Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Torben Becker

Identifikation des Lenksystems in autonomen Fahrzeugen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Torben Becker

Identifikation des Lenksystems in autonomen Fahrzeugen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Stephan Pareigis
Zweitgutachter: Prof. Dr. Zhen Ru Dai

Abgegeben am 31. August 2011

Torben Becker

Thema der Bachelorarbeit

Identifikation des Lenksystems in autonomen Fahrzeugen

Stichworte

Nullstellung Lenkung autonom Fahrzeug T.O.R.C.S.

Kurzzusammenfassung

Durch verschiedene äußere Einflüsse auf die Lenkung kann sich diese verstellen und in einem autonomen Fahrzeug dazu führen, dass bei einer eigentlichen Nullstellung der Lenkung eine seitlich versetzte Idealfahrlinie gefahren wird, die die Fahrbahnmarkierung überschreitet. Diese Arbeit behandelt mehrere Algorithmen, die in der Lage sind, selbstständig die neue Nullstelle der Lenkung während der Fahrt zu bestimmen und einzustellen. Dafür wird in T.O.R.C.S. (The Open Racing Car Simulator) eine Simulation der Algorithmen entwickelt und bewertet mit anschließender Portierung der Algorithmen auf die Software Architektur des FAUST Projektes.

Title of the paper

Identification of a steering system in autonomous vehicles

Keywords

zero position steering autonomous vehicle T.O.R.C.S.

Abstract

Caused by various external influences on the steering it is possible that it dissimulates. In an autonomous vehicle this may leed to driving a off-centered ideal driving line which exceeds the lane marking, even when the steering is in the actual zero position. This work addresses several algorithms that are able to independently determine and adjust the new zero position of the steering while driving. A simulation of the algorithms is developed and evaluated in T.O.R.C.S. (The Open Racing Car Simulator). Then the algorithms are ported to the software architecture of the FAUST project.

Inhaltsverzeichnis

1 Einleitung

Es ist normal, dass sich bei Fahrzeugen nach einer gewissen Fahrleistung oder kleineren Zusammenstößen mit einem Bordstein oder Kanten die Lenkspur ein wenig verstellt. In einem normalen Fahrzeug, das von einem Menschen gelenkt wird, spielt das keine große Rolle, da der Mensch selbst intuitiv das Lenkrad für sich selbst nachjustiert und so problemlos weiterfährt. Für autonome Fahrzeuge und besonders für autonome Modellfahrzeuge, wo bereits durch Vibration des Fahrzeuggestells kleine Änderung an dem Lenksystem vorfallen können, ist diese Verstellung der Lenkspur allerdings ein großes Problem.

Besonders bei den Modellfahrzeugen, wie sie in dem Projekt FAUST (FAUST) entwickelt werden, ist die Lenkung ein sehr sensibles System und bereits leichte Zusammenstöße können eine enorme Änderung im Lenkverhalten nach sich ziehen. Dabei wird im Projekt FAUST sowohl von Teams aus Studenten als auch von Bachelorabsolventen bzw. Masterabsolventen versucht, Algorithmen zu entwickeln, die besonders robust gegen Einflüsse von außen sind und ein möglichst stabiles Verhalten besitzen. Allerdings ist eine Verstellung der Lenkung für den derzeitigen Spurführungsalgorithmus Pure Pursuit keine einfache Aufgabe und solch eine Verstellung kann diesen Algorithmus sogar komplett unbrauchbar machen, da dieser nicht mehr die ideale Fahrlinie innerhalb einer Spur halten kann und regelmäßig die Fahrbahnbegrenzung überschreitet.

Passiert dieser Fehler in Wettbewerben wie dem Carolo-Cup (Carolo-Cup) kann das regelmäßige Überschreiten der Fahrbahnbegrenzung sehr viele Punkte kosten. Der Carolo-Cup wird jährlich im Frühling von der Technischen Universität Braunschweig veranstaltet. In diesem Wettbewerb messen sich mehrere Universitäten und Fachhochschulen gegeneinander in verschiedenen Disziplinen (Carolo-Cup, 2011) mit autonomen Modellfahrzeugen im Maßstab 1:10. Dabei kommt es nicht nur darauf an, ein Fahrzeug zu besitzen, das perfekt einen Rundkurs durchfahren kann ohne die Fahrbahnbegrenzung zu überschreiten, sondern auch, wie gut das Fahrzeug in einem Rundkurs fährt, wenn Hindernisse aufgestellt werden oder auf das selbstständige Einparken in eine Parklücke. Dabei stehen verschiedene Größen zur Auswahl und das Fahrzeug muss zunächst eine Parklücke finden, die groß genug ist und dann rückwärts-seitwärts einparken. Allerdings sollen die Teams nicht nur die Fahrzeuge gegeneinander antreten lassen, sondern auch einen Vortrag über die Entwicklungskosten und -aufwand, über die verwendeten Algorithmen und mögliche Fallback Routinen halten.

Diese Arbeit beschäftigt sich mit dem Thema Algorithmen zu entwickeln, die versuchen, die Lenkung nach Detektion einer Lenkspurverstellung eines autonomen Fahrzeuges neuzujustieren. Dabei werden verschiedene Ansätze untersucht und miteinander verglichen. Dabei sind die beiden großen Bereiche der möglichen Algorithmen einmal nach dem Prinzip Trail and Error bzw. durch Messungen und dementsprechender Anpassung. Es wird zunächst versucht den theoretischen Hintergrund zu beleuchten und die mathematischen Ansätze zu erklären, als nächstes folgt eine Implementierung mit Beleuchtung sämtlicher Details und dem Ablauf der Algorithmen sowie ein ausführlicher Test der Algorithmen.

Zur Entwicklung der Algorithmen kommt das Computer Simulationsspiel T.O.R.C.S. (T.O.R.C.S.) zum Einsatz, um der Beschädigung der Fahrzeuge aus dem Projekt FAUST entgegenzuwirken. Dabei müssen bestimmte Funktionen selbst implementiert werden, da T.O.R.C.S. diese nicht zur Verfügung stellt. Später sollen die unter T.O.R.C.S. entwickelten und getesteten Algorithmen auf die Software Architektur des Projektes FAUST portiert und erneut getestet werden, um ihre Funktionalität auch in einer realen Umgebung zu verifizieren.

2 T.O.R.C.S.

T.O.R.C.S. (T.O.R.C.S.) ist ein Open Source Rennsimulationsspiel und steht für The Open Racing Car Simulator, das unter der GNU GPL veröffentlicht wird. Es läuft auf vielen Betriebssystemen (Windows, Linux, MacOS, FreeBSD) und wurde mittels C++ programmiert. T.O.R.C.S. dient nicht nur dem persönlichen Vergnügen, sondern arbeitet auch mit sogenannten Robotern. Zudem bietet T.O.R.C.S. ein realistisches Fahrverhalten der Fahrzeuge. In dieser Arbeit wird die Version 1.3.1 verwendet, die letzte veröffentlichte stabile Version zu diesem Zeitpunkt.

Abbildung 1: T.O.R.C.S. im laufenden Betrieb

2.1 Das Fahrzeug und die Umgebung

Fahrzeuge und deren Umgebung werden in T.O.R.C.S. durch die Struktur *tCarCtrl* beschrieben. Diese bietet in T.O.R.C.S. zahlreiche Daten über den Zustand des Fahrzeuges, z.B. den Motor, Schäden, die Reifen oder über den Fahrer. Darüber hinaus kann man über die Struktur *tCarCtrl* Entfernungen des Fahrzeuges zum Ziel oder zum Start abfragen, allerdings mit der Einschränkung, das diese nur segmentgenau ist. Auch bietet die Struktur die Möglichkeit die Entfernungen zur Mittellinie zurückzugeben oder zum rechten bzw. linken Fahrbahnende respektive der Leitplanke der Rennbahn. Dabei gibt das Vorzeichen dieser Werte die Position des Fahrzeuges an, auf welcher Hälfte sich das Fahrzeug befindet. Sind

die Werte positiv, befindet sich das Fahrzeug auf der linken Hälfte. Auf der rechten Hälfte befindet sich das Fahrzeug, wenn die Werte negativ sind. Weiterhin ist in *tCarCtrl* die Struktur *tTrackSeg* enthalten, die viele Informationen über die Strecke und die Beschaffenheit der Strecke enthalten, ob man z.B. auf einer Teerstraße fährt oder auf Sand, ob man sich auf der vorgegebenen Strecke befindet oder im Grünstreifen. Ebenfalls kann die Struktur darüber Auskunft geben, wie der Kurvenradius des aktuellen Segments ist und ob die Kurve eine Anhöhung an den Rändern besitzt. Die Struktur *tTrackSeg* besitzt eine Verkettung aller Segmente mittels doppelt verketteter Liste.

Abbildung 2: Das Testfahrzeug

Das Testfahrzeug hat die Bezeichnung *cg-nascar-rwd* und ist ein fiktives Fahrzeug. Es ist keinem realen Fahrzeug nachempfunden. Es besitzt eine Länge von 5 Metern, eine Breite von 2 Metern und eine Höhe von 1.1 Metern. Der Lenkwinkel beträgt -45 Grad bis +45 Grad und die Schaltzeit zwischen zwei Gängen hat eine Dauer von 250 Millisekunden.

2.2 Die Lenkung

Die Lenkung in T.O.R.C.S. wird in Prozent des maximalen Lenkwinkels angegeben. Dieser Wert kann von Fahrzeug zu Fahrzeug unterschiedlich sein. Allerdings bietet T.O.R.C.S. keine Möglichkeit, diesen Lenkwinkel in Grad zu messen, wenn man z.B. einen Wert von 50 Prozent vorgibt. Allerdings wird aufgrund der Abbildung 3 angenommen, dass sich die Verteilung des Lenkwinkels auf den Prozentbereich sehr gleichmäßig darstellt. Das bedeutet, dass das Verhältnis von Einstellung des Lenkwinkels, die sowohl auf den FAUST Fahrzeugen als auch in T.O.R.C.S. in Prozent zwischen null und eins angegeben werden, zu realem Lenkwinkel linear verläuft und dies sowohl im positivem wie auch im negativen Lenkwinkel so hält. Die Abweichung im Diagramm wurde in Schritten von 0.1 Prozent gemessen im Intervall von -60 bis +60.

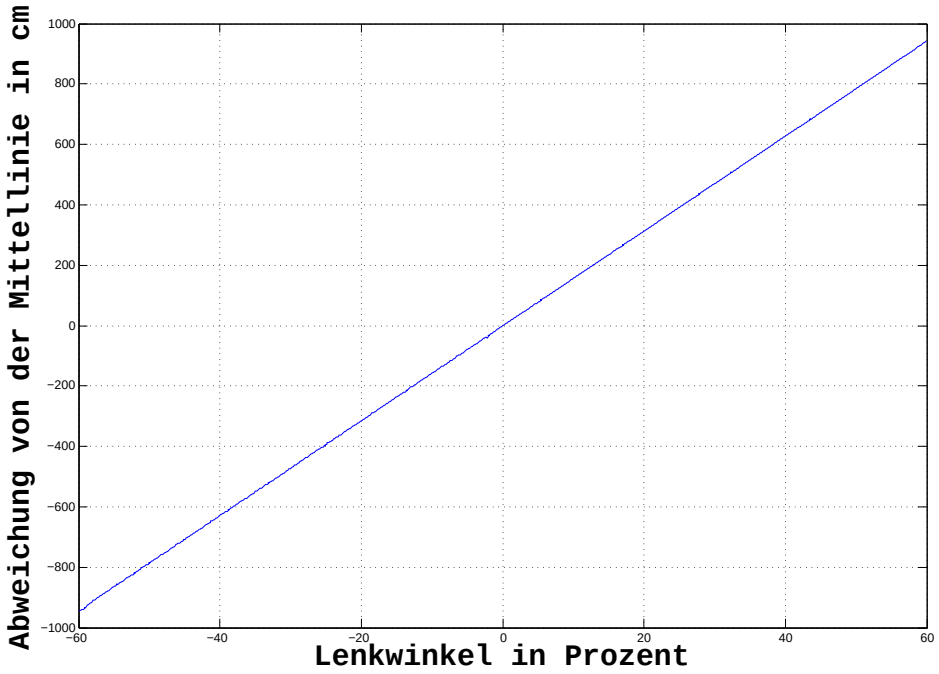

Abbildung 3: Lenkwinkel in Prozent und dazu gemessene Abweichung von der Mittellinie

Dagegen haben die Fahrzeuge des FAUST Projektes sowie reale Fahrzeuge nicht unbedingt einen linearen Verlauf. Bei diesen Fahrzeugen sind im Verlauf mehrere Umbrüche möglich. Diese Umbrüche sorgen dafür, dass das Fahrzeug im entsprechenden Intervall keine Änderung am Lenkwinkel vornehmen kann.

