildung 1: Bild des Lego Bricks EV3

Der EV3 ist die sechste Generation des LEGO Mindstorms Systems. Bereits 1998 wurde das erste System mit dem Namen Robotics Invention System in der Version 1.0 vorgestellt. 1999 folgte die Version 1.5 und 2001 der letzte Teil der Robotics Invention System Serie mit der Version 2.0. Anfang 2006 wurde die vierte Generation des Mindstorms Systems vorgestellt und startete mit einem neuen Namen. Diese und die fünfte Version trugen den Namen Mindstorms NXT in der Version 1.0 und 2.0. 2013 ist die bisher letzte Ausprägung erschienen und es trägt den Namen Mindstorms EV3. [12]

Der EV3 bietet Anschlüsse für je vier Sensoren und Aktoren, einen USB-Port und einen Steckplatz für eine microSD Karte. Über den USB-Port kann ein WLAN Stick eingebunden werden. Somit verfügt der EV3 auch eine Schnittstelle zum W-LAN, außerdem ist auch eine Bluetooth Schnittstelle integriert. Über diese kann der EV3 programmiert werden. [13]

2.2.2 Java

Java gehört zur Klasse der objektorientierten Programmiersprachen und erschien in der ersten Version bereits 1995. Sie ist ein Teil der Java-Technologie. Diese besteht zusätzlich noch aus dem Entwicklungswerkzeug JDK und der Laufzeitumgebung JRE. [14]

Java ist selten innovativ, doch zeichnet die Sprache genau das auch aus. In den Sprachkern werden nur Funktionalitäten aufgenommen, die sich bereits in anderen Sprachen bewährt haben. Auch dadurch gilt Java als sehr sicher. [15]
Was Java von anderen Sprachen abgrenzt, ist die Art, wie der geschriebene Code verarbeitet wird. Im Normalfall wird von Übersetzern ein Maschinencode für einen speziellen Anwendungsfall erzeugt, der dann auch nur für diesen anwendbar ist [16]. Java Compiler erzeugen einen sogenannten Bytecode, der von der, in der Laufzeitumgebung JRE beheimateten, „Java Virtual Machine“, kurz JVM, interpretiert und auf die jeweiligen Systembegebenheiten angewendet wird [17]. Somit wird eine Plattformunabhängigkeit geschaffen. Ist auf einem System die JVM installiert, ist es möglich Java Code auszuführen. Somit lässt sich Java sowohl in komplexen Applikationen als auch in Embedded Systems, wie Waschmaschinen, finden [18].

Einzelne Java Dateien werden als Klassen bezeichnet, die Funktionen, die diese bieten als Methoden. [19]
Da Java immer weiter wächst, ist es nahezu unmöglich einen Überblick über alle Funktionalitäten zu behalten. Daher werden aufbauend auf Java verschiedene Frameworks

und Bibliotheken entwickelt, die die Entwicklung komplexer Applikationen erleichtern. Dabei wird der Fokus immer auf einen bestimmten Bereich von Java gelegt. [20]

2.2.3 LeJOS

Eines dieser Frameworks für Java ist „LeJOS“. Dieses kann genutzt werden, um Programme für das Lego Mindstorms System zu schreiben. [21]
Normalerweise wird Software hierfür durch eine leicht anzuwendende Bausteinsprache in einer eigenen Entwicklungsumgebung programmiert [22]. Dies ist für die meisten Anwendungsfälle ausreichend.
Sollen jedoch spezifischere Programme geschrieben werden, empfiehlt es sich auf eine andere Programmiersprache auszuweichen und das Mindstorms System über eine Schnittstelle anzusprechen. Hierfür dient die Bibliothek mit den Namen „LeJOS“. Der Name soll der Marke LEGO ähneln, nur das das „J“ für Java eingefügt wurde. Der Namensteil „JOS“ entspricht dem Akronym „Java Operating System“. Der ganze Begriff stammt auch aus dem spanischen und bedeutet „weit“. Die Entwicklung an diesem Paket wurde bereits 1999 als Hobby eines Entwicklers begonnen. Um dieses Open-Source Projekt bildete sich schnell eine Gruppe an Entwicklern, die die Software immer weiter entwickeln. Seit 2006 wird nicht mehr die Standard Lego Software auf der Zentraleinheit genutzt, sondern eine eigene Firmware mit einer Java Virtual Machine aufgespielt. Die momentan aktuelle Version für den EV3 Brick ist die 0.9.1-beta. [21]
Seit 2008 gibt es auch Plug-Ins für eine der bekanntesten Java Entwicklungsumgebung Eclipse. Diese wird im Folgenden beschrieben.

2.2.4 Entwickeln mit Eclipse und LeJOS

Eclipse ist eine sogenannte „integrierte Entwicklungsumgebung“, kurz IDE. Sie ist ein quelloffenes Programmierwerkzeug zur Entwicklung von Software, ursprünglich nur für JAVA Anwendungen, mittlerweile aber für eine Vielzahl weiterer Programmiersprachen. Den Hauptbestandteil stellt ein Texteditor mit vielen Komfortfunktionen dar, in dem Quellcode geschrieben werden kann. Zusätzlich lassen sich verschiedene Ansichten beliebig im Fenster platzieren. [23]

In Eclipse wird das Framework LeJOS installiert. Damit werden Funktionalitäten eingefügt, die verwendet werden können, um einen Lego Brick zu programmieren. Die Programme werden in Java Klassen geschrieben. Damit eine solche vom Roboter ausgeführt werden kann, benötigt sie eine sogenannte „main-Methode“. Methoden können als gebotene Funktionen des Selbst-geschriebenen Programmes verstanden werden. In dieser „main“ können auch weitere Methoden aufgerufen werden. [24]
Abbildung 2 zeigt eine solche, die auf dem EV3 Brick „Hello World“ ausgeben soll.

```
public class HelloWorld {

    public static void main(String[] args) {
        LCD.drawString("Hello World", 0, 4);
        Delay.msDelay(5000);
    }
}
```

Abbildung 2: Main Methode für die Ausgabe "Hello World" auf dem EV3

Hier sind die Klassen „LCD“ und „Delay“ aus dem Paket des installierten LeJOS Frameworks und sind spezifisch für das Lego Mindstorms System entwickelt worden.
Um diesen Code ausführen zu können, wird der EV3 Brick, der sich im W-LAN befinden muss, in Eclipse gekoppelt und die vorliegende Java Klasse wird als EV3 Programm ausgeführt. Das Programm wird dann an den Brick gesendet und dort erscheint das Ergebnis, welches in Abbildung 3 zu sehen ist.

Abbildung 3: Ausgabe "Hello World" auf Bildschirm des EV3

3. Anforderungen

An dieses Projekt werden gewisse Anforderungen gestellt. Diese werden in folgender Tabelle 1 zusammengefasst.

Muss - Anforderungen	**Kann – Anforderungen**
Auto fährt	Auto parkt auf Wunsch ein
Auto hat eine Lenkung	Auto erkennt Parklücken seitlich, quer
Auto erkennt Parklücken seitlich	Auto parkt automatisch quer rückwärts ein
Auto parkt automatisch seitlich rückwärts ein	
Nachvollziehbare Dokumentation	

Tabelle 1: Anforderungen an das Projekt

4. Konzept

In diesem Kapitel werden die Vorarbeiten vorgestellt, die nötig sind, um einen Algorithmus für das autonome Einparken zu entwickeln. Zunächst wird auf die mathematischen Hintergründe eingegangen. Anschließend wird die Entwicklung des Autos gezeigt. Als letztes wird das Testkonzept vorgestellt.

4.1 Mathematische Betrachtung des Einparkvorgangs

Um relevante Parameter für den Einparkvorgang zu berechnen, wird zunächst eine mathematische Simulation durchgeführt. Die Ziele hierbei liegen darin, zu wissen, wie groß eine Parklücke sein muss, wie weit das Auto nach vorne fahren soll, bevor es mit dem Einparkvorgang beginnen kann und inwiefern der Einparkvorgang verändert werden muss, wenn der Abstand zum nebenstehenden Fahrzeug größer oder kleiner ist. Um diese Fragen zu klären, wird im Folgenden der Verlauf des Einparkvorgangs analysiert.

4.1.1 Allgemeine Betrachtung des Einparkvorgangs

Abbildung 4 zeigt das Konzept.

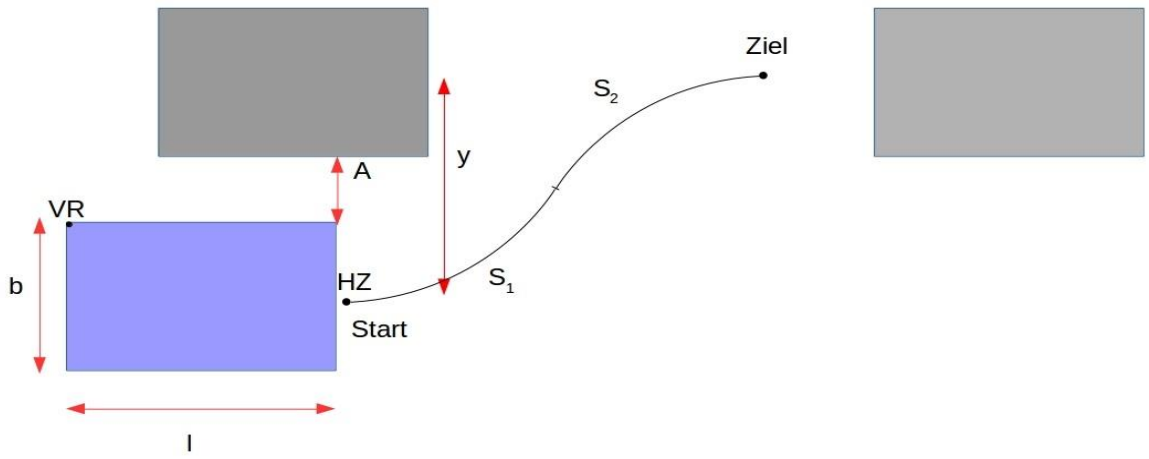

Abbildung 4: Schematischer Einparkvorgang

Tabelle 2 fasst kurz die Bedeutung der einzelnen Zeichen zusammen.

Zeichen	**Bedeutung**
b	Breite des einparkenden Autos
l	Länge des einparkenden Autos
y	Weg, der zum Einparken durchfahren wird
A	Abstand zum nächsten Auto
S_1	Kreisbahn, erste Hälfte des Einparkens
S_2	Kreisbahn, zweite Hälfte des Einparkens
Start	Punkt an dem die Simulation beginnt
Ziel	Punkt an dem die Simulation endet
VR	Vorderster Punkt auf der rechten Seite
HZ	Punkt zentral mittig auf der Hinterachse

Tabelle 2: Verwendete Zeichen im schematischen Einparkvorgang

Der Einparkvorgang setzt sich aus zwei punktsymmetrischen Kreisbahnsegmenten zusammen. Den Symmetriepunkt bildet der Punkt in der Mitte des Einparkvorganges, an dem die Reifen umgeschlagen werden.

Um den in Abbildung 4 gezeigten Verlauf mathematisch beschreiben zu können, ist es erforderlich, die Geometrie des Autos zu betrachten.