2.3 Geschwindigkeit, Schaltung und Bremsung

Man kennt von Rennsimulationen am Computer und Konsolen, sofern diese nicht über ein Lenkungssystem verfügen, dass man mittels den Pfeiltasten beschleunigen bzw. abbremsen kann. Dabei fährt man sehr häufig mit Hilfe eines Automatikgetriebes. In T.O.R.C.S. steht dem physikalischen Spieler zwar ebenfalls ein Automatikgetriebe zur Seite, sobald man aber einen Roboter (2.5) entwickelt, steht dieses Automatikgetriebe nicht mehr zur Verfügung. Der Entwickler muss dem Roboter beibringen, wie er zu schalten hat, in dem er das mittels Reinforcement Learning Algorithmen erlernt oder dem Roboter einprogrammiert ab einer gewissen Drehzahl einen Gang hochzuschalten. Darüber hinaus muss der Entwickler dem Roboter auch noch sagen, wie schnell das Fahrzeug fahren darf. Dieses Problem kann auch mittels Reinforcement Learning Algorithmen **Hr.Pareignis nach Projekt von Jan fragen** gelöst werden. Weitere Alternativen wären ein Regelungssystem oder Fuzzy Logic.

2.4 Strecke (Track)

In T.O.R.C.S. besteht eine Strecke (Track) aus Segmenten. Jedes Segment ist einem bestimmten Typ zugeordnet: Rechtskurve, Linkskurve oder geradeaus. Die Segmente können eine definierte Länge haben, die in Metern angegeben ist. Dabei ist es nicht wichtig, dass aufeinander folgende Segmente die gleiche Länge haben. Häufig sind die Segmente in den Standardstrecken in T.O.R.C.S. sehr kurz gewählt (ca. 4 Meter). Es können allerdings auch Abschnitte vorkommen, wo die Segmente noch kleiner unterteilt sind, z.B. in langen Geraden oder großen Kurven. Ist ein Segment eine gerade Strecke hat es eine Länge und eine Breite, Kurven haben zusätzlich noch einen Radius. Alle Angaben werden ausgehend von der Mitte des Segments angegeben, wobei alle Segmente tangential miteinander verbunden sind. Durch diese Art des Verbundes ergibt sich eine geglättete Mittellinie.

Abbildung 4: Die Teststrecke

Die Teststrecke nennt sich *E-Track 5* und ist aus der Kategorie *Oval Tracks*. Sie besitzt eine Länge von 1621.73 Metern und hat eine Spurbreite inklusive des Grünstreifens von 20 Metern. Es gibt sechs Kurven und zwei gerade Strecken. Die Strecke ist von *E.Espie* entwickelt und befindet sich in der dritten Version. Außerdem verfügt die Strecke an bestimmten Stellen über Anhöhungen an der Außenkante der Strecke.

2.5 Roboter

Roboter in T.O.R.C.S. sind künstliche Intelligenzen. Diese werden in C++ programmiert und steuern das Fahrzeug anstelle eines Spielers. Dabei gilt es im Allgemeinen bestimmte Hürden zu überwinden und letztendlich die Roboter gegeneinander antreten zu lassen. Im Rahmen des FAUST Projekts werden mit Hilfe der Roboter die Spurerkennung verbessert sowie verschiedene lernende Algorithmen entwickelt.

Anders als die Fahrzeuge in FAUST verfügen die Roboter über keine Kamerasicht. Wie schon in den Kapitel 2.1, 2.2, 2.3 und 2.4 erwähnt, stellt T.O.R.C.S. dem Roboter diver-

se Informationen zur Verfügung, woran sich der Roboter orientieren muss. Darüber hinaus müssen dem Roboter auch einfache Fähigkeiten wie richtiges Bremsen oder Schalten implementiert werden sowie ein System zum Halten einer bestimmten Geschwindigkeit. (Wymann, 2005a)

3 Grundlagen

3.1 Pure Pursuit

Der Algorithmus Pure Pursuit (Coulter, 1992) zur Verfolgung eines Pfades oder Weges ist dem menschlichem Verhalten während des Autofahrens sehr ähnlich. Ein Mensch fixiert in einiger Entfernung vor sich selbst einen Punkt und fährt dann auf diesen zu, indem er das Lenkrad kontinuierlich um einen gewissen Grad gedreht hält. Dabei fahren beide, sofern es sich nicht um eine gerade Strecke handelt, einen gewissen Kreis. Der Algorithmus versucht für diesen Kreis die Krümmung zu berechnen, die durch die aktuelle Position des Fahrzeuges und des Zielpunktes führt.

Der Zielpunkt wird im Pure Pursuit Algorithmus als *goal point* bezeichnet, wobei dieser *goal point* nur ein Zielpunkt auf dem zurückzulegenden Pfad darstellt. Die Entfernung von der aktuellen Fahrzeugposition bis zum *goal point* heißt *lookahead distance* und ist gleichzeitig die Kreissehne. Die wesentliche Aufgabe des Algorithmus ist die Bestimmung des *goal points* und die Berechnung der Krümmung des Kreises, um diesen Punkt von der aktuellen Fahrzeugposition zu erreichen.

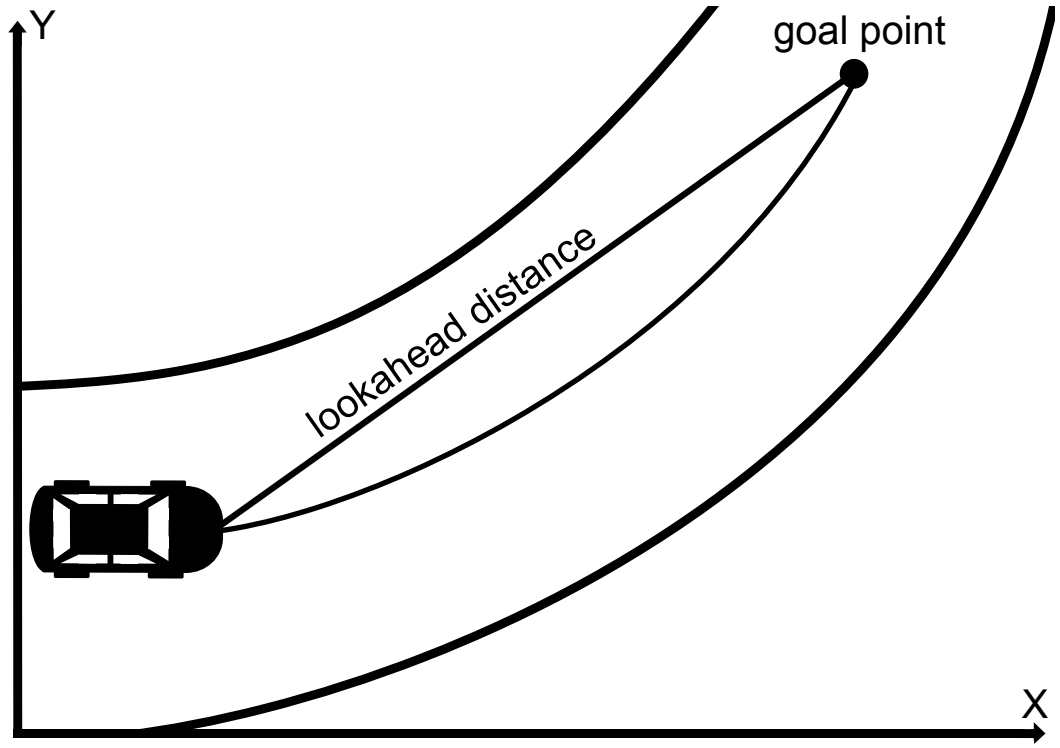

Abbildung 5: Darstellung des Pure Pursuit Algorithmus

3.2 Lenksystem in T.O.R.C.S.

Während der Pure Pursuit Algorithmus sich einen Zielpunkt in der Ferne sucht und eine möglichst optimale Krümmung für eine Durchfahrt errechnet, basiert das sehr einfache Lenksystem in T.O.R.C.S. nur auf der aktuellen Position des Fahrzeuges. Dabei versucht das Fahrzeug möglichst auf der Mittellinie zu fahren und wenn es von dieser abkommt, versucht der Algorithmus, möglichst schnell wieder zur Mittellinie zu gelangen.

Um dies zu erreichen, wird zunächst ein Eingangslenkwinkel berechnet, indem von dem tangentialen Winkel des aktuellen Streckensegments der Winkel abgezogen wird, den das Fahrzeug zum aktuellen Streckensegment hat. Dann wird dieser Wert in einen Bereich von $-\pi$ bis π normiert und das Verhältnis von Entfernung des Fahrzeuges zur Mittellinie und Fahrbahnbreite im aktuellen Streckensegment gebildet und von dem Eingangslenkwinkel subtrahiert.

3.3 Verhalten der Lenkung mit Offset

Wenn die Lenkung von einem Regelungssystem gesteuert wird um die Spur zu halten, so würde ein Offset, der zum aktuell eingestellten Lenkwinkel hinzuaddiert wird, das System zum Schwingen bringen und es müssten diese Schwingungen beseitigt werden. Dies würde bedeuten, dass das Fahrzeug sich immer wieder in der Mitte befinden müsste, um dann eine seitliche Verschiebung innerhalb eines bestimmten Intervalls zu messen oder ein etwas komplexeres System mithilfe eines bisektionalen Algorithmus, wo das Problem darin bestünde, den Wert für die Korrektur nicht zu schnell und stark zu ändern.

Wird aber anstelle eines Regelungssystems nun Pure Pursuit oder ein sehr einfaches System zum Halten der Spur, welches in dem T.O.R.C.S. Robot Manual erläutert wird, genutzt, so löst der Offset keine Schwingungen aus. Der Offset verursacht lediglich eine versetzte Spur und fährt somit eine seitlich versetzte Ideallinie, die auch die komplette Zeit gehalten werden kann. Diese Versetzung der Ideallinie ist mit einem bisektionalem Algorithmus sehr einfach zu detektieren und zu korrigieren. Außerdem kann auch ein Messsystem für diese Aufgabe eingesetzt werden, das verschiedene seitliche Abweichungen bei einer gut eingestellten Lenkung misst und dann einen Vergleich zwischen Abweichung und gespeicherten Werten durchführt und dementsprechend korrigiert.

Sowohl Pure Pursuit als auch das einfache Lenksystem aus dem T.O.R.C.S. Robot Manual suchen sich auf der Strecke einen Zielpunkt und berechnen, welchen Lenkwinkel das Fahrzeug haben muss, damit dieser Punkt erreicht werden kann. Bei Pure Pursuit liegt dieser Zielpunkt in einer gewissen Entfernung, während er sich bei dem einfachen Lenksystem in dem gleichen Segment befindet. Wird nun ein Offset zum berechneten Lenkwinkel der Algorithmen hinzuaddiert, so wird der eigentlich angesteuerte Zielpunkt nicht geändert. Es

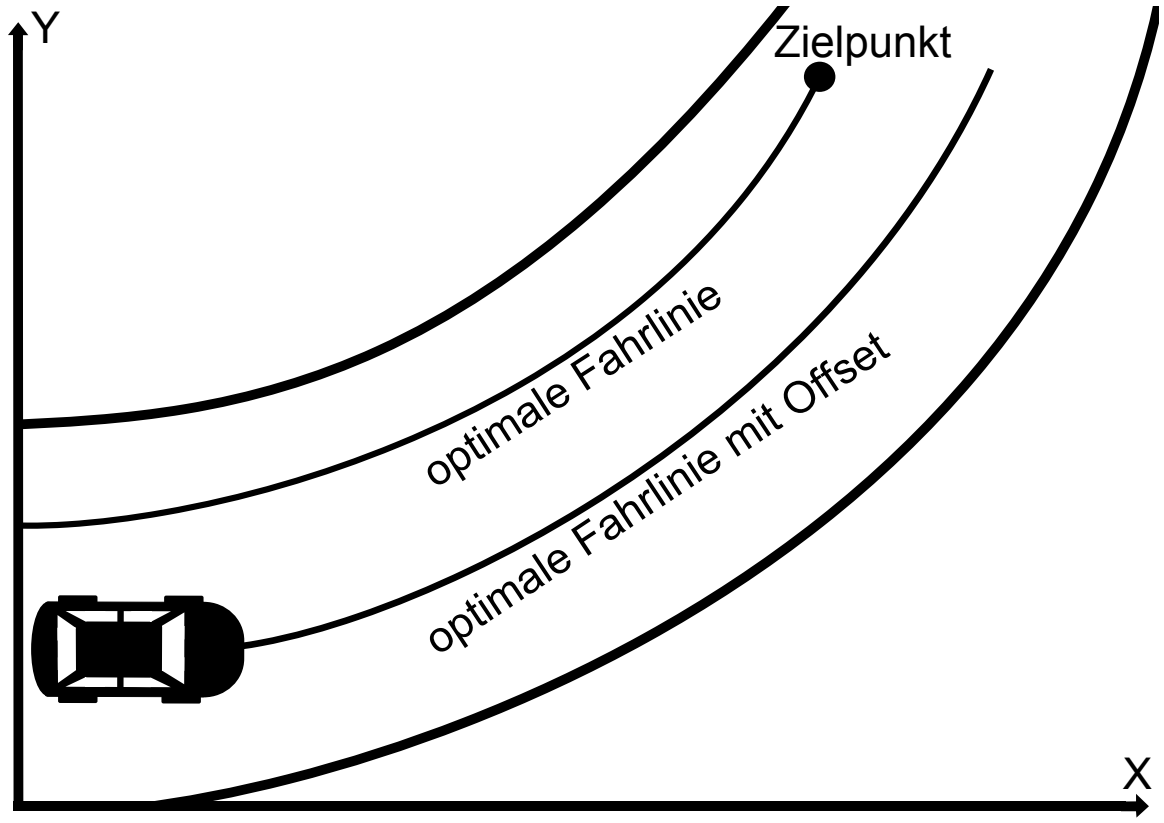

Abbildung 6: Grafische Darstellung des Verhaltens einer Lenkung mit Offset

wird der Lenkwinkel verändert und somit die Krümmung des gefahrenen Teilkreises beeinflusst und dementsprechend eine zur Ideallinie versetzte Fahrlinie gefahren. Dabei wird die versetzte Fahrlinie durchgehend gehalten und es treten keine Schwingungen auf.

4 Beschreibung der Algorithmen

4.1 Bisektionaler Algorithmus

Ein einfacher Algorithmus stellt das bisektionale Verfahren dar. Es wird theoretisch eine der drei Linien einer Straße beobachtet und gemessen, wie weit sich das Fahrzeug von dieser Linie entfernt. Überschreitet die gemessene Distanz eine selbst definierte Entfernung, so fängt der Algorithmus an, den Lenkwinkel um einen bestimmten Wert zu verändern, bis sich dieser wieder im Toleranzbereich befindet.

Da es in T.O.R.C.S. keine Leitlinien wie auf einer richtigen Straße gibt, sondern nur die Entfernungen zur Mittellinie gemessen werden kann, weicht der Algorithmus in T.O.R.C.S. im Vergleich zur Realisierung auf den FAUST Fahrzeugen ein wenig ab.

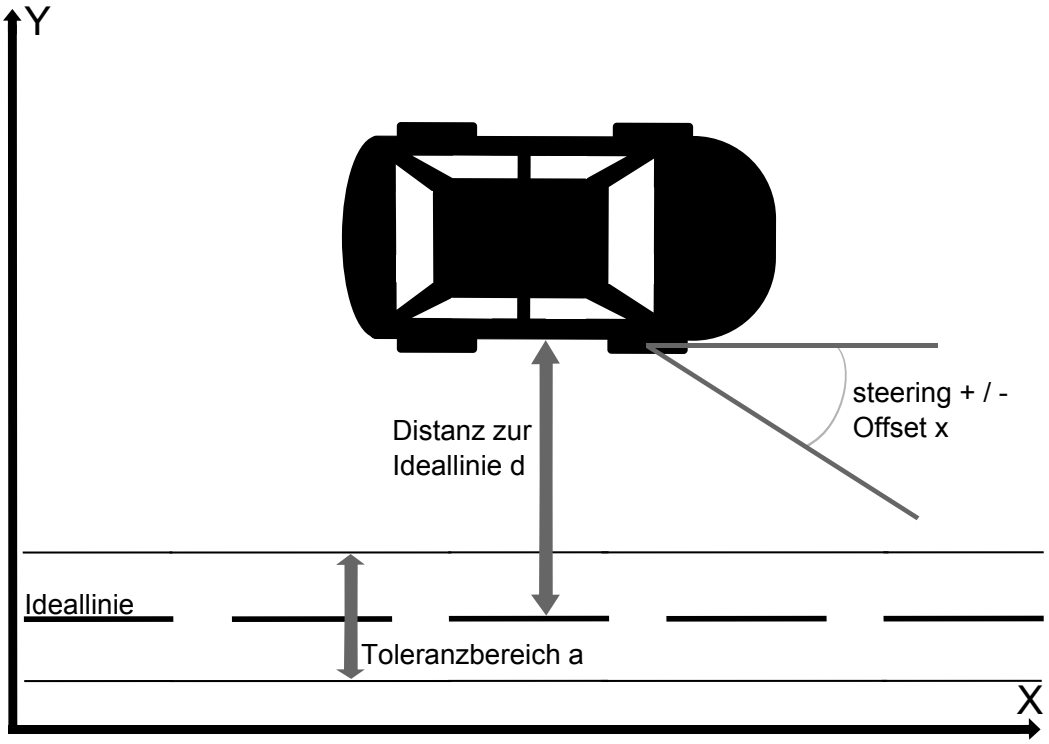

Abbildung 7: Variablen des bisektionalen Algorithmus

Der Algorithmus wird durch folgende Formel beschrieben:

$$steering := \begin{cases} steering - x \text{ falls} & d \geq a \\ steering + x \text{ falls} & d \leq -a \end{cases}$$

Wenn die Entfernung *d* der Mitte des Fahrzeuges den Toleranzbereich *a* verlässt, so weiß der Algorithmus anhand des Vorzeichens von *a*, in welche Richtung das Fahrzeug ausbricht. Ist es im positiven Bereich, bewegt sich das Fahrzeug zur linken Seite. Befindet sich der Wert von *a* allerdings im negativen Bereich, so fährt das Fahrzeug zur rechten Seite. Je nach Richtung des Ausbruchs wird nun ein bestimmter Wert *x* zum Lenkwinkel *steering* hinzuaddiert, wenn das Fahrzeug rechtsseitig fährt bzw. subtrahiert, sollte das Fahrzeug sich nach links bewegen. Ist die Entfernung *d* der Mitte des Fahrzeuges wieder im Toleranzbereich *a*, so hört der Algorithmus auf zu arbeiten. Ab nun führt der Algorithmus nur noch geringe Korrekturen in Kurven durch, da das Fahrzeug in Kurven leicht durch die seitlichen auf das Fahrzeug auftretenden Kräfte nach außen gedrückt wird.

4.2 Messender Algorithmus

Ein großes Problem, das dem bisektionalen Algorithmus anlastet, ist, dass er die verstellte Nullstellung nicht sehr effizient, sondern nur durch das Probieren eines selbst definierten festen Wertes, korrigieren kann. Dies kann zeitaufwendig sein und der selbst definierte Wert muss zunächst bestimmt werden. Dieses Problem kann der messende Algorithmus beherrschen, in dem er sich ein bestimmtes Wissen aneignet und immer wieder auf das gelernte Wissen zurückgreifen kann.

Dabei misst der Algorithmus in der Lernphase bei einem bestimmten Streckenintervall die seitliche Abweichung einer der Straßenlinien. Die Messung soll über einen bestimmten Bereich des Lenkwinkels laufen, denn bei einem zu großen Intervall könnte das Fahrzeug die Linie verlieren oder zu schnell die Strecke verlassen, wodurch Gefahren für das Fahrzeug entstehen könnten. Außerdem wäre es möglich, dass verschiedene Profile oder andere Gewichtungen gelernt werden. Es könnte durchaus sinnvoll sein, den unteren Prozentbereich, z.B. -10 bis +10 Prozent, sehr fein granular zu messen und die oberen Bereiche, z.B. -50 bis -10 und +10 bis +50 Prozent, etwas gröber bzw. sehr grob zu messen.

Hat sich der messende Algorithmus dieses Wissen angeeignet, dann weiß er, zu welcher Abweichung welcher Offset auf den angegebenen Lenkwinkel hinzugerechnet werden muss. Sollte dann immer noch eine gewisse Abweichung auftreten, so kann er die vorher eingemessenen feineren Messungen verwenden, um sich noch genauer einzustellen.

4.2.1 Mögliche Formeln

Es gibt mehrere Möglichkeiten um eine Korrektur des Lenkwinkels zu berechnen. Die beiden hier verwendeten Möglichkeiten umfassen einerseits eine Approximation der Messwerte mittels eines Polynoms und andererseits das Suchen in den Messwerten und die Subtraktion des gefundenen Messwertes.