Damit das Auto eine Kurve fährt, müssen die Vorderreifen eingeschlagen werden. Dieser Winkel wird als α bezeichnet. Durch eine Rückwärtsbewegung mit eingelenkten Reifen entsteht eine zusätzliche Bewegung, parallel zu den Reifen. Aus der Überlagerung dieser beiden Bewegungen, entsteht die bereits erwähnte Kreisbahn, die mit der Winkelgeschwindigkeit ω durchlaufen wird.

Die beiden Geschwindigkeitsvektoren werden in Abbildung 5 gezeigt.

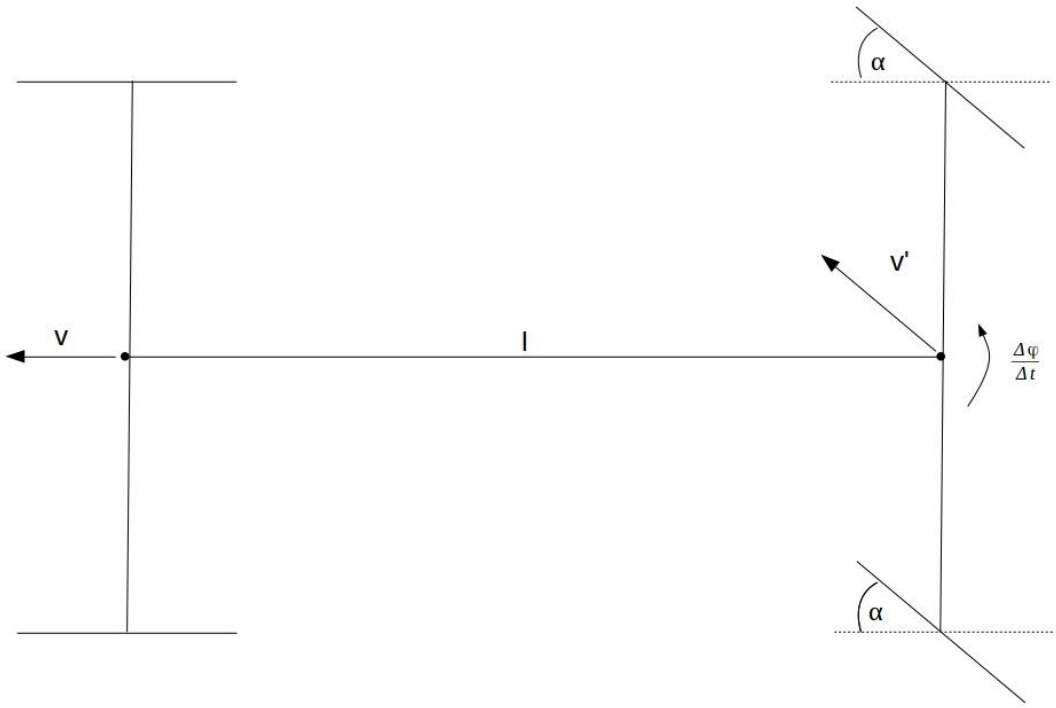

Abbildung 5: Auto beim Einparkvorgang

Auch für diese Abbildung werden die Zeichen in folgender Tabelle 3 kurz erklärt.

Zeichen	**Bedeutung**
v_0	Geschwindigkeit horizontal in Fahrtrichtung
v	Geschwindigkeit in Lenkrichtung
l	Länge des Autos
α	Lenkwinkel beim Einparken
$\frac{\Delta\varphi}{\Delta t}$	Winkelgeschwindigkeit mit der das Auto die Kreisbahn durchläuft

Tabelle 3: Verwendete Zeichen für das Auto beim Einparkvorgang

Die Geschwindigkeit in Lenkrichtung ist abhängig vom Winkel und der Geschwindigkeit horizontal zur Fahrtrichtung. Dies lässt sich anhand eines Dreieckes, in Abbildung 6 gezeigt, veranschaulichen.

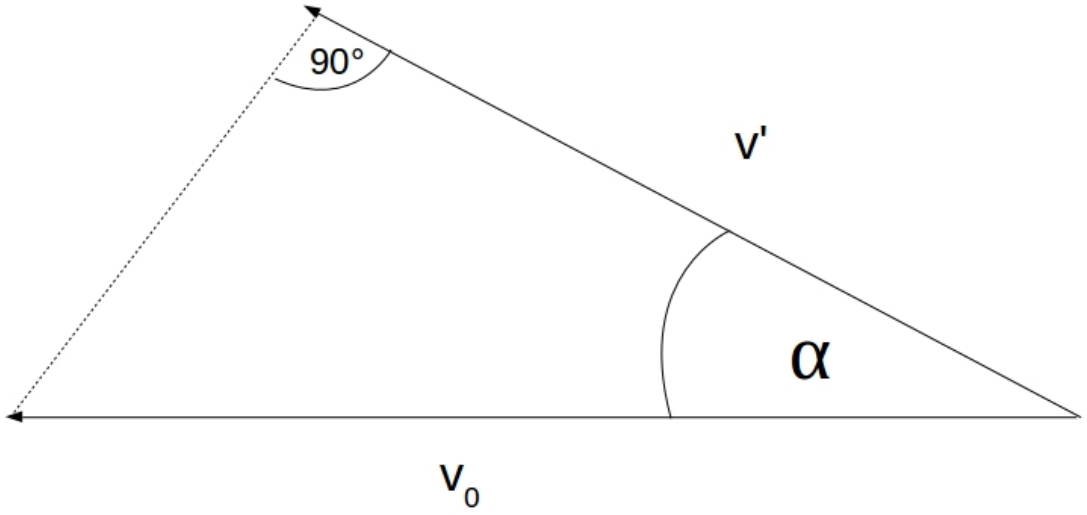

Abbildung 6: Dreieck zur Veranschaulichung des Vektors der Geschwindigkeit

Aufgrund des 90° Winkels in dem Dreieck gilt:

$$v' = cos(\alpha) \cdot v_0 \qquad (1)$$

Gleichermaßen lässt sich v‘ aber auch abhängig von der Winkelgeschwindigkeit ω darstellen. So ist die Geschwindigkeit in Lenkrichtung gleich der Länge des Autos multipliziert mit der Winkelgeschwindigkeit.

$$v' = \omega \cdot l \qquad (2)$$

(1) wird in (2) eingesetzt, um eine Gleichung für ω zu erhalten:

$$\omega = \frac{\Delta\varphi}{\Delta t} = \frac{v_0 * cos(\alpha)}{l} \qquad (3)$$

Mit der Winkelgeschwindigkeit lässt sich nun darstellen, wie sich der Geschwindigkeitsvektor v0 bei der Kurvenfahrt in der Ebene dreht. Dies lässt sich mathematisch durch einen komplexen Raumzeiger in Form von

$$\overrightarrow{v_0} = |v_0| * e^{j\omega t}$$

geschickt beschreiben. Durch diesen Raumzeiger kann für jeden Zeitpunkt der x-Wert mithilfe des Realteils und der y-Werts mithilfe des Imaginärteils in einem Koordinatensystem berechnet werden. Dies ist in Abbildung 7 gezeigt.

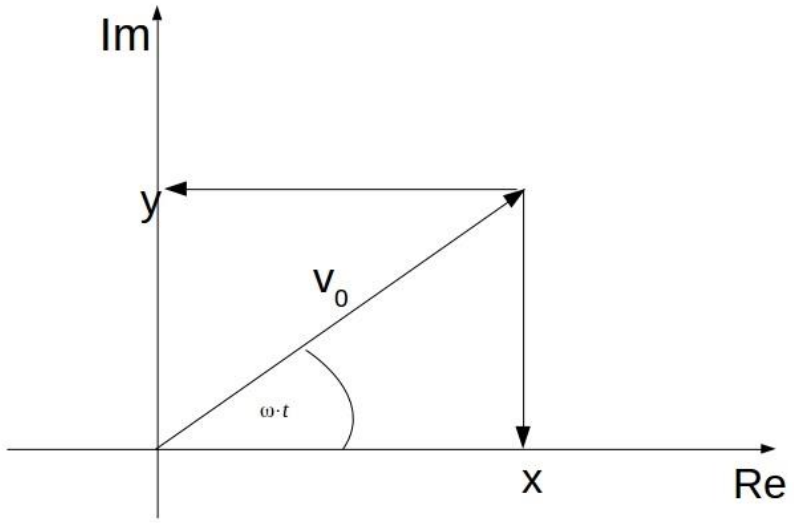

Abbildung 7:Realteil und Imaginärteil im Koordinatensystem

Daraus lässt sich der Raumzeiger der Strecke bestimmen, da dies das Integral des Raumzeigers der Geschwindigkeit ist. In folgender Gleichung wird dieses Integral aufgestellt und berechnet.

$$\overrightarrow{s(t)} = \int_0^t \overrightarrow{v_0(\tau)}\, dt =$$

$$= \int_0^t |v_0|\, e^{j\omega\tau} d\tau =$$

$$= -j\frac{|v_0|}{\omega} \cdot \left(e^{j\omega t} - 1\right) =$$

$$= j\frac{|v_0|}{\omega} \cdot \left(1 - e^{j\omega t}\right) =$$

$$= \frac{|v_0|}{\omega} \cdot \left(j + e^{j\cdot\left(\omega t - \frac{\pi}{2}\right)}\right) =$$

$$= j\frac{|v_0|}{\omega} + \frac{|v_0|}{\omega} \cdot e^{j\cdot\left(\omega t - \frac{\pi}{2}\right)} \qquad (4)$$

Der erhaltene Raumzeiger stellt S_1 dar.

Um einen solchen auch für S_2 zu erhalten, wird der Geschwindigkeitsvektor mit einer negativen Winkelgeschwindigkeit erneut integriert, da in S_2 derselbe Weg durchfahren wird wie in S_1, jedoch in die entgegengesetzte Richtung.

$$\overrightarrow{v_0} = |v_0| * e^{-j\omega t}$$

$$\overrightarrow{s(t)} = \int_0^t \overrightarrow{v_0(\tau)}\, dt =$$

$$= \int_0^t |v_0|\, e^{-j\omega\tau} d\tau =$$

$$= j\frac{|v_0|}{\omega} \cdot \left(e^{-j\omega t} - 1\right) =$$

$$= j\frac{|v_0|}{\omega} \cdot \left(-1 + e^{-j\omega t}\right) =$$

$$= \frac{|v_0|}{\omega} \cdot \left(-j + e^{j\cdot(\omega t)}\right) =$$

$$= -j\frac{|v_0|}{\omega} + \frac{|v_0|}{\omega} \cdot e^{j\cdot(\omega t)} \qquad (5)$$

4.1.2 Bestimmung des Umlenkzeitpunktes

Damit das Auto optimal in der Parklücke steht, muss auf der y-Achse der Weg

$$A + b$$

zurückgelegt werden. Da die beiden Kreissegmente S_1 und S_2 symmetrisch sind, liegt der Umschlagpunkt in der Mitte, also bei

$$\frac{A + b}{2}$$

Es ist also möglich, mithilfe der in 4.1.1 hergeleiteten Gleichungen, einen Algorithmus zu entwickeln, der in Abhängigkeit vom Abstand zum nebenstehenden Auto den Umschlagzeitpunkt bestimmt.