Die erste Möglichkeit ist das Suchen eines Messwertes in einer Liste. Dies geschieht nach folgendem Schema:

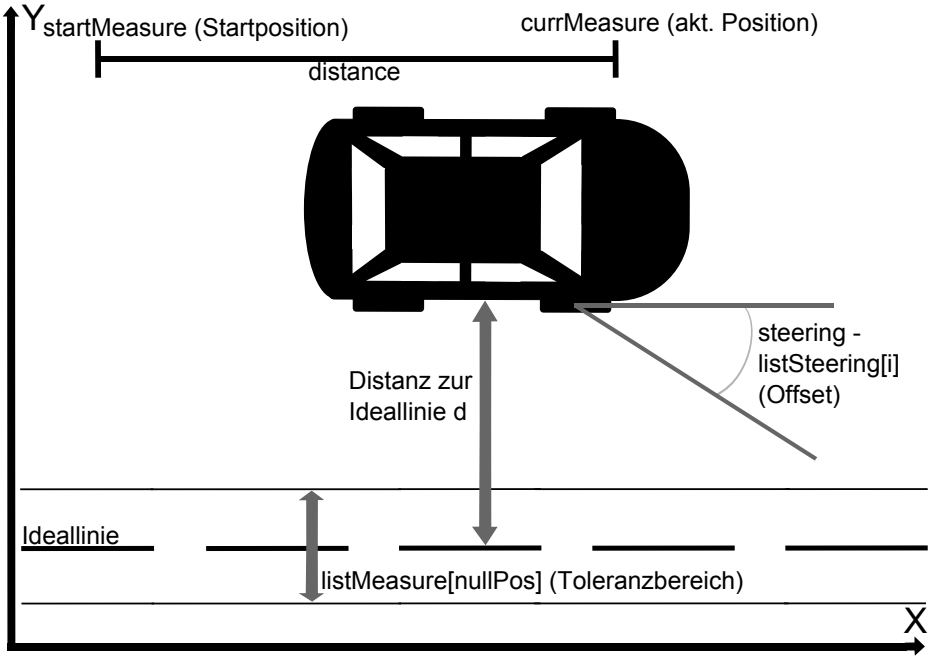

Abbildung 8: Variablen des messenden Algorithmus

$$i := \begin{cases} nullPos \text{ falls} & currMeasure >= startMeasure + distance \\ i+1 \text{ falls} & d < -listMeasure[nullPos] \text{ solange} \quad listMeasure[i] \leq d \\ i-1 \text{ falls} & d > listMeasure[nullPos] \text{ solange} \quad listMeasure[i] \geq d \end{cases}$$

Ist der entsprechende Wert gefunden, wird der Index i in folgende Formel eingesetzt:

$$steering = steering - \frac{listSteering[i]}{100}$$

Hat das Fahrzeug eine bestimmte Distanz überwunden, d.h. *currMeasure*, der die aktuell zurückgelegte Distanz angibt, ist größer als *startMeasure*, der die beim Messanfang zurückgelegte Distanz angibt, plus *distance*, der die zurückzulegende Strecke angibt. Diese Distanzüberwindung ist nötig um ein einigermaßen stabiles Messergebnis zu erhalten und große Veränderungen auszuschließen. Ist diese Distanz überwunden, wird die Index Variable i mit *nullPos* initialisiert. *nullPos* enthält den Index, der beim Messvorgang um null Prozent des Lenkwinkels erreicht wurde. Dieser Wert ist abhängig von der Anzahl der Messungen. Ist die Distanz d zur Mittellinie größer als *listMeasure[nullPos]*, der den Messwert um null Prozent des Lenkwinkels enthält, so wird der Index i um eins dekrementiert, bis der

Messwert *listMeasure[i]* größer oder gleich der Distanz *d* zur Mittellinie ist. Sollte die Distanz *d* kleiner als *-listMeasure[nullPos]* sein, so wird der Index *i* um eins inkrementiert, bis der Messwert *listMeasure[i]* kleiner oder gleich der Distanz *d* ist. Ist der Index *i* bestimmt, wird anhand dieses Indexes ein Korrekturwert für die Lenkung aus der Liste *listSteering[i]* entnommen und durch 100 dividiert, um den Wert im Bereich - eins bis + eins zu normieren. Dieser Wert wird vom aktuellen Korrekturwert subtrahiert. Durch die Subtraktion wird automatisch die richtige Korrektur vorgenommen.

Eine andere Möglichkeit besteht in der Approximation der Messwerte mittels eines Polynoms. Dabei kann der Grad des Polynoms variieren. Bilden die Messwerte fast eine Gerade, so reicht ein Polynom ersten Grades. Sind mehrere Kurven vorhanden, so wird ein Polynom höheren Grades benötigt. Diese Methode der Berechnung läuft nach folgendem Schema ab:

$$steering = steering - \frac{polyApp(d)}{100}$$

, wenn $currMeasure >= startMeasure + distance$ und $d > nullPos$ oder $d < -nullPos$ erfüllt ist

Ebenfalls wie bei der vorherigen Methode, muss das Fahrzeug eine bestimmte Distanz überwinden. Es muss also *currMeasure* größer als *startMeasure* plus *distance* sein. Ist diese Bedingung erfüllt, muss der Abstand *d* zur Mittellinie größer sein als *nullPos* bzw. kleiner sein als *-nullPos*. *nullPos* gibt in dieser Methode der Berechnung keinen Index an, sondern den um den Nullpunkt gemessenen Abstand zur Mittellinie. Sind beide Bedingungen erfüllt, wird das Polynom *polyApp*, welches vor Beginn der Korrektur mittels Messwerten approximiert wird, in Abhängigkeit vom Abstand *d* zur Mittellinie berechnet, durch 100 dividiert und vom aktuellen Korrekturwert subtrahiert.

4.2.2 Mögliche Lernszenarien

Es gibt unterschiedliche Szenarien, nach dem der Algorithmus sich das benötigte Wissen aneignen kann. Und je nach Szenario kann der Algorithmus später besser bzw. schlechter die Nullstellung der Lenkung ermitteln.

Dabei ist das einfachste Szenario, dass der Algorithmus in bestimmten Intervallschritten auf dem gesamten festgelegten Intervall die Lenkung verstellt und dann die Abweichung misst. Durch dieses Verfahren bekommt der Algorithmus eine gleichmäßige Darstellung des gesamten festgelegten Intervalls der Lenkung. Nachteil dieses Verfahrens ist, dass sich das Fahrzeug nur grob die neue Nullstellung justieren kann, maximal auf einen Intervallschritt genau, da das gelernte Wissen nicht detaillierter ist.

Ein besseres Lernszenario könnte sein, dass man bestimmte Bereiche fokussiert, in denen in besonders kleinen Schritten gemessen wird und andere Bereiche eher grob ausgemessen werden. Dabei bietet sich vor allem der Bereich um null Prozent des Lenkwinkels an, diesen

sehr fein darzustellen, da durch dieses Wissen eine sehr genaue Bestimmung der neuen Nullstelle stattfinden kann. Wird der höhere Bereich eher grob dargestellt, z.B. in zehn oder fünf Prozent Schritten, dann kann eine erste grobe Justierung erfolgen und anschließend eine feinere.

4.2.3 Reinforcement Learning als Alternative

Eine Implementierung eines Reinforcement Learning Algorithmus ist in diesem Fall nicht möglich. Ein Grund, den Reinforcement Learning Algorithmus zu verwenden, wäre, dass man neue Abschnitte neu einmessen könnte, wenn der Algorithmus auf keine akzeptable Lösung kommt. Aber dieser Umstand kann nur erkannt werden, wenn bereits die Lenkung verstellt ist. Allerdings setzten die Messungen voraus, dass der Lenkwinkel nahezu perfekt eingestellt ist, damit die Messung später auch mit einem Messergebnis eines verstellten Lenkwinkels zu Recht kommen kann. Wird eine Messung mit verstelltem Lenkwinkel vorgenommen, so wird der Algorithmus später bei der Korrektur nicht den Offset verwenden, der eigentlich richtig wäre. Aus diesem Grund ist eine Implementierung eines Reinforcement Learning Algorithmus nicht möglich.

4.3 Die Zusatzfunktionen Geschwindigkeitsregler, Distanzmessung und Schaltung

T.O.R.C.S. stellt zwar eine Fülle an Daten und Kontrollmöglichkeiten zur Verfügung, dennoch kann es passieren, dass gewisse Daten nicht im gewünschten Typ oder gewünschter Einheit vorliegen bzw. gänzlich fehlen. In T.O.R.C.S. fehlt eine genaue Messung der Distanz, die das Fahrzeug bereits auf der Fahrbahn zurückgelegt hat. Es liefert immer nur segmentweise die Entfernung des Fahrzeuges zur Start- oder Ziellinie. Aus diesem Grund wurde ein eigenes Messsystem zur Distanzmessung entwickelt. T.O.R.C.S. bietet auch nicht die Möglichkeit, einem Roboter zu sagen, wie schnell er fahren soll. Dies kann nur über die Drehzahl des Motors geregelt werden. Deshalb wurde ein Regler implementiert, der diese Aufgabe übernimmt.

4.3.1 Algorithmus zur Messung von Distanzen

Da T.O.R.C.S. die Messung von Strecken nur in Segmentlängen erlaubt, wurde für die Distanzmessung ein eigener Algorithmus entwickelt, der auch wesentlich kleinere Distanzen messen kann. Dabei hängt die minimale messbare Distanz von der Geschwindigkeit des Fahrzeuges sowie der vergangen Zeit seit dem letzten Aufruf des Algorithmus ab. Anhand dieser beiden Werte wird die in der vergangen Zeit zurückgelegte Strecke ausgerechnet und zu einem Gesamtwert hinzuaddiert, den dann andere Funktionen benutzen können.

Die Formel für die Berechnung der Distanz sieht wie folgt aus:

$$distance = distance + speed * \Delta t$$

speed hat als Einheit $\frac{cm}{s}$, wird von T.O.R.C.S. allerdings als $\frac{m}{s}$ zurückgegeben und somit umgerechnet. Δt ist die Zeitdifferenz zwischen zwei Messungen.

4.3.2 Algorithmus zum Halten einer Geschwindigkeit

In T.O.R.C.S. ist keine Funktion implementiert, die es erlaubt, eine bestimmte und konstante Geschwindigkeit zu fahren. Aus diesem Grund musste ein eigenes Regelungssystem entwickelt werden. Dabei kann dies nur erreicht werden, in dem der Motor eine bestimmte Beschleunigung erhält. Die Beschleunigung muss in T.O.R.C.S. einen Wert zwischen null und eins haben, welcher für die prozentuale Beschleunigung des Motors steht. Der entwickelte Regler beschleunigt so lange mit Vollgas, bis die Istgeschwindigkeit sehr nahe an der Sollgeschwindigkeit ist. Zum Einsatz kommt ein PI-Regler, da zwar das P-Glied eine schnelle und starke Beschleunigung ermöglicht, allerdings wenn der Sollwert erreicht ist, kann die Geschwindigkeit nicht stabil gehalten werden. Deswegen wurde zusätzlich ein I-Glied hinzugefügt, welches die Geschwindigkeit fast konstant halten kann.

Das PI Glied berechnet mit folgenden Formeln den Ausgang:

$$controllerDifference = vSoll - (vIst)$$

$$iPart = iPart + controllerDifference$$

$$out = Kp * controllerDifference + Ki * tn * iPart$$

vSoll und *vIst* haben beide als Einheit $\frac{km}{h}$. Dabei gibt *vSoll* die Geschwindigkeit an, die konstant gehalten werden soll und *vIst* die momentane Geschwindigkeit. *iPart* integriert die Differenz aus Soll- und Istgeschwindigkeit *controllerDifference*. *Kp* ist der Anteil des Proportionalgliedes und *Ki* der Anteil des Integriergliedes. Die zeitliche Differenz wird durch *tn* dargestellt.

4.3.3 Algorithmus zum Schalten von Gängen

Der Algorithmus zum Hoch- und Runterschalten der Gänge wurde aus dem *T.O.R.C.S. Manual installation and Robot tutorial* (Wymann, 2005b) entnommen. Dabei berechnet dieser Algorithmus anhand von bestimmten Daten die Geschwindigkeit, ab der in den nächsthöheren bzw. kleineren Gang geschaltet werden muss.