Im Folgenden sollen die hergeleiteten Zusammenhänge in einer Simulation verifiziert werden. Hierfür wird ein Softwaretool namens Scilab verwendet. Scilab ist ein open-source Programm, das für Anwendungen in der numerischen Mathematik entwickelt wurde. Es bietet, unter Anderem, Funktionen zum Plotten von 2D und 3D Kurven. [25]

Um die Darstellung in einem Programm zu vereinfachen, werden die Gleichungen (4) und (5) jeweils in Realteil und Imaginärteil aufgeteilt. Aus (4) ergibt sich:

$$\text{realteil\_von\_(4)} = \frac{v_0}{\omega \cdot cos\left(\omega \cdot t(k) - \frac{\pi}{2}\right)} \qquad (6)$$

$$\text{imaginärteil\_von\_(4)} = \frac{v_0}{\omega} + \frac{v_0}{\omega \cdot sin\left(\omega \cdot t(k) - \frac{\pi}{2}\right)} \qquad (7)$$

Aus (5) ergibt sich:

$$\text{realteil\_von\_(5)} = \frac{v_0}{\omega \cdot cos(-\omega \cdot t(k) + \pi)} \qquad (8)$$

$$\text{imaginärteil\_von\_(5)} = \frac{v_0}{\omega} + \frac{v_0}{\omega \cdot sin(-\omega \cdot t(k) + \pi)} \qquad (9)$$

Des Weiteren wurde in den oberen Gleichungen die Variable t durch $t(k)$ersetzt, wobei k als Laufvariable dient. In Abbildung 8 sind die Gleichungen 6 - 9 im Scilab Code zu sehen.

```
function xdot=f(t,x,x_d,zeitdiskret)
    global k realteil1 imaginaerteil1 realteil2 imaginaerteil2 realteil1AmWendepunkt imaginaerteil1AmWendepunkt
    k = 1;
    if zeitdiskret == 0 then
        xdot(1) = 0;

    else
        if x_d(2) < ((A+b)/2) then
            realteil1 = v_0/omega*cos(omega*t(k)-%pi/2);
            imaginaerteil1 = v_0/omega + v_0/omega*sin(omega*t(k)-%pi/2);
            xdot(1) = realteil1;
            xdot(2) = imaginaerteil1;

            realteil1AmWendepunkt = realteil1;
            imaginaerteil1AmWendepunkt = imaginaerteil1;

        else

            realteil2 = v_0/omega*cos(-omega*t(k)+%pi) + 2*realteil1AmWendepunkt;
            imaginaerteil2 = v_0/omega + v_0/omega*sin(-omega*t(k)+%pi) - 2*(v_0/omega - imaginaerteil1AmWendepunkt) ;
            xdot(1) = realteil2;
            xdot(2) = imaginaerteil2;

        end

        k = k+1;
    end
endfunction
```

Abbildung 8: Scilab Code für die Simulation des Einparkvorgangs

Die obenstehende Funktion „xdot“ verhält sich ähnlich einer „for“-Schleifen, da so oft Werte berechnet werden, bis k seinen definierten Maximalwert angenommen hat [26]. In diesem Fall läuft k von 0s bis 4s und das pro Durchlauf in 0,1s Schritten.

Die erste „if“-Bedingung in Abbildung 8 dient zur Berechnung kontinuierlicher Differenzialgleichungen, die in dieser Simulation keine Anwendung findet. Sie ist jedoch für den Aufruf der Funktion „xdot“ erforderlich. Erst durch den Sprung in den „else“ - Zweig wird die eigentliche Rechnung begonnen.

Die Bedingung „if x_d(2) < ((A+b)/2)“ ist solange zutreffend, bis das Auto zur Hälfte in der Parklücke steht, vgl. Abbildung 4. Im „else“ - Zweig dieser Bedingung stehen die Gleichungen (8) und (9) in einer leicht modifizierten Variante. Da die Simulation der Bewegung nicht erneut am Nullpunkt beginnen sollen, werden die Ergebnisse des letzten Schleifendurchlaufs für die Bewegung S_1 auf die entsprechenden Real – und Imaginärteil aufaddiert, sodass S_2 direkt an S_1 anschließt.

Die weiteren verwendeten Formelzeichen werden in Abbildung 9 eingeführt.

```
//Definition der Konstanten
n = 1; //[Umdrehungen/s]
r_Reifen = 0.0275; //[m]
v_0 = n*2*r_Reifen*%pi; //[m/s] Fahrtgeschwindigkeit

alpha = 45; //[°] Winkel zwischen eingelenktem Reifen und Parallele zum Auto, ergo "Lenkwinkel"
l = 0.225; //[m] Länge des Autos
omega = v_0*cos(alpha)/l; //[1/s] Winkelgeschwindigkeit für die Kreisbewegung des Autos

b = 0.195; //[m] Breite des Autos
A = 0.06; //[m] Abstand zum nächsten Auto

t_0 = 0;
t_max = 4;
Delta_t = 0.1;
t = [t_0:Delta_t:t_max]; //[s] Zu untersuchende Zeitpunkte von 0s bis 4s in 0,1s-Schritten
```

Abbildung 9: Konstanten in der Scilab Simulation

Das Ergebnis dieser Simulation ist in Abbildung 10 zu sehen.

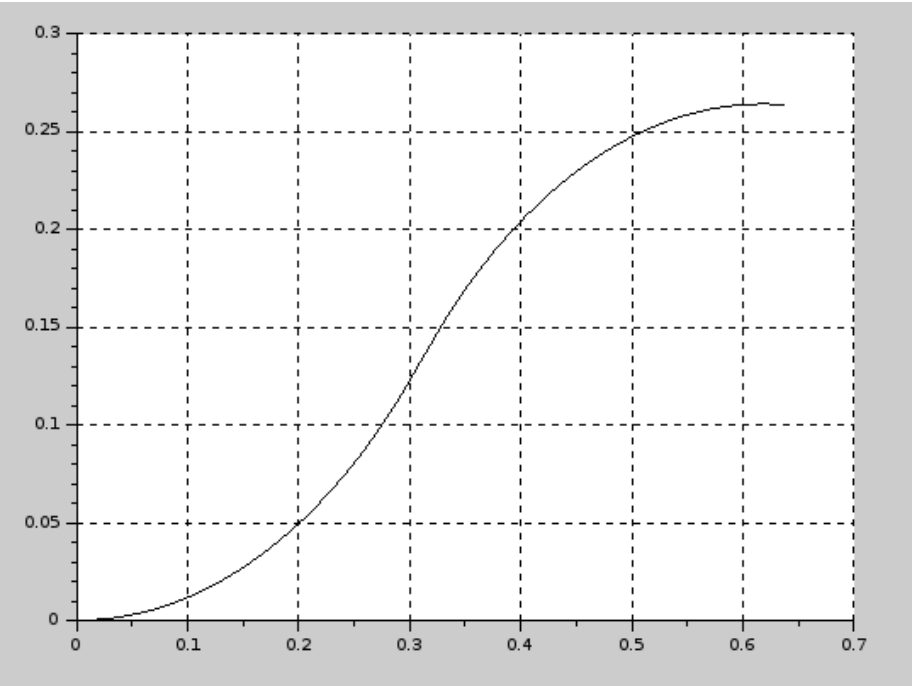

Abbildung 10: Scilab Simulation des Punktes HZ beim Einparkvorgang

Hier bleibt zu bemerken, dass das Auto in diesem Fall vom Punkt HZ simuliert wurde, damit ist gemeint, dass HZ auf dem Ursprung liegt.

Der Verlauf entspricht dem erwarteten Ergebnis. Mithilfe der Simulation wurden die erarbeiteten Gleichungen verifiziert. Diese sollen als Grundlage für den Einparkalgorithmus dienen. Mit ihnen wird ermöglicht, eine Vorhersage über den Umschlagzeitpunkt in Abhängigkeit vom Abstand A zu treffen.

4.1.3 Bestimmung der minimalen Parklückenlänge

Im zweiten Schritt bei der mathematischen Betrachtung wird untersucht, wie viel Abstand das Auto in x-Richtung zum nebenstehenden Fahrzeug haben soll, um die Größe der Parklücke zu minimieren. Entscheidend hierfür ist der Punkt VR, vergleiche Tabelle 2 und der linke hintere Punkt an dem Auto, hinter dem eingeparkt werden soll. Abbildung 11 veranschaulicht die Problematik.

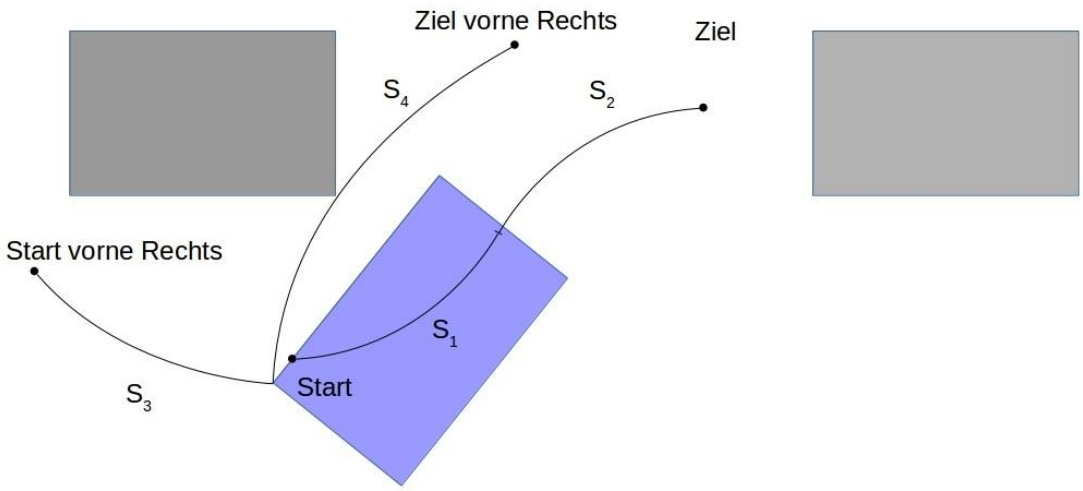

Abbildung 11: Schematischer Einparkvorgang, ergänzt um Kurve des Punktes VR

Wie in der Abbildung zu sehen, verhält sich VR anders als HZ, in Kapitel 4.1.1 beschrieben. Die Kurve S_4 darf das erste Hindernis nie schneiden, ansonsten würde es zu einer Kollision kommen. Um Gleichungen für das Verhalten der Kurven S_3 und S_4 aufstellen zu können, muss das einparkende Auto genauer betrachtet werden.

Die Bahnen S3 und S4 lassen sich aus den Werten für S1 und S2, mithilfe von Transformationsgleichungen, bestimmen. Diese werden im Folgenden hergeleitet.

Abbildung 12 stellt das einparkende Auto mit den für die Herleitung benötigten geometrischen Zusammenhängen dar.