Die Formeln zur Berechnung dieser Geschwindigkeit, um einen Gang hochzuschalten, sind folgende:

$$Wenn\ \omega \cdot w_r \cdot SHIFT < speed\ ,\ dann\ gear = gear + 1$$

Um einen Gang herunterzuschalten, muss folgende Formel angewendet werden:

$$Wenn\ \omega \cdot w_r \cdot SHIFT < speed + SHIFT_MARGIN\ ,\ dann\ gear = gear - 1$$

In dieser Formel ist das ω eine Division der maximal möglichen Drehzahl des Motors durch das Schaltverhältnis des nächsthöheren bzw. niedrigeren Ganges. w_r ist der Umfang des Reifens in Metern. *SHIFT* und *SHIFT_MARGIN* sind selbst definierte Werte, die die Schaltung beeinflussen. Je nachdem wie diese Werte gewählt werden, wird der Gang früher oder später hoch- bzw. runtergeschaltet.

5 Implementierung der Zusatzfunktionen und Algorithmen

5.1 Implementierung der Zusatzfunktionen

5.1.1 Distanzmessung

Die Funktion *double getDistance(tCarElt* car)* erwartet einen Pointer auf die Fahrzeugstruktur vom Typ *tCarElt*. Als Rückgabewert hat die Funktion *getDistance()* den Wert *double*, der die insgesamt zurückgelegte Strecke pro Runde seit dem letzten Aufruf zurückgibt.

An Variablen verwendet die Funktion *getDistance()* folgende: *distance* ist vom Typ *static double* und speichert die zurückgelegte Strecke, die am Ende der Funktion auch zurückgegeben wird. In der Variable *timeOld* wird die aktuelle Rundenzeit gespeichert, um sie bei der nächsten Berechnung zum Bilden der Differenz zu verwenden. Die Variable hat den Typ *static double*. Die letzte Variable ist *lapsOld*. Diese Variable zählt die Anzahl der Runden, um sie bei der nächsten Berechnung zu vergleichen und zu schauen, ob das Fahrzeug eine neue Runde beginnt. Der Typ ist *static int*.

Bei der Berechnung der Distanz gibt es in der ersten Runde eine kleine Besonderheit. Das Fahrzeug startet hinter der Startlinie und zählt somit die ersten gefahrenen Meter auch mit, allerdings wird beim Überfahren der Startlinie die Rundenzeit nicht zurückgesetzt. Aus diesem Grund wurde eine if-Abfrage eingebaut, die dieses Phänomen abfragt und dann die Distanz wieder auf null setzt, aber die Zeit weiter laufen lässt.

5.1.2 Geschwindigkeitsregeler

double getAcc(tCarElt car, double vmax)* ist die Funktion zum Berechnen der Beschleunigung des Motors. Dabei erwartet *getAcc()* zwei Übergabeparameter: einen Pointer von dem Datentyp *tCarElt*, der auf eine Fahrzeugstruktur zeigt sowie einen Wert mit dem Datentyp *double*. Dieser Wert beinhaltet die maximale Geschwindigkeit, die das Fahrzeug fahren soll. Als Rückgabeparameter hat die Funktion *getAcc()* einen Wert vom Datentyp *double*.

Die Funktion *getAcc()* besitzt fünf Variablen und zwei Defines: *timeOld* speichert die Rundenzeit des letzten Aufrufes. Die Variable hat den Datentyp *static double*. Der I-Anteil des Reglers wird mithilfe von den Variablen *iPart* und *tn* sowie *Ki* berechnet. Dabei besitzt die Variable *iPart* den Datentyp *static double* und addiert die Differenz zwischen Sollwert und Istwert auf. In der Variablen *tn* wird die zeitliche Differenz zwischen dem letzten und aktuellen Aufruf der Funktion gespeichert. Sie ist vom Datentyp *double*. *Ki* ist ein Define und enthält den Anteil des I-Gliedes.

Kp und *controllerDifference* sind für die Berechnung des P-Anteils. Das Define *Kp* enthält den Anteil des P-Gliedes, während *controllerDifference* eine Variable vom Datentyp *double* ist und die Differenz zwischen Sollwert und Istwert speichert. Die letzte Variable *out* hat ebenfalls den Datentyp *double* und speichert den Ausgang, der zurückgegeben wird.

Sollte *controllerDifference* größer als null werden und *iPart* kleiner als null oder *iPart* größer als null und *controllerDifference* kleiner als null, dann wird *iPart* wieder auf null zurückgesetzt.

Weil die Beschleunigung des Motors nur Werte zwischen null und eins sein kann, wird der Ausgang *out* vor der Rückgabe noch normiert. Alle Werte, die größer als eins sind, werden auf eins abgebildet, alle Werte, die kleiner als null sind, werden auf null abgebildet.

5.1.3 Schaltung

Die Funktion *int getGear(tCarElt *car)* berechnet anhand bestimmter Daten, die aus dem Pointer *car* entnommen, in welchen Gang das Fahrzeug zurzeit fahren muss. Als Übergabewert erwartet die Funktion *getGear()* einen Pointer auf die Struktur *tCarElt*. Der Rückgabewert ist der Gang, mit dem das Fahrzeug zurzeit fahren soll.

Da das Fahrzeug hauptsächlich vorwärts fahren soll und es eigentlich nicht vorgesehen ist, dass es rückwärts fährt, wird, sobald der eingelegte Gang kleiner gleich null ist, eine Eins zurückgegeben, damit das Fahrzeug in den ersten Gang schaltet. Ist dies nicht der Fall, so werden bestimmte Parameter berechnet.

Der erste Parameter, der berechnet wird, ist *gr_up*. Dieser ist vom Typ *float* und gibt das Schaltverhältnis an. Dabei wird der Index durch die Addition des aktuellen Ganges plus einem fahrzeugspezifischen Offset bestimmt. Der Parameter *omega* ist ebenfalls vom Typ *float* und ist die Division der Drehzahl, die sich zwischen g"runem und rotem Bereich befindet, durch den Parameter *gr_up*. Der letzte Parameter ist *wr*. Auch dieser Parameter besitzt den Typ *float* und beinhaltet den Radius des Rades.

Ist die Multiplikation von *omega*, *wr* und *SHIFT* kleiner als die aktuelle Geschwindigkeit, die in $fracms$ angegeben wird, so wird ein Gang hochgeschaltet. Die Konstante *SHIFT* gibt an, ab wie viel Prozent der Drehzahl zwischen null und dem roten Bereich geschaltet werden soll. Sollte dieser Fall nicht eintreffen, wird ein neuer Parameter *gr_down* berechnet. Genau wie bei *gr_up* wird auch hier wieder das Schaltverhältnis ermittelt, allerdings wird für den Index nun von aktuellem Gang plus Offset eine Eins subtrahiert. Ausgehend von dem *gr_down* wird der Parameter *omega* neu berechnet.

Sofern das Fahrzeug sich nicht im ersten Gang befindet, wird wieder die Multiplikation von *omega*, *wr* und *SHIFT* durchgeführt und sollte dieser größer sein als die aktuelle Geschwindigkeit plus *SHIFT_MARGIN*, so wird ein Gang heruntergeschaltet. *SHIFT_MARGIN* ist eine Konstante und gibt den Schwellwert an, ab dem heruntergeschaltet werden soll.

5.2 Bisektionaler Algorithmus

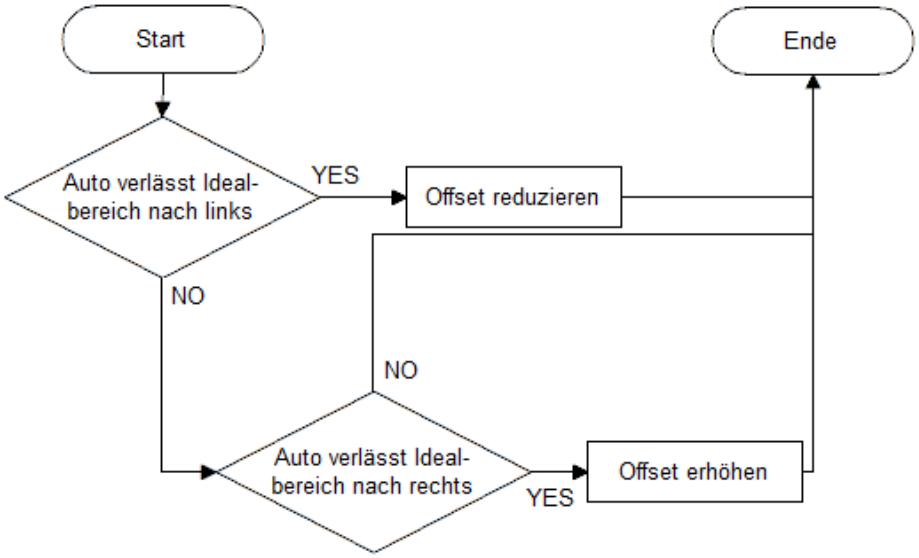

Abbildung 9: Ablaufdiagramm der Funktion *bisectionalAlgorithm()*

Der Algorithmus wird so implementiert, wie es im Kapitel 4.1 beschrieben ist. Dazu wird eine Funktion *double bisectionalAlgorithm(tCarElt* car)* erstellt. Als Übergabeparameter wird eine Referenz auf das Fahrzeug erwartet. Der Rückgabewert ist die Korrektur des Lenkwinkels mit dem Datentyp *double*.

In der Funktion wird abgefragt, ob sich das Fahrzeug im Bereich von *-bisecAlgo_distToMid* bis *+bisecAlgo_distToMid* von der Mittellinie *car->_trkPos.toMiddle* entfernt hat. Dabei ist *bisecAlgo_distToMid* ein Define. Ist dies der Fall, so wird ein selbst definierter Wert *bisecAlgo_intervall*, der ebenfalls ein Define ist, zur Korrektur je nach Richtung des Fahrverhaltens hinzuaddiert oder subtrahiert. Bricht das Fahrzeug nach links aus, so wird *bisecAlgo_intervall* von der Korrektur abgezogen. Sollte das Fahrzeug nach rechts ausbrechen, wird *bisecAlgo_intervall* hinzuaddiert.

5.3 Messender Algorithmus

Die Implementierung des messenden Algorithmus besteht im Wesentlichen aus zwei Funktionen. Die eine Funktion misst die Abweichung von einer der Leitlinien, wenn ein Offset zu der Lenkung hinzuaddiert wird und speichert diesen Wert in einer Datei. Die andere Funktion liest diese Datei aus und approximiert aus den vorhandenen Werten ein Polynom oder durchsucht die Werte nach der am besten passenden Abweichung und subtrahieren das Ergebnis bzw. den Wert, den die Abweichung referenziert.