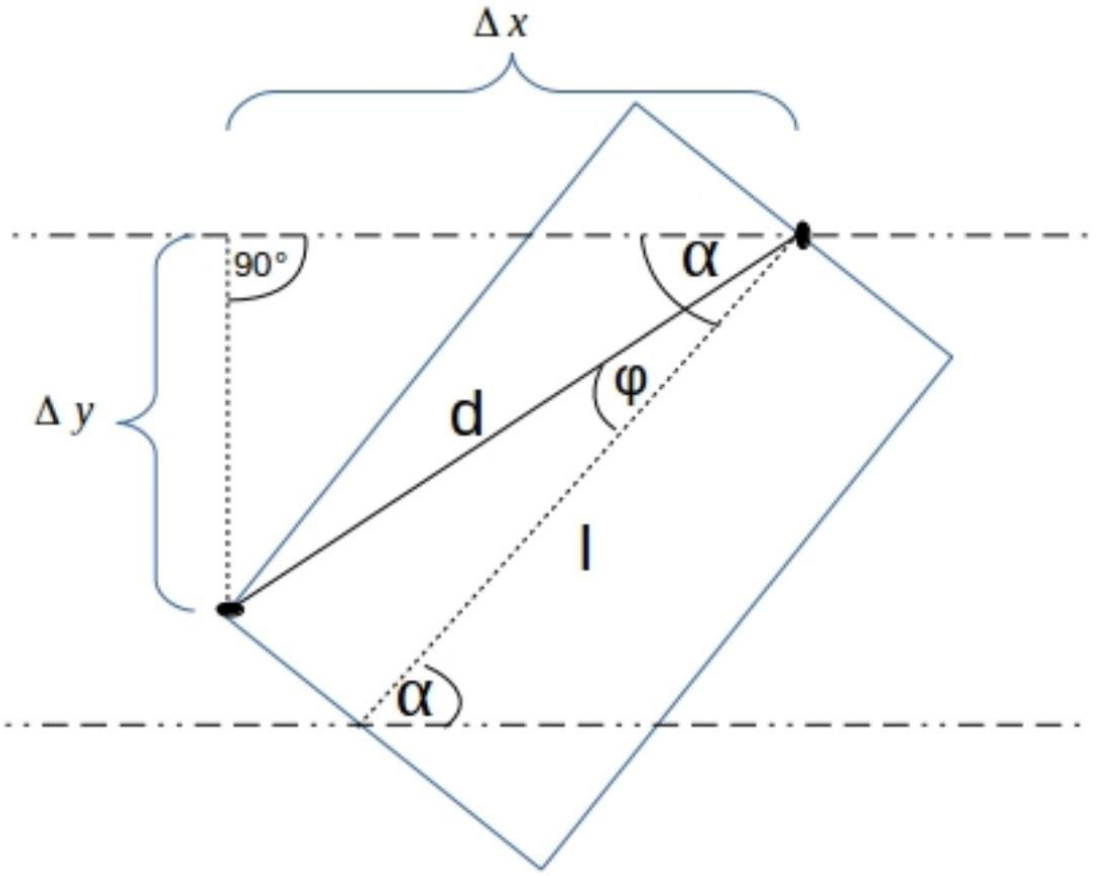

Abbildung 12: Analyse des einparkenden Autos

Der Zusammenhang zwischen VR und HZ lässt sich mithilfe von zwei verschiedenen Betrachtungen beschreiben. Zum einen werden die zeitlich konstanten Winkelzusammenhänge aus der Autogeometrie betrachtet. Diese überlagern sich zum anderen mit der zeitabhängigen Stellung des Autos in der Ebene.

Die zeitabhängige Komponente lässt sich durch den Winkel ∝ beschreiben. Dieser ändert sich mit der Steigung auf dem jeweiligen Punkt der Kurve.

Die zeitunabhängigen Komponenten sind der Winkel Phi und die Strecke d.

Die Strecke d verbindet die Punkte VR und HZ und lässt sich durch den Satz des Pythagoras bestimmen, da die anderen Seiten des rechtwinkligen Dreiecks bekannt sind. Zum einen ist es die Hälfte der Breite, zum anderen die Länge des Autos. Somit ergibt sich für d folgende Gleichung.

$$d = \sqrt{\frac{b^2}{4} + l^2} \qquad (10)$$

Auch ist in Abbildung 12 das Steigungsdreieck eingezeichnet, dass die Punkte VR und HZ verbindet. Dadurch lassen sich für jeden berechneten Wert von HZ auch der entsprechende Wert für VR bestimmen. Die folgenden beiden Gleichungen stellen den Unterschied in x-Richtung und in y-Richtung dar.

$$\Delta y = d \cdot sin(\alpha - \varphi) \quad (11)$$

$$\Delta x = d \cdot cos(\alpha - \varphi) \quad (12)$$

Um Δx und Δy zu berechnen, werden die Werte der in der Gleichung vorhandenen Winkel α und φ benötigt. φ ist rein abhängig von festen Werten, die durch das Auto festgelegt werden, nämlich der Breite und der Länge.

$$\varphi = tan\left(\frac{b}{2l}\right) \qquad (13)$$

α ist im Gegensatz zu φ abhängig von der Stellung des Autos im Vergleich zum Raum und stellt die Steigung am Punkt VR dar.

$$\alpha = \frac{sin(\omega t)}{cos(\omega t)} \qquad (14)$$

Die Gleichungen (10) bis (14) werden für die Scilab – Simulation verwendet. Der Code für die erweiterte Simulation ist in Abbildung 13 zu sehen.

```
function xdot=f(t,x,x_d,zeitdiskret)
    global k realteil1 imaginaerteil1 realteil2 imaginaerteil2 realteil1AmWendepunkt imaginaerteil1AmWendepunkt m delta_y delta_x
    k = 1;
    if zeitdiskret == 0 then
        xdot(1) = 0;

    else
        if x_d(2) < ((A+b)/2) then
            realteil1 = v_0/omega*cos(omega*t(k)-%pi/2);
            imaginaerteil1 = v_0/omega + v_0/omega*sin(omega*t(k)-%pi/2);
            xdot(1) = realteil1;
            xdot(2) = imaginaerteil1;

            realteil1AmWendepunkt = realteil1;
            imaginaerteil1AmWendepunkt = imaginaerteil1;

            m = sin(omega * t(k)) / cos(omega * t(k));
            delta_y = d * sin(m - phi);
            delta_x = d * cos(m - phi);

            xdot(3) = realteil1 - delta_x;
            xdot(4) = imaginaerteil1 - delta_y;
            xdot(5) = m;

        else

            realteil2 = v_0/omega*cos(-omega*t(k)+%pi) + 2*realteil1AmWendepunkt;
            imaginaerteil2 = v_0/omega + v_0/omega*sin(-omega*t(k)+%pi) - 2*(v_0/omega - imaginaerteil1AmWendepunkt) ;
            xdot(1) = realteil2;
            xdot(2) = imaginaerteil2;

            m = (sin(-omega*t(k)+(3*%pi/2)) / cos(-omega*t(k)+(3*%pi/2)));
            delta_y = d * sin(m - phi);
            delta_x = d * cos(m - phi);

            xdot(3) = realteil2 - delta_x;
            xdot(4) = imaginaerteil2 - delta_y;
            xdot(5) = m;

        end

        k = k+1;
    end
endfunction
```

Abbildung 13: Erweiterte Scilab Simulation, sowohl HZ als auch VR

Für die Simulation wird zusätzlich noch der kritische Punkt am vorderen Auto, hinter dem eingeparkt wird, mit angezeigt. Er befindet sich im Abstand von 6 cm vom einparkenden Fahrzeug. Die Simulation ist in folgender Abbildung 14 zu sehen.

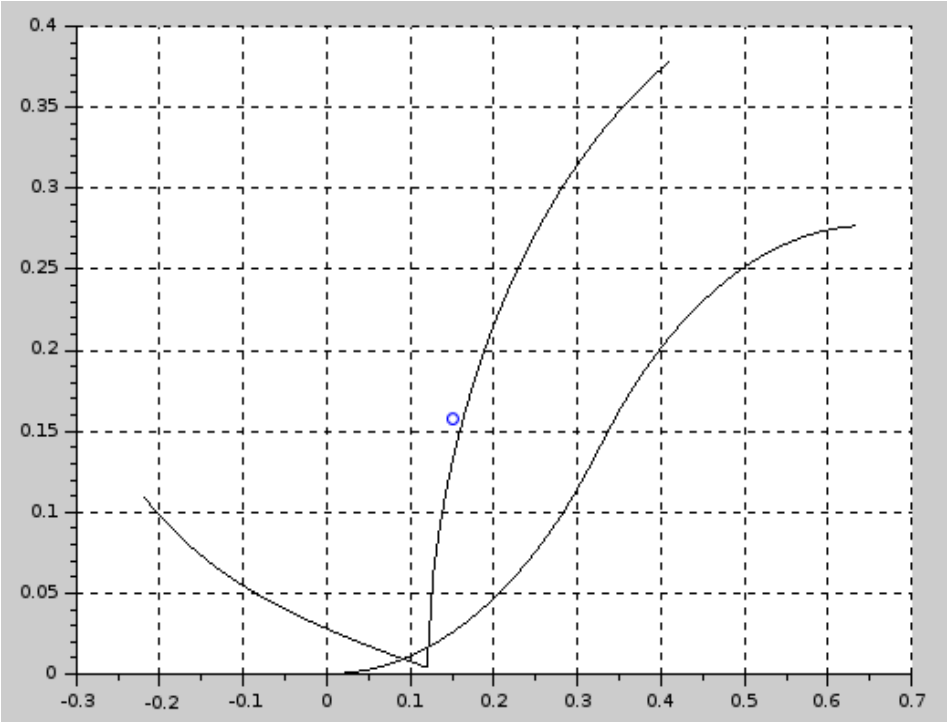

Abbildung 14: Scilab Simulation der Punkte HZ und VR beim Einparkvorgang

4.1.4 Ergebnisse aus der mathematischen Simulation

Aus der Abbildung 10 in Kapitel 4.1.2 geht hervor, dass bei einem Abstand von sechs Zentimetern des Autos zum rechtsstehenden Fahrzeug für den Einparkvorgang 26 cm überstrichen werden. Somit liegt der Umschlagpunkt für die Lenkung bei 13 cm. An der x-Achse lässt sich ablesen, dass die Parklücke mindestens 60 cm groß sein muss.
Um die Größe der Parklücke optimieren zu können, wurde in Kapitel 4.1.3 der kritische Punkt VR des einparkenden Autos mit dem nebenstehenden Auto simuliert. Daraus lässt sich ablesen, dass das Auto 15 cm an dem nebenstehenden Auto vorbeifahren kann. Wenn der Einparkvorgang erst dann gestartet wird, sollte es keinen Unfall geben. Damit lässt sich die Größe der Parklücke auf circa 45 cm verringern.

Außerdem lässt sich erkennen, dass die Gleichungen mithilfe der Simulation verifiziert wurden und für den Algorithmus verwendet werden können.

4.2 Konstruktion des Autos

Für das Mindstorms System werden Anleitungen für mögliche Roboter bereits mitgeliefert. [27] Für diese Arbeit soll das Fahrzeug möglichst einem realen Auto ähneln. Dies ist ursprünglich nicht vorgesehen. Somit wird das Auto neu designt.

4.2.1 Konzept

Besonders auf die Lenkung muss wert gelegt werden, so dass sich das Auto möglichst real steuern lässt. Gelenkt wird bei vielen Autos an der Vorderachse, während an der Hinterachse der Motor die Reifen antreibt und so für die Geschwindigkeit sorgt [5]. Somit werden in dem Fahrzeug zwei Motoren benötigt.
Bei den Sensoren werden zwei Ultraschallsensoren für die Abstandsmessung und ein Berührungssensor für das Starten des Einparkvorgangs benötigt. Abbildung 15 zeigt die Positionen der Sensoren und Motoren und der Zentraleinheit.