5.3.1 Die Funktion measure()

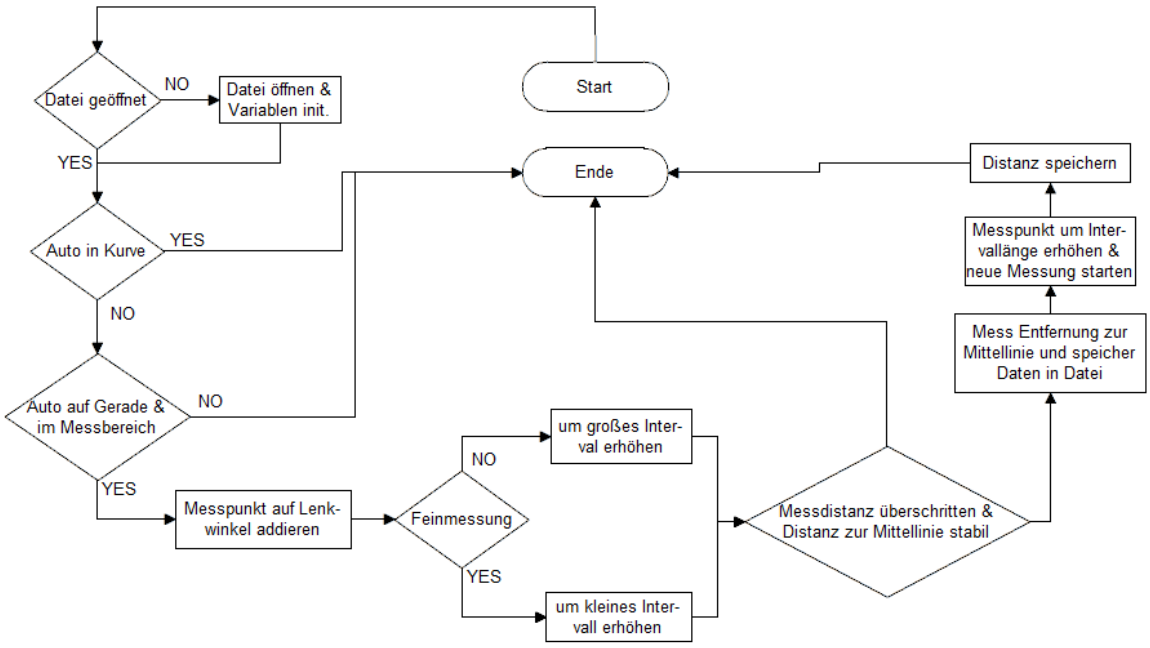

Abbildung 10: Ablaufdiagramm der Funktion *measure()*

Die Funktion *void measure(tCarElt* car, double distance)* besitzt die Aufgabe, die Abweichung von einer der Leitlinien, in diesem Fall der Mittellinie, zu messen und in eine Datei zu speichern. Für diese Aufgabe benötigt die Funktion zwei Übergabeparameter: einmal einen Pointer auf die Struktur *tCarElt*, die sämtlichen Fahrzeugdaten enthält und zum anderen die Distanz *distance*, die seit dem Überschreiten der Startlinie zurückgelegt wurde.

Durch acht Definitionen kann man die Funktion *measure()* einstellen. Es gibt die Definition *learnAlgo_driveDistance*, welcher die Distanz in Metern angibt, die zurückgelegt werden muss, bis ein Messpunkt erfasst wird. Dies dient dazu, um keine Schwankungen im Lenkverhalten mehr zu haben. *learnAlgo_intervallMeasure* gibt die Intervalllänge an, die nach jeder Messung hinzuaddiert wird. Die Definition *learnAlgo_precisionMeasure* gibt die Intervalllänge an, die im Präzisionsbereich addiert werden soll. *learnAlgo_fineMeasureStart* und *learnAlgo_fineMeasureEnd* geben das Intervall an, auf dem die präzisen Messungen durchgeführt werden sollen. Äquivalent dazu geben *learnAlgo_measureStart* und *learnAlgo_measureEnd* das Intervall an, auf dem insgesamt gemessen werden soll. Das Define *learnAlgo_stabi* setzt die Differenz, die die Entfernung der Mittellinie zwischen zwei Messungen haben darf, um als stabil zu gelten.

Fünf Variablen werden benötigt, damit die Funktion *measure()* korrekt arbeiten kann. Die Variable *started* ist vom Datentyp *static boolean* und speichert, ob die Funktion bereits

gestartet wurde oder noch gestartet werden muss. *offset* hat den Datentyp *static double*. Diese Variable speichert den Wert, der zum aktuellen Lenkwinkel hinzuaddiert wird, um den eingestellten Offset der Lenkung zu eliminieren. Die Startdistanz wird in der Variable *startDistance* gespeichert und ist vom Typ *static double*. Die Variable *toMidOld* ist vom Datentyp *static double* und *toMid* ist vom Datentyp *double*. Sie speichern beide die Distanz zur Mittellinie. *toMid* speichert die aktuelle Distanz, *toMidOld* die Distanz aus dem vorherigen Aufruf der Funktion *measure()*.

Es gibt vier if-Abfragen, die das Grundgerüst der Funktion *measure()* darstellen. Die erste if-Abfrage überprüft, ob die Daten Datei schon geöffnet wurde. Wenn dies nicht der Fall ist, wird eine Datei namens *data.txt* zum Schreiben geöffnet. Die zweite if-Abfrage überprüft, ob die Funktion noch nicht gestartet wurde, ob zum ersten Mal die Startlinie überfahren wurde und sich das Fahrzeug auf einer geraden Strecke befindet. Sollten alle drei Abfragen positiv ausfallen, werden die Variablen *offset*, *startDistance* sowie *started* initialisiert bzw. gesetzt. Dabei bekommt *offset* die Definition von *measureStart*, *startDistance* wird mit *distance* initialisiert und *started* wird auf *true* gesetzt. Die dritte if-Abfrage kontrolliert, ob sich das Fahrzeug auf einer geraden Strecke befindet. Sollte dies nicht der Fall sein, so wird *startDistance* immer wieder überschrieben mit dem aktuellen Inhalt von *distance*.

Die vierte if-Abfrage kontrolliert, ob die Funktion gestartet wurde, das Intervallende noch nicht erreicht wurde sowie sich das Fahrzeug auf einer geraden Strecke befindet. Sind alle Bedingungen erfüllt, wird die Variable *offset* normiert und zum aktuellen Lenkwinkel hinzuaddiert. Als Nächstes wird das Intervall festgelegt. Wenn sich die Variable *offset* im Bereich der Feinmessung befindet, wird als Intervall die Definition *learnAlgo_precisionMeasure* verwendet, ansonsten *learnAlgo_intervallMeasure*. Sollte sich die Entfernung zur Mittellinie im Vergleich zum letzten Aufruf nicht mehr als *learnAlgo_stabi* verändert haben und die zurückzulegende Strecke *driveDistance* überschritten sein, wird die Entfernung zur Mittellinie in die Daten Datei geschrieben inklusive dem aktuellen Offset.

Als Letztes wird der Wert von *toMid* in die Variable *toMidOld* gespeichert.

5.3.2 Die Funktion detectAndCorrect() mittels Suchen in einer Map

Die Funktion *void detectAndCorrect(tCarElt* car, double distance)*, welche als Übergabeparameter einen Pointer auf die Struktur *tCarElt*, der sämtlichen Fahrzeugdaten enthält und zum anderen die Distanz *distance*, die seit dem Überschreiten der Startlinie zurückgelegt wurde, misst die seitliche Versetzung des Fahrzeuges zur Mittellinie und gleicht den gemessenen Wert mit den zuvor gemessenen Daten ab und korrigiert anhand dieser Daten den Lenkwinkel. Dafür wird zunächst die Datei *data.txt* eingelesen und in zwei Maps übertragen, sofern diese Operation noch nicht erledigt wurde: *listMeasure* ist die Map, die die seitliche Entfernung zur Mittellinie beinhaltet und *listSteering* enthält die dazugehörige Lenkwinkelkorrektur. Beide Maps haben als Suchindex den Datentyp *int* und als Datenwert

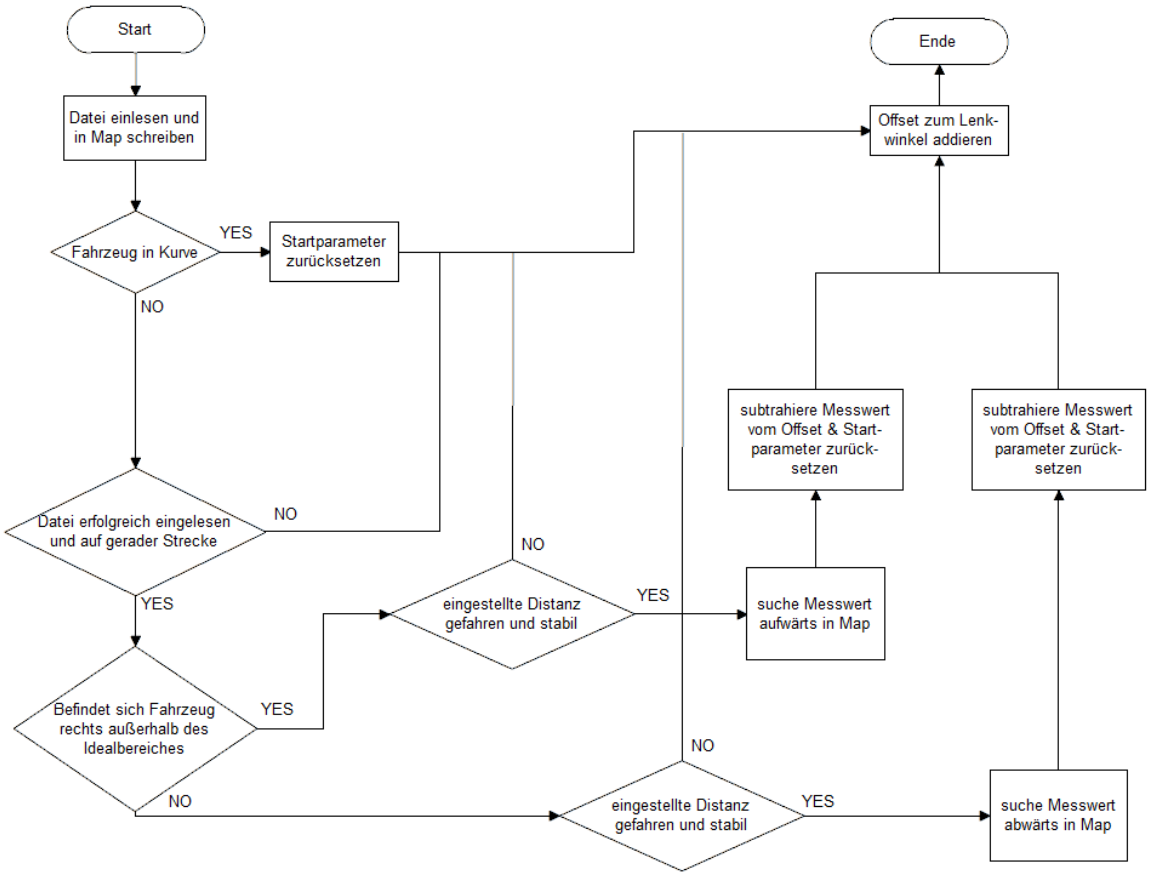

Abbildung 11: Ablaufdiagramm der Funktion *detectAndCorrect()* mittels Suche in Map

den Datentyp *double* sowie sind *static*. Dabei wird die Variable *opened*, die vom Typ *static boolean* ist, auf *true* gesetzt und *steeringNull*, die vom Typ *static int* ist, bekommt den Index, bei der der Lenkwinkel gleich null ist.

Ist das Einlesen der Datei erledigt, sind nur noch zwei if-Abfragen relevant. Die erste if-Abfrage überprüft, ob sich das Fahrzeug auf einer geraden Strecke befindet. Ist dies nicht der Fall wird die Variable *startDistance*, welche vom Datentyp *static double* ist, kontinuierlich mit dem Übergabeparameter *distance* überschrieben. Die zweite if-Abfrage kontrolliert, ob die Datei erfolgreich eingelesen wurde, was durch die Variable *opened* überprüft wird und ob sich das Fahrzeug auf einer geraden Strecke befindet.