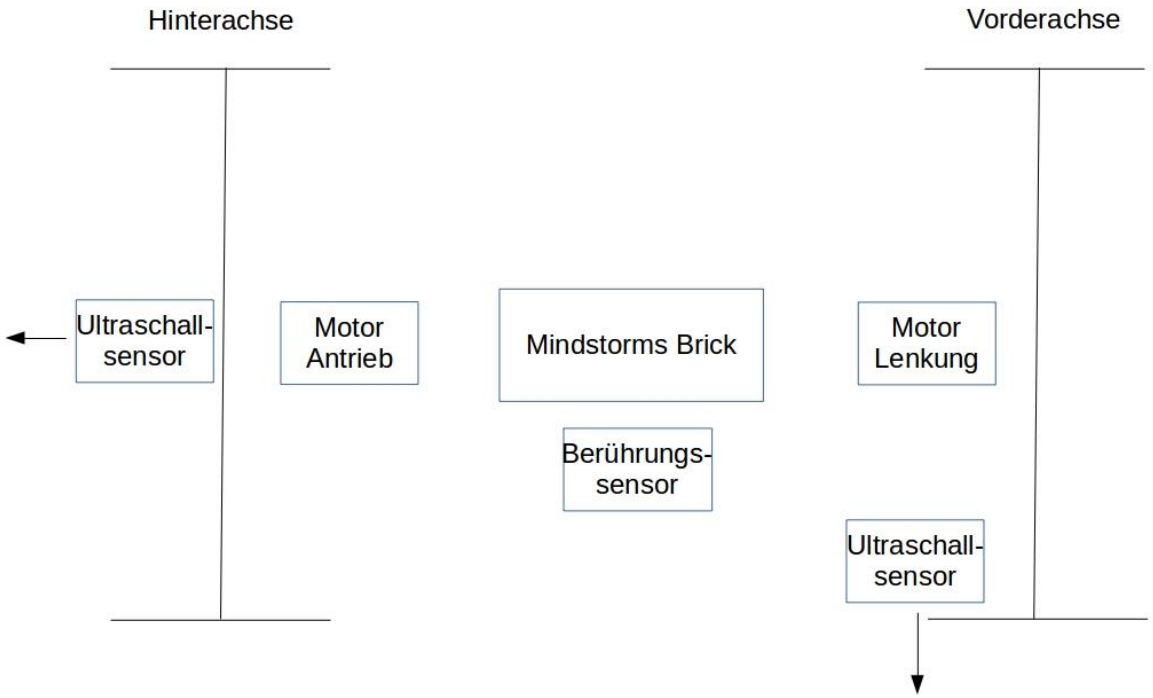

Abbildung 15: Positionierung der Sensoren und Motoren auf dem Fahrzeug

Der Ultraschallsensor an der Vorderachse überprüft den Abstand des Autos zur rechten Seite. Damit wird verhindert, dass das Auto einem Hindernis zu nahe kommt. Ebenso wird die Tiefe einer Parklücke vermessen und der Start- und Endzeitpunkt einer Parklücke kann bestimmt werden. Der Ultraschallsensor an der Hinterachse dient zur Überprüfung, ob beim Einparken das hintere Hindernis in Gefahr ist. Dies ist ein reiner Sicherheitsmechanismus, da theoretisch durch die Berechnung ein Anstoßen nach hinten ausgeschlossen wird.
Der Berührungssensor wird zentral am Fahrzeug angebracht. Im Normalfall fährt das Auto geradeaus. Soll eingeparkt werden, kann der Fahrzeugführer durch betätigen des Sensors in den Einparkmodus wechseln. Ab diesem Zeitpunkt überprüft das Auto alle möglichen Parklücken und startet bei positivem Bewertungsergebnis den Einparkvorgang.
Die beiden Motoren dienen an der Vorderachse zum Lenken des Fahrzeugs und an der Hinterachse zum Beschleunigen des Fahrzeugs.
An die Zentraleinheit werden alle Sensoren und Motoren angeschlossen. Ebenso läuft das Anwendungsprogramm auf diesem ab. Sie wird möglichst zentral und nahe dem eigentlichen Fahrzeug positioniert. Dies ist notwendig, da aufgrund des Gewichts das System schwingt, je weiter der Brick vom Boden entfernt ist.

4.2.2 Lenkung

Die Herausforderung bei der Lenkung liegt darin, die vertikale Bewegung des Motors in eine horizontale zu verwandeln, so dass die Reifen gedreht werden können. Abbildung 16 zeigt das Auto von der Vorderansicht.

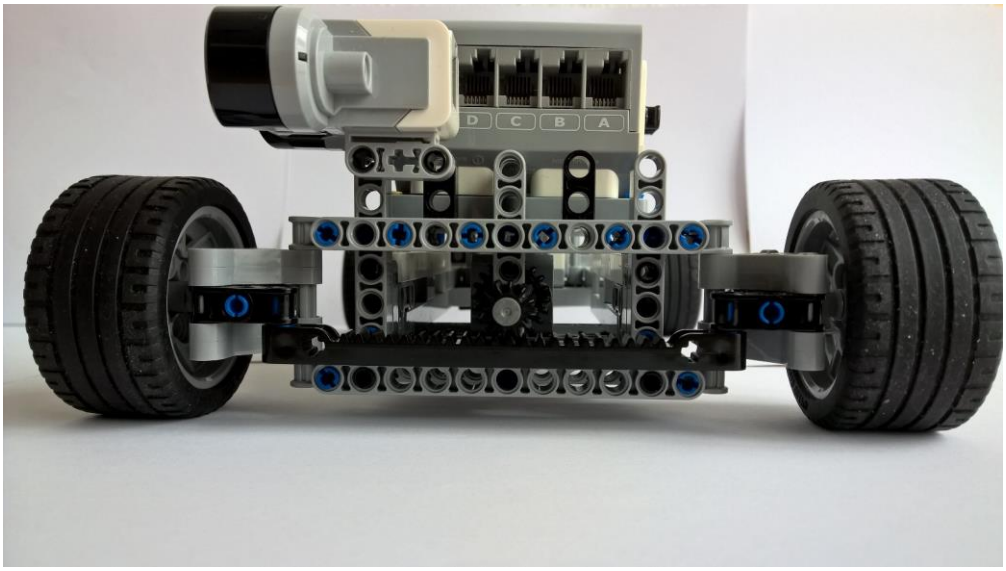

Abbildung 16: Vorderansicht des Autos

Der Motor rotiert für eine bestimmte Gradzahl und bewegt damit das in Abbildung 16 zu sehende Zahnrad zentral vorne am Auto. Das Zahnrad greift in eine ebenso gezahnte Leiste, die dann die Reifen verdreht. Dadurch kommt die Lenkbewegung zustande.

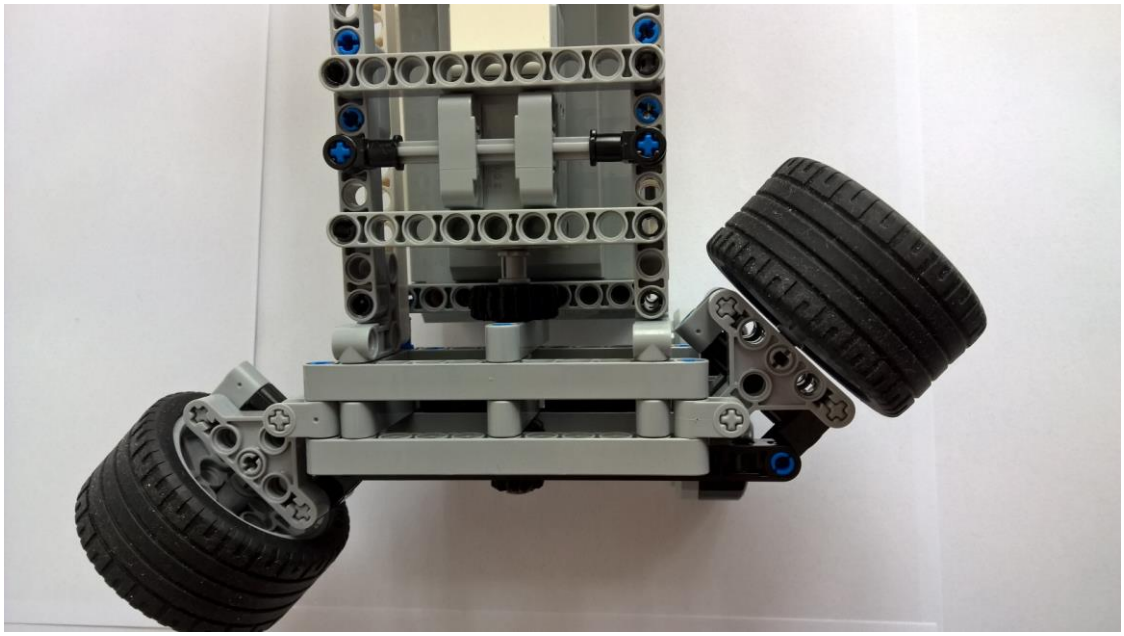

Abbildung 17: Lenkung des Autos von unten

Um die Reifen auch manuell bewegen zu können, wurde ein weiteres Zahnrad angebracht, das in Abbildung 17 zu sehen ist. Dieses kann verwendet werden, um die Reifen gerade auszurichten.

4.2.3 Ergebnis des Autobaus

Abbildung 18 zeigt das Auto, wie es zusammengesetzt aussieht.

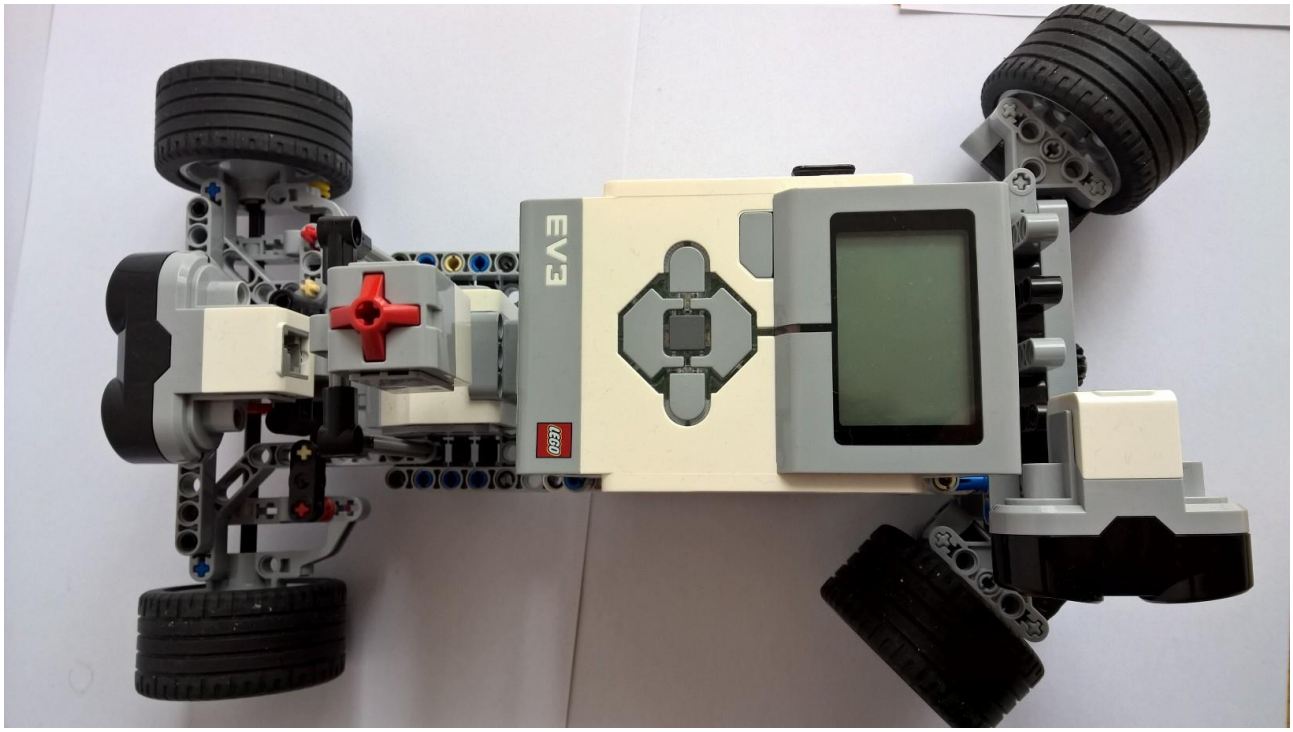

Abbildung 18: Ansicht von oben auf das Auto

Zur besseren Sichtbarkeit wurde für dieses Bild auf die benötigten Kabel zwischen Zentraleinheit, Sensoren und Aktoren verzichtet.

4.2.4 Softwarekonzept anhand eines Zustandsübergangsdiagramms

Der Algorithmus wird in Form eines Zustandsübergangsdiagramms dargestellt. Dieser ist in Abbildung 19 zu sehen.

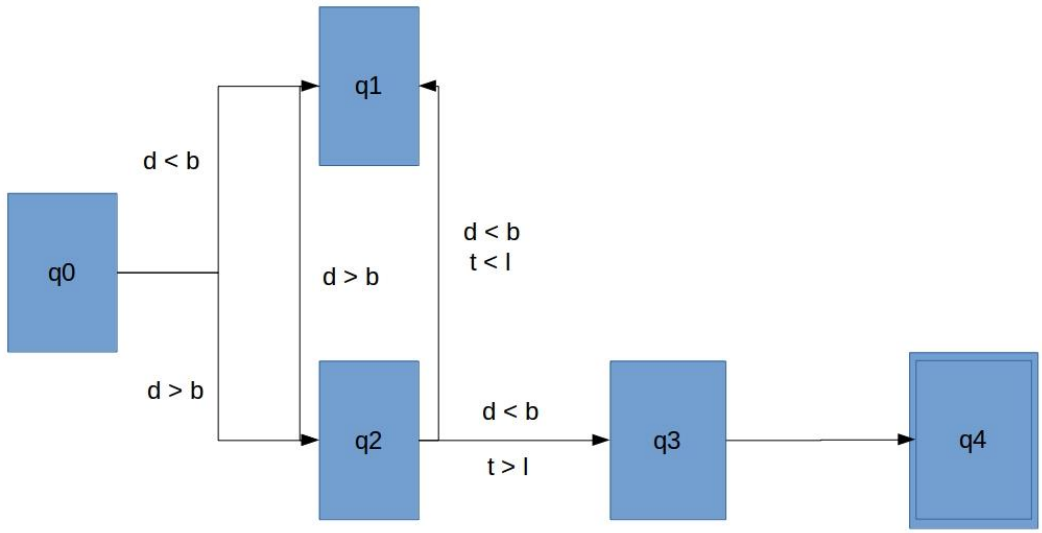

Abbildung 19: Zustandsübergangsdiagramm

In folgender Tabelle 4 wird die Bedeutung der in Abbildung 19 verwendeten Formelzeichen erklärt.

Zeichen	**Bedeutung**
q0, q1, q2, q3, q4	Zustände des Programmablaufs
d	Abstand des Autos nach rechts
b	Maximaler Abstand zu Beginn des Einparkens
t	Abstand zwischen zwei Hindernissen, bzw. Gemessene Größe der Parklücke
l	Mindestlänge der Parklücke

Tabelle 4: Erklärung der Zeichen im Zustandsübergangsdiagramm

Der Einparkvorgang beginnt im Zustand q0. Für den nächsten gibt es zwei Möglichkeiten. Befindet sich rechts neben dem Fahrzeug ein Hindernis in weniger als 15 Zentimetern Abstand wechselt das Programm in den Zustand q1. Sowohl aus q0 also auch aus q1 kann in den Zustand q2 gesprungen werden, falls der Abstand nach rechts größer als 15 Zentimetern ist, also eine Parklücke vorhanden ist. In diesem Zustand wird zu Beginn der aktuelle Stand des Motors aufgenommen, um später überprüfen zu können, ob die Parklücke groß genug ist. Das Auto fährt nun an diesem Punkt weiter geradeaus, bis das nächste Auto auf der rechten Seite erkannt wird. Hier wird erneut der Stand des Motors aufgenommen und mit dem zum Start der Lücke verglichen. Ergibt die Prüfung, dass die Parklücke groß genug ist, wird in den Zustand q3 gesprungen und der Einparkvorgang eingeleitet. Ist die Lücke zu klein wird zurück in den Zustand q1 gewechselt.

4.3 Testkonzept

Wenn die Durchführung abgeschlossen ist, muss auch das korrekte Verhalten des Autos getestet werden. Abbildung 20 zeigt den geplanten Testlauf.

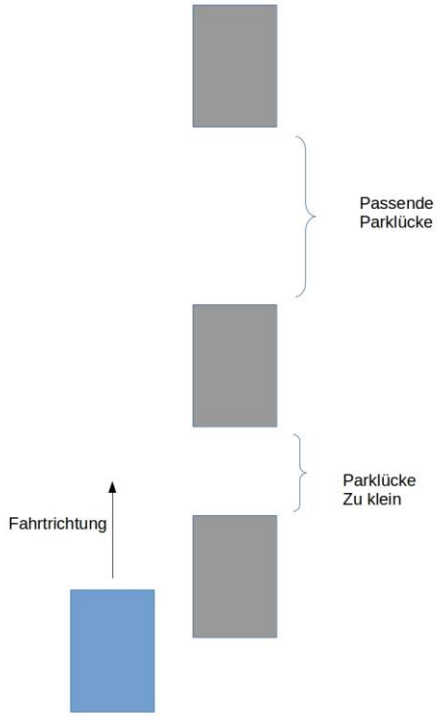

Abbildung 20: Testdurchlauf

Der Parcours soll zweimal durchfahren werden. Im ersten Durchlauf ohne Aktivieren des Parkassistenten, so dass das Auto einfach geradeaus vorbeifährt. Im zweiten Durchlauf wird dann der Parkassistent neben dem ersten Fahrzeug aktiviert. Die erste Parklücke soll ignoriert werden, da sie zu klein ist. Die zweite Parklücke jedoch ist passend von der Größe. Das Auto soll hier einparken.

5. Beschreibung der Umsetzung

In diesem Kapitel wird die praktische Umsetzung beschrieben. Zunächst wird auf die Software, die in der Entwicklungsumgebung Eclipse mit dem Framework LeJOS, wie in Kapitel 2.2.4 beschrieben, realisiert wurde, beschrieben. Anschließend wird das Ergebnis des Testkonzeptes präsentiert. Zum Schluss wird eine Soll-/Ist-Analyse durchgeführt und es wird auf Probleme eingegangen.

5.1 Erklärung des Code

Zu Beginn des Quellcodes, werden die Motoren und Sensoren angelegt und aktiviert. Ebenso werden benötigte Variablen und Konstanten deklariert und initialisiert. Die Abbildungen 21, 22, 23 und 24 zeigen dies.

```
// Motoren
public static RegulatedMotor motorLenkung = new EV3MediumRegulatedMotor(
        MotorPort.A);
public static RegulatedMotor motorAntrieb = new EV3LargeRegulatedMotor(
        MotorPort.B);
```

Abbildung 21: Motoren

```
// Sensoren
public static EV3UltrasonicSensor ultraschallSensorVorne = new EV3UltrasonicSensor(
        SensorPort.S1);
public static EV3UltrasonicSensor ultraschallSensorHinten = new EV3UltrasonicSensor(
        SensorPort.S2);
public static EV3TouchSensor touchSensorOben = new EV3TouchSensor(
        SensorPort.S3);
```

Abbildung 22:Sensoren

```
// Steuervariablen
public static int zustandDesEinparkens = -1;
static float parklueckenBeginn = 0;
static float parklueckenEnde = 0;
static float abstandNachRechtsZuBeginnDesEinparkens = 0;
static double streckeBiszurHaelfteInDerParkluecke;
```

Abbildung 23: Variablen

```
// Autokonstanten
static final double breiteDesAutos = 0.195;
static final double laengeDesAutos = 0.225;
static final double streckeProReifenUmdrehung = 0.1728;
static final double abstandSensorRechtsZumReifenRechts = 0.04;
```

Abbildung 24: Auto-abhängige Konstanten

Wie in Kapitel 2.2.4 beschrieben, ist das zentrale Element eines jeden Java Programmes, die Main-Methode. Diese macht Quellcode erst ausführbar. Abbildung 25 zeigt sie für den Einparkvorgang.

```
public static void main(String[] args) {

    while (Button.ESCAPE.isUp()) {
        fahreVorwaertsUndWarteAufEinparksignal();
        if (zustandDesEinparkens != -1) {
            motorAntrieb.flt();
            parkeEin();
        }
    }
}
```

Abbildung 25: Main-Methode der Software

Wie zu erkennen, ist diese sehr kurz gehalten und besteht nur aus fünf Zeilen Code. Zentraler Inhalt ist eine „While“-Schleife. Diese überprüft zu Beginn jedes Durchlaufes, ob die Bedingung zu „true“ evaluiert und führt dann den Code im Körper der Anweisung durch. In diesem Fall wird in der Bedingung überprüft, ob der „Escape“-Button am EV3 Brick nicht gedrückt wurde. Im Körper wird zunächst eine Methode aufgerufen, „fahreVorwartsUndWarteAufEinparksignal“. Diese wird in Abbildung 26 gezeigt.

```
private static void fahreVorwaertsUndWarteAufEinparksignal() {
    motorAntrieb.setAcceleration(150);
    motorAntrieb.setSpeed(200);
    motorAntrieb.backward();

    einparkVorgangAusloeser.fetchSample(touchSensorObenMesswerte, 0);
    if (touchSensorObenMesswerte[0] == 1 && zustandDesEinparkens == -1) {
        wechselZustand(0, -motorAntrieb.getTachoCount());
        Button.LEDPattern(5);
        Sound.beep();
        sucheParkluecke();
    }
}
```