Ist diese Bedingung erfüllt, wird geschaut, ob die aktuelle Entfernung zur Mittellinie größer bzw. kleiner ist, als der Bereich, der bei der Funktion *measure()* bei einem Lenkwinkel von 0 Prozent gemessen wurde. Ist die Entfernung größer als der Wert, der bei *measure()* bei einem Lenkwinkel von 0 Prozent gemessen wurde, wird noch überprüft, ob der Messwert stabil ist, d.h. er ändert sich nicht mehr als einen Millimeter und die zu fahrende Distanz ist

auch überschritten. Die aktuelle Distanz *distance* muss größer sein als *startDistance* plus *learnAlgo_driveDistance*, welches ein Define ist. Sollte dies der Fall sein, wird in der Map nach einem entsprechenden Eintrag gesucht, indem die Indexvariable *i* mit *steeringNull* initialisiert wird und danach inkrementiert wird, der am besten zur gemessenen seitlichen Entfernung zur Mittellinie passt und der dazu gehörige Offset wird von der Variablen *offset*, welche vom Datentyp *static double* ist, subtrahiert. Danach wird die Variable *startDistance* mit dem aktuellen Inhalt von *distance* neu beschrieben.

Ist die gemessene Entfernung allerdings kleiner als der Wert, der bei *measure()* bei einem Lenkwinkel von null Prozent gemessen wurde, wird ebenfalls überprüft, ob der Messwert stabil ist und das Fahrzeug die Distanz *learnAlgo_driveDistance* zurückgelegt hat. Sind diese Bedingungen erfüllt, wird die Indexvariable *i* wird mit *steeringNull* initialisiert, allerdings wird beim Durchlaufen der Map nun die Indexvariable *i* dekrementiert, bis ein passender Wert gefunden wurde. Danach wird dieser Wert ebenfalls von der Variablen *offset* subtrahiert und eine neue Messung durchgeführt, in dem *startDistance* neu beschrieben wird.

Als letzter Schritt wird die Variable *offset* zum aktuellen Lenkwinkel *car->_steerCmd* hinzuaddiert und die Variable *toMidOld* wird mit der aktuellen Entfernung zur Mittellinie *toMid* überschrieben.

5.3.3 Die Funktion detectAndCorrect() mittels Polynomapproximation

Eine andere Möglichkeit zur Bestimmung des passenden Offsets bietet die Polynomapproximation anstelle des Suchens in einer Map. Dabei gibt es zwei mögliche Implementierungsarten. Die eine ist, dass die Implementierung aus den gemessenen Abweichungen selbst die Polynomapproximation vornimmt. Der andere Weg ist, dass die Messwerte von Matlab mittels der Funktion *polyfit()* in ein approximiertes Polynom umgewandelt werden und das Polynom eins zu eins implementiert wird.

Der Vorteil der zweiten Methode liegt klar in dem geringeren Programmieraufwand und dem exakteren Polynom. Wenn die erste Methode verwendet wird, kann man aus den Messwerten nur schlecht erkennen, welchen Grad das Polynom haben muss, um eine gute Approximation zu erreichen. Entweder es wird ein zu hoher Grad gewählt und die meisten Koeffizienten sind null oder der Grad wird zu gering gewählt und die Approximation gilt nur für einen sehr kleinen Bereich, den man aber ohne vorherige eigene Überprüfung nicht feststellen kann. Aus diesem Grund wird in T.O.R.C.S. die zweite Methode angewendet.

Während bei der Implementierung der Funktion *detectAndCorrect()* mittels Suchen (Kapitel 5.3.2) viel Programm-Code und zum Teil auch viel fast gleicher Programm-Code benötigt wird, um die jeweilige Seite zu bearbeiten, kann dies in der Implementierung mittels Polynomapproximation durch zwei if-Abfragen realisiert werden.

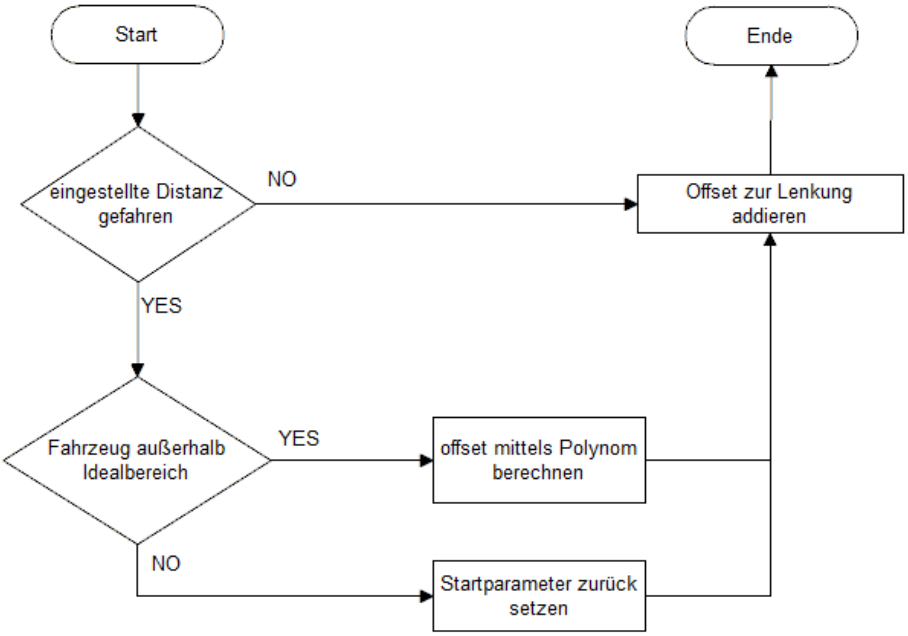

Abbildung 12: Ablaufdiagramm der Funktion *detectAndCorrect()* mittels Polynomapproximation

Die genutzten Variablen sind einmal *offset* und zum anderen *startDistance*. Beide Variablen sind vom Typ *static double*. Die Variable *offset* beinhaltet den aktuellen Offset, der zur Lenkung hinzuaddiert wird und die Variable *startDistance* die Distanz von der Startlinie bis zum Begin der Messung. Darüber hinaus gibt es drei Defines. Zum einen das bereits bei der Funktion *measure()* genutzte Define *learnAlgo_driveDistance*, *learnAlgo_timeToWait*, welches die Zeit angibt, die vergehen muss, bis eine Änderung durchgeführt werden darf sowie ein Define *learnAlgo_disToMid* für den Idealbereich, innerhalb dessen keine Korrekturen vorgenommen werden müssen. Als Alternative kann man anstelle der Distanz auch mit der Zeit arbeiten. Dabei wird dann die Startzeit gespeichert sowie mittels eines Defines anstelle von *driveDistance* ein Define namens *waitingTime* genutzt.

Die Funktion *void detectAndCorrectPoly(tCarElt* car, double distance)* besitzt keinen Rückgabewert, allerdings zwei Übergabeparameter. Beim ersten Übergabeparameter handelt es sich um die Fahrzeugdaten, ein Pointer auf die Struktur *tCarElt*, beim zweiten Übergabeparameter um die zurückgelegte Distanz. Dieser kann entfallen, wenn anstelle der Distanz die Zeit genommen wird.

Zwei if-Abfragen überprüfen, ob der Algorithmus eine Korrektur vornehmen muss. Dabei überprüft die erste if-Abfrage, ob die festgelegte Strecke *learnAlgo_driveDistance* gefahren wurde oder die eingestellte Zeit *learnAlgo_timeToWait* vergangen ist. Die zweite if-Abfrage kontrolliert, ob das Fahrzeug sich im Idealbereich befindet oder außerhalb. Befindet es sich

außerhalb des Idealbereiches, wird mittels des Polynoms und der Entfernung zur Mittellinie ein Offset berechnet, der von der Variablen *offset* subtrahiert wird. Sollte sich das Fahrzeug innerhalb des Idealbereiches befinden, wird nur der Startparameter zurückgesetzt, d.h. dass *startDistance* mit der aktuellen Distanz überschrieben wird bzw. in *startTime* die aktuelle Zeit hinein geschrieben wird. Als Letztes wird die Variable *offset* zum aktuellen Lenkwinkel hinzuaddiert.

6 Test der Zusatzfunktionen und Algorithmen

Alle Tests in T.O.R.C.S. werden auf der in Kapitel 2.4 beschrieben Strecke durchgeführt. Dabei wird eine fast konstante Geschwindigkeit von 50 $\frac{km}{h}$ gefahren. Die Funktionen werden ca. alle 20 Millisekunden von T.O.R.C.S. aufgerufen.

Dabei muss Folgendes beachtet werden: T.O.R.C.S. besitzt eine Mindestrundenzeit. Wird diese nicht eingehalten, so wird das Auto von der Fahrbahn genommen und die Übung beendet. Um diese Mindestrundenzeit nicht zu unterschreiten, muss eine Geschwindigkeit von 25 bis 30 $\frac{km}{h}$ gehalten werden. Aus diesem Grund fährt das Fahrzeug 50 $\frac{km}{h}$, damit dieses Problem nicht auftritt.

6.1 Test der Zusatzfunktionen

6.1.1 Distanzmessung

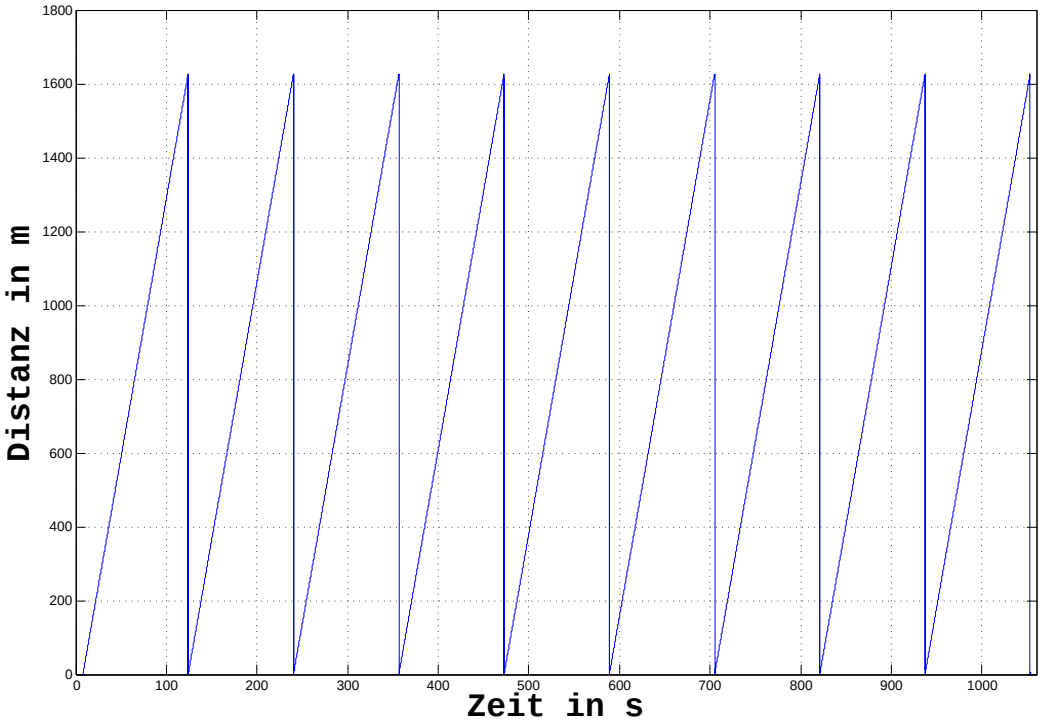

Abbildung 13: Distanzmessung der Teststrecke

Die Abbildung 13 zeigt, dass die Distanz immer gleichmäßig gemessen wird. Gemessen

wurde die Länge der Teststrecke über neun Runden, beginnend bei der ersten Runde. Es wurde also nicht mitten in der Fahrt angefangen zu messen.