Abbildung 26: Methode zum vorwärts fahren

In dieser Methode wird für den Antriebsmotor die Geschwindigkeit und die Beschleunigung gesetzt. Anschließend wird die Bewegung begonnen. Hier ist zu bemerken, dass der Motor sich „backward“, also rückwärts drehen muss, damit das Auto vorwärts fährt. Dies hat bauliche Gründe. In der „if“-Bedingung wird abgeprüft, ob der Berührungssensor betätigt wurde und ob gleichzeitig die Variable „zustandDesEinparkens“ den Wert -1 hat. Falls dies zutrifft, wird die Variable in den Zustand 0 gewechselt und die aktuelle Anzahl an Grad, die der Motor bereits gedreht hat, wird abgespeichert. Ebenfalls werden ein visuelles und ein akustisches Signal ausgegeben, sodass der Fahrer informiert ist, dass die Suche nach einer Parklücke nun beginnen wird. Hierfür wird wieder eine Methode aufgerufen, die in Abbildung 27 zu sehen ist. Diese implementiert den Zustandsautomat aus Kapitel 4.2.4 in Abbildung 19.

```
private static void sucheParkluecke() {
    while (zustandDesEinparkens != 3) {

        if (zustandDesEinparkens == 0) {

            seitenAbstandNachRechts.fetchSample(
                    seitenAbstandNachRechtsMesswerte, 0);

            if (seitenAbstandNachRechtsMesswerte[0] < 0.15
                    && seitenAbstandNachRechtsMesswerte[0] != Float.NaN) {
                wechselZustand(1, -motorAntrieb.getTachoCount());
            }
            if (seitenAbstandNachRechtsMesswerte[0] > 0.15
                    | seitenAbstandNachRechtsMesswerte[0] == Float.NaN) {
                wechselZustand(2, -motorAntrieb.getTachoCount());

            }

        }
        if (zustandDesEinparkens == 1) {
            seitenAbstandNachRechts.fetchSample(
                    seitenAbstandNachRechtsMesswerte, 0);

            if (seitenAbstandNachRechtsMesswerte[0] > 0.15
                    | seitenAbstandNachRechtsMesswerte[0] == Float.NaN) {
                wechselZustand(2, -motorAntrieb.getTachoCount());

            }
        }
        if (zustandDesEinparkens == 2) {
            seitenAbstandNachRechts.fetchSample(
                    seitenAbstandNachRechtsMesswerte, 0);

            if (seitenAbstandNachRechtsMesswerte[0] < 0.15
                    && seitenAbstandNachRechtsMesswerte[0] != Float.NaN) {
                parklueckenEnde = -motorAntrieb.getTachoCount();

                if ((parklueckenEnde - parklueckenBeginn) > 540) {
                    wechselZustand(3, -motorAntrieb.getTachoCount());
                } else {
                    wechselZustand(1, -motorAntrieb.getTachoCount());
                }
            }
        }
        if (zustandDesEinparkens == 3) {
            seitenAbstandNachRechts.fetchSample(
                    seitenAbstandNachRechtsMesswerte, 0);
            motorAntrieb.setAcceleration(6000);
            abstandNachRechtsZuBeginnDesEinparkens = seitenAbstandNachRechtsMesswerte[0];

        }
    }
```

Abbildung 27: Realisierung des Zustandsübergangsdiagramms

Jede der „if“- Bedingungen, in denen der „zustandDesEinparkens“ überprüft wird, entspricht einem Zustand aus dem Zustandsübergangsdiagramms in Abbildung 19. Eingerahmt ist dieser Zustandsautomat ebenfalls wieder von einer „while“-Schleife, die erst verlassen wird, wenn der Zustand drei entspricht. Ist dies der Fall, begibt sich das Programm zurück in die main-Methode aus Abbildung 25, der Motor wird gestoppt und die Methode „parkeEin“ wird aufgerufen. Diese ist in Abbildung 28 dargestellt.

```
private static void parkeEin() {

    berechneEinparkLaenge();

    motorAntrieb.setSpeed(100);
    motorAntrieb.setAcceleration(150);

    int gradFuerMotor = (int) ((streckeBiszurHaelfteInDerParkluecke / streckeProReifenUmdrehung) * 360);

    motorAntrieb.rotate(-450);
    motorLenkung.rotate(130);
    motorAntrieb.rotate(gradFuerMotor);

    motorLenkung.rotate(-260);
    motorAntrieb.rotate(gradFuerMotor);

    motorLenkung.rotate(130);

    Button.LEDPattern(4);
    Sound.twoBeeps();

    Delay.msDelay(2000);

    System.exit(0);
}
```

Abbildung 28: Methode zum Einparken

Zu Beginn dieser Methode wird eine weitere Methode aufgerufen. In „berechneEinparkLaenge“ wird ermittelt, abhängig vom Abstand zum nebenstehenden Fahrzeug, welche Strecke das Auto fahren muss, bis umgelenkt werden muss. Dies ist vergleichbar mit den Strecken S_1 und S_2 aus der mathematischen Simulation, gezeigt in Kapitel 4.1.2. Mit dieser Methode wird sichergestellt, dass das Auto unabhängig vom Abstand korrekt einparkt. In Abbildung 29 ist der Code dargestellt.

```
private static void berechneEinparkLaenge() {

    streckeBiszurHaelfteInDerParkluecke = (Math
            .asin(((breiteDesAutos + abstandNachRechtsZuBeginnDesEinparkens - abstandSensorRechtsZumReifenRechts) / 2)
                    * (Math.cos(Math.toRadians(45)) / laengeDesAutos) - 1) + (Math.PI / 2))
            * laengeDesAutos / Math.cos(Math.toRadians(45));
}
```

Abbildung 29: Berechnung des Weges zum Einparken

Nach dieser Berechnung geht die Ausführung des Codes in der Methode „parkeEin“ weiter. Zu Beginn werden erneut die Geschwindigkeit und die Beschleunigung des Motors auf einen geringen Wert gesetzt. Anschließend wird aus dem gerade berechneten Wert eine Gradzahl ermittelt, die verwendet wird, um den Motor für die richtige Anzahl rotieren zu lassen. Nachdem der Einparkvorgang abgeschlossen ist, erhält der Fahrer erneut eine visuelle und akustische Bestätigung. Danach schaltet sich das Programm ab. Das Auto steht nun in einer Parklücke.

5.2 Umsetzung des Testkonzepts

In diesem Abschnitt wird das Ergebnis der Durchführung des erarbeiteten Testkonzeptes aus Kapitel 4.3 vorgestellt. Der vollständige Durchlauf im Video ist über folgenden QR-Code, Abbildung 30, erreichbar.

Abbildung 30: QR Code zum Videoabruf des Testdurchlaufs

Außerdem befinden sich ausführliche Bilder im Anhang dieser Arbeit.

Im ersten Durchlauf sollte nicht eingeparkt werden, das Auto fährt einfach gerade aus. Abbildung 31 zeigt ein Bild der Durchführung.

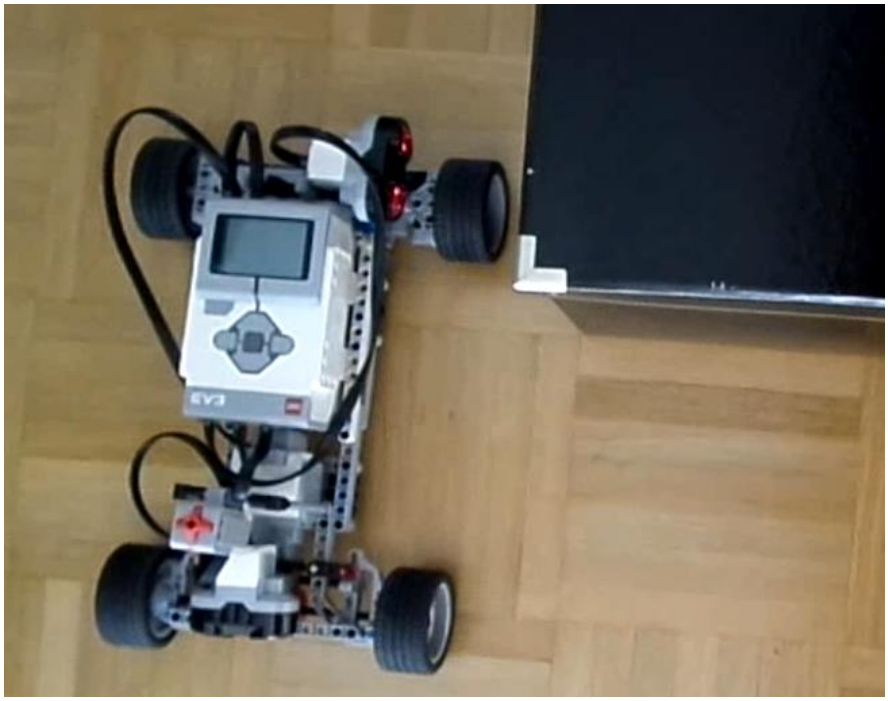

Abbildung 31: Testdurchlauf 1: Auto ignoriert Parklücke

Im zweiten Durchlauf wird der Schalter zu Beginn aktiviert. Während die erste Parklücke ignoriert werden soll, parkt das Auto in die zweite Parklücke ein. Abbildung 32 und 33 zeigen, wie das Auto den Einparkvorgang startet und das Ergebnis in der Parklücke.

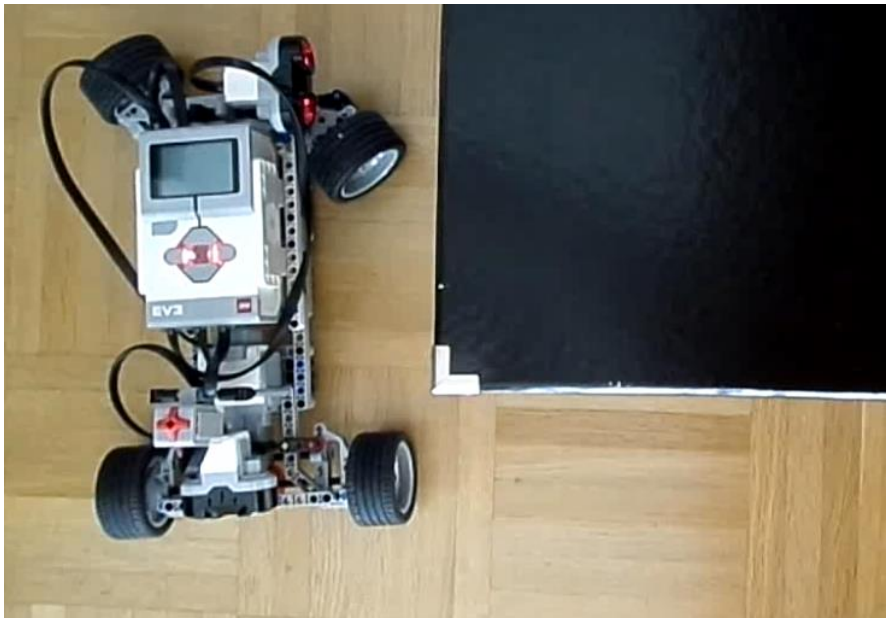

Abbildung 32: Testdurchlauf 2: Auto startet Einparkvorgang

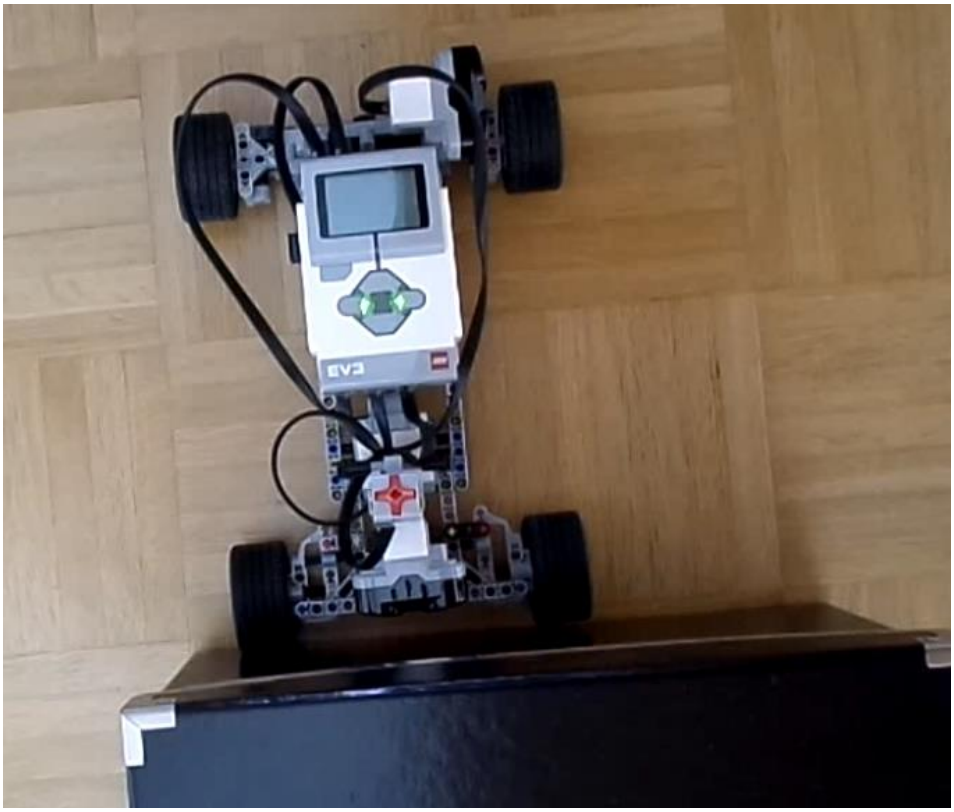

Abbildung 33: Testdurchlauf 2: Auto steht in Parklücke

5.3 Soll-/Ist-Analyse

Für die Soll-/Ist-Analyse wird die Tabelle 1 aus Kapitel 3 erneut aufgegriffen und die Anforderungen werden mit den Ergebnissen verglichen.

Muss - Anforderungen	Kann – Anforderungen
Auto fährt	Auto parkt auf Wunsch ein
Auto hat eine Lenkung	Auto erkennt Parklücken seitlich, quer
Auto erkennt Parklücken seitlich	Auto parkt automatisch quer rückwärts ein
Auto parkt automatisch seitlich rückwärts ein	
Nachvollziehbare Dokumentation	

Tabelle 5: Umsetzung der Anforderungen

Die Muss – Anforderungen sind alle erfüllt. Das Auto fährt vorwärts, hat eine Lenkung, Parklücken werden erkannt und auf die passende Größe hin untersucht. Ist die Lücke passend, parkt das Auto automatisch seitlich rückwärts ein. Sowohl der Scilab – als auch der Java – Code sind dokumentiert. Zusätzlich ist in dieser vorliegenden Arbeit das Vorgehen festgehalten.

Bei den Kann – Anforderungen ist erfüllt, dass das Auto erst auf Wunsch einparkt. Hierfür wurde ein Berührungssensor installiert, der erst betätigt werden muss, bevor das Auto seinen Einparkvorgang startet. Die weiteren beiden Anforderungen, die sich auf das quer, rückwärts einparken beziehen, konnten nicht umgesetzt werden.

5.4 Probleme

Es lässt sich bemerken, dass die realen Testdurchläufe durchaus von der Theorie abweichen. Jedoch ist dies auf das verwendete Material zur Realisierung zurückzuführen. Das Lego Mindstorms System eignet sich sehr gut zum Aufbauen von Modellen, falls schnell etwas überprüft werden soll, jedoch ist es nicht ideal für exakte Realisierungen. So war zu bemerken, dass die Teile nicht fest in ihrer Position zu halten waren.
Auch ist problematisch, dass mit der Lenkung keine festen Winkel einstellbar waren. Der Motor für die Lenkung kann sich nur um eine gewisse Gradzahl drehen. Diese entspricht jedoch nicht dem Winkel, die die Reifen anschließend annehmen.
Ein weiteres Problem ist die Menge an Sensoren. Für einen vollständig sicheren autonomen Einparkvorgang würden weitere benötigt werden.

6. Fazit und Ausblick

In dieser Studienarbeit wurde ein funktionierender Algorithmus zunächst mathematisch hergeleitet und anschließend programmiert und getestet, mit dem ein beliebiges Auto in eine Parklücke autonom einparken kann.

Zu Beginn wurde auf theoretische und technische Grundlagen eingegangen. So wurde erklärt, was eine Einparkhilfe ist, vgl. Kapitel 2.1. und was diese von autonomen Einparken abhebt, vgl. Kapitel 2.1.2. In den technischen Grundlagen wurde zunächst auf das Lego Mindstorms System eingegangen, vgl. Kapitel 2.2.1. Anschließend wurde die Programmiersprache Java, Kapitel 2.2.2 und das Framework LeJOS, Kapitel 2.2.3 eingegangen. Zum Abschluss der Grundlagen wurde das Entwickeln mit Eclipse als Entwicklungsumgebung beschrieben. In Kapitel 3 wurden die Anforderungen an das Projekt genannt. Anschließend wurden im Konzeptteil die mathematischen Berechnungen beschrieben, vgl. Kapitel 4.1. Hier wurden Gleichungen für den Verlauf eines Einparkvorgangs aufgestellt, Kapitel 4.1.1, und anschließen simuliert, Kapitel 4.1.2 und 4.1.3.
Im zweiten Abschnitt des Konzeptteils wurden die Konstruktion des Autos und das Ergebnis des Autobaus gezeigt. Zum Abschluss wurde in Kapitel 4.3 das Testkonzept zur Erfolgskontrolle vorgestellt.
Kapitel 5 zeigt die Programmierung des Algorithmus in Kapitel 5.1 und geht dann auf das Umsetzen des Testkonzepts ein. In einer Soll-/Ist-Analyse wird in Kapitel 5.3 überprüft, inwiefern die Anforderungen aus Kapitel 3 umgesetzt werden konnten.

Für diese modellhafte Umsetzung hat sich der Algorithmus bewährt und könnte theoretisch auch auf ein reales Automobil übertragen werden, jedoch müssten die in Kapitel 5.4 angesprochenen Probleme beseitigt werden.

IV. Literaturverzeichnis

[1] ADAC, „adac.de," 12 April 2016. [Online]. Available: https://www.adac.de/infotestrat/technik-und-zubehoer/fahrerassistenzsysteme/grundlagen/default.aspx?ComponentId=236162&SourcePageId=227535.

[2] A. Fahren, „Autonomes Fahren," [Online]. Available: http://www.autonomes-fahren.de/. [Zugriff am 12 April 2016].

[3] Autobild, „Autobild," 10 Februar 2016. [Online]. Available: http://www.autobild.de/artikel/autonomes-fahren-was-moeglich-ist-und-was-erlaubt-7191393.html. [Zugriff am 12 April 2016].

[4] Autoscout24, „Elektronische Einparkhilfe," Autoscout24, [Online]. Available: http://themen.autoscout24.de/extras/elektronische-einparkhilfe-funktionsweise. [Zugriff am 12 April 2016].

[5] H. Wallentowitz und K. Reif, Handbuch Kraftfahrzeugelektronik: Grundlagen - Komponenten - Systeme - Anwendungen, Vieweg+Teubner Verlag, 2010.

[6] Expertentesten, „Expertentesten.de," 11 April 2016. [Online]. Available: http://www.expertentesten.de/elektronik/einparkhilfe-test/. [Zugriff am 12 April 2016].

[7] U. Nations, „Report of the sixty-eighth session of the Working Party on Road Traffic Safety," UN, Genéva, 2014.

[8] Autogazette, „Zukunftsforscher Alexander Mankowsky," 3 April 2015. [Online]. Available: http://www.autogazette.de/daimler/mankowsky/autonom/ein-autonomes-auto-ist-weder-ethisch-noch-unethisch-512192.html. [Zugriff am 12 April 2016].

[9] n-tv, „Autonomes Parken in naher Zukunft," 7 August 2015. [Online]. Available: http://www.n-tv.de/auto/Handy-oder-Fernbedienung-parken-Auto-ein-article15674896.html. [Zugriff am 12 April 2016].

[10] „Ziel und Herausforderung der Roberta® - Initiative," Fraunhofer IAIS, [Online]. Available: http://roberta-home.de/de/konzept. [Zugriff am 12 April 2016].

[11] LEGO, „lego.com," 12 April 2016. [Online]. Available: http://shop.lego.com/en-US/EV3-Intelligent-Brick-45500. [Zugriff am 12 April 2016].

[12] LEGO, „History of LEGO Robotics.," LEGO, [Online]. Available: http://www.lego.com/de-de/mindstorms/history. [Zugriff am 12 April 2016].

[13] LEGO, „support," [Online]. Available: http://www.lego.com/en-us/mindstorms/support. [Zugriff am 12 April 2016].

[14] C. Ullenboom, „Java ist auch eine Insel," in *Java ist auch eine Insel*, Rheinwerk Verlag, 2011, p. Abschnitt 1.3.9.

[15] C. Ullenboom, „Java ist auch eine Insel," in *Java ist auch eine Insel*, Rheinwerk Verlag, 2011, p. Abschnitt 1.2.

[16] C. Ullenboom, „Java ist auch eine Insel," in *Java ist auch eine Insel*, Rheinwerk Verlag,, 2011, p. Abschnitt 1.2.1.

[17] C. Ullenboom, „Java ist auch eine Insel," in *Java ist auch eine Insel*, Rheinwerk Verlag, 2011, p. Abschnitt 1.2.1 & 1.2.2.

[18] C. Ullenboom, „Java ist auch eine Insel," in *Java ist auch eine Insel*, Rheinwerk Verlag, 2011, p. Abschnitt 1.2.3.

[19] C. Ullenboom, „Java ist auch eine Insel," in *Java ist auch eine Insel*, Rheinwerk Verlag, 2011, p. Abschnitt 2.7.

[20] tutego.de, „tutego.de Java Bibliotheken," [Online]. Available: http://www.tutego.de/java/java-open-source.htm. [Zugriff am 04 09 2015].

[21] lejos. [Online]. Available: https://sourceforge.net/p/lejos/wiki/Home/. [Zugriff am 12 April 2016].

[22] F. Deitelhoff, „LEGO Mindstorms EV3 Education – Offizielle Software-Blöcke," 31 Dezember 2015. [Online]. Available: http://www.fabiandeitelhoff.de/2015/12/lego-mindstorms-ev3-education-offizielle-software-bloecke/. [Zugriff am 12 April 2016].

[23] E. Burnette und J. Staudemeyer, Eclipse IDE kurz & gut, O'Reilly Verlag, 2013.

[24] U. z. Köln, „Die main-Methode,“ 3 Februar 2015. [Online]. Available: http://spinfo.phil-fak.uni-koeln.de/spinfo-java-mainmethode.html?&L=0. [Zugriff am 12 April 2016].

[25] Scilab, „About,“ 2015. [Online]. Available: http://www.scilab.org/. [Zugriff am 12 April 2016].

[26] Scilab, „Help - dae,“ 05 Oktober 2011. [Online]. Available: https://help.scilab.org/docs/5.3.3/en_US/dae.html. [Zugriff am 12 April 2016].

[27] LEGO, „Baue einen Roboter,“ 2016. [Online]. Available: http://www.lego.com/de-de/mindstorms/build-a-robot. [Zugriff am 12 April 2016].

V. Anhang