Auf der x-Achse wird die Zeit in Sekunden angegeben. Die y-Achse zeigt die gefahrene Distanz in Metern. Dabei kann man auch sehen, dass das Fahrzeug für eine Runde ca. zwei Minuten benötigt und dabei eine Strecke von etwas mehr als 1600 Metern zurücklegt.

6.1.2 Geschwindigkeitsregeler

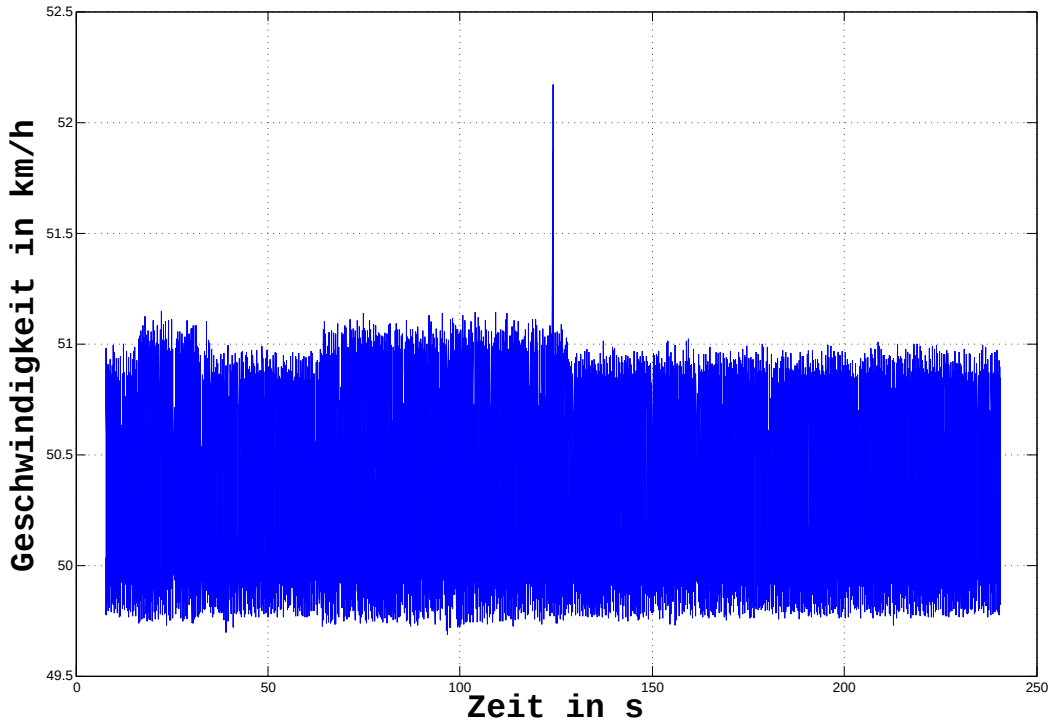

Abbildung 14: Messung der Geschwindigkeit

Auf der x-Achse in Abbildung 14 ist die Zeit in Sekunden aufgetragen, auf der y-Achse die Geschwindigkeit in $\frac{km}{h}$. Man kann in dieser Abbildung die Funktion des Geschwindigkeitreglers gut erkennen, dass dieser eine relativ konstante Geschwindigkeit von ca. 50 $\frac{km}{h}$, die eingestellt wird, halten kann.

Um diesen Wert zu erreichen, werden folgende Anteile verwendet: Der P-Anteil beträgt 10. Der I-Anteil nur 0.5. Mit diesem Wert ist es dem Regler möglich sehr schnell an die gewünschte Geschwindigkeit heranzukommen und der geringe I-Anteil glättet das Ergebnis,

allerdings variiert die Geschwindigkeit in einem Bereich von ca. 49.75 $\frac{km}{h}$ und ca. 51 $\frac{km}{h}$. Dabei gibt es einen Ausreißer bei Sekunde 120, wo die Geschwindigkeit kurzzeitig auch 52 $\frac{km}{h}$ beträgt.

6.2 Test des bisektionalen Algorithmus

Bei dem Test des bisektionalen Algorithmus wird auf der x-Achse die gefahrene Distanz in Metern aufgetragen, auf der y-Achse die Abweichung von der Mittellinie in Metern.

Dabei kann man in Abbildung 15 erkennen, dass eine Änderung des Offsets in Schritten von einem Prozent zu grob gewählt ist. Das Fahrzeug schwingt aufgrund der sehr schnellen und starken Offset-Änderung unaufhörlich, weil der Algorithmus seine eigenen Korrekturen wieder korrigieren muss.

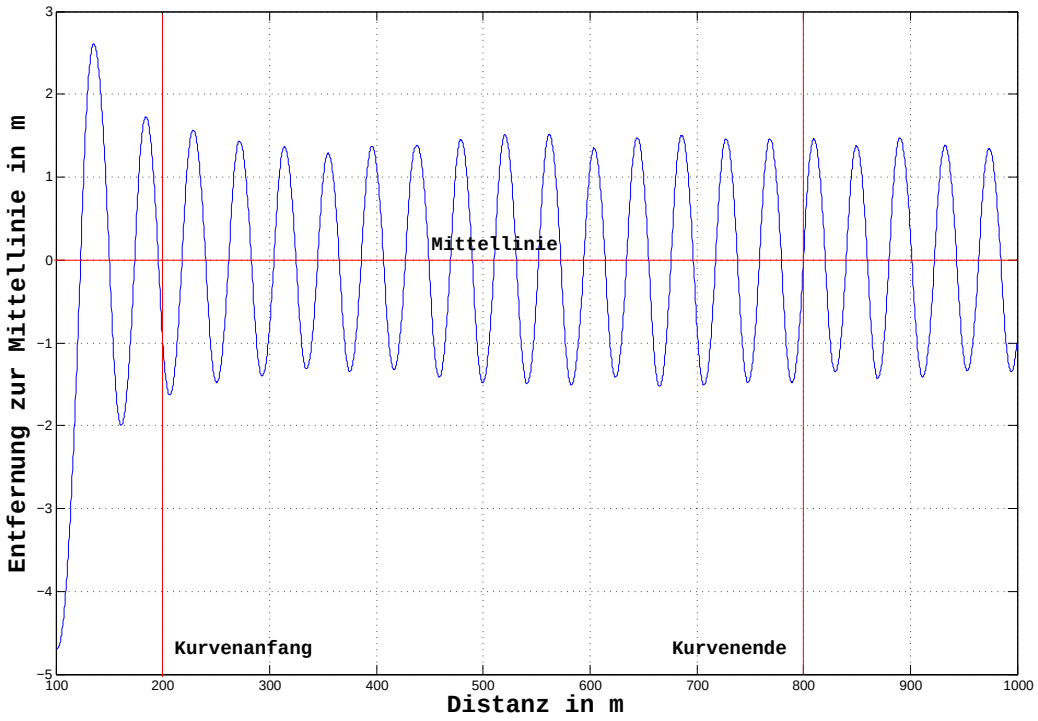

Abbildung 15: Bisektionaler Algorithmus mit Intervallschritt 0.01

Vergleicht man Abbildung 15 mit der Abbildung 16, so kann man sehr schnell erkennen, dass eine Verkleinerung des Intervallschrittes um das Zehnfache des vorherigen Wertes eine sehr gute Lösung bietet. Nach etwa 200 Metern hat der Algorithmus den Offset ausgeglichen. Man kann aber sowohl in der Abbildung als auch am Fahrverhalten erkennen, dass

dieses noch sehr leicht schwingt. Im Kurvenabschnitt tritt dies besonders deutlich hervor, wobei hier die physikalischen Gesetze in Form der Fliehkraft ebenfalls wirken, die das Fahrzeug nach außen drücken. Den Fliehkräften versucht der Algorithmus ebenfalls entgegen zu wirken und korrigiert. Dieses Verhalten begünstigt in Kurvenfahrten das Schwingen. Wenn das Stück hinter der Kurve betrachtet wird, fällt auf, dass das Fahrzeug hier ebenfalls Korrekturen vornimmt. Allerdings ist die Korrektur das Ergebnis des falsch eingestellten Offsets während der Kurvenfahrt.

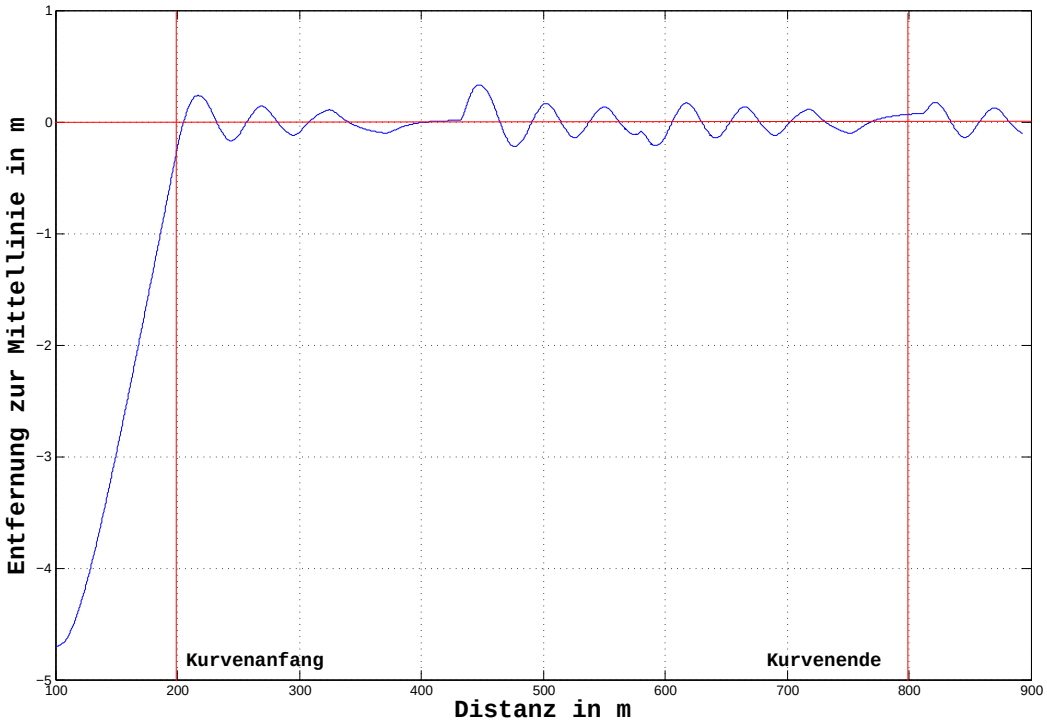

Abbildung 16: Bisektionaler Algorithmus mit Intervallschritt 0.001

Wird die Abbildung 17 betrachtet, so fällt auf, dass bei einer weiteren Halbierung des Intervallschrittes die Strecke zum Ausgleichen des Offsets sich um ca. 100 Meter verlängert, aber auch gleichzeitig deutlich weniger schwingt. Die Schwingungen, die innerhalb des Kurvenabschnittes zu sehen sind, sind die fälschlicherweise durchgeführten Korrekturen des Algorithmus. Dabei kann man am Kurvenende auch wieder sehr gut erkennen, dass der Algorithmus diese nach der Kurvenfahrt erneut korrigieren muss. Nach ca. 80 Metern ist die Korrektur durchgeführt und das Fahrzeug kann seine Spur wieder gerade halten, ohne zu schwingen. Im Gegensatz zu Abbildung 15, wo klar zu erkennen ist, dass nach 100 Metern noch immer leichte Schwingungen vorhanden sind und ausgeglichen werden müssen.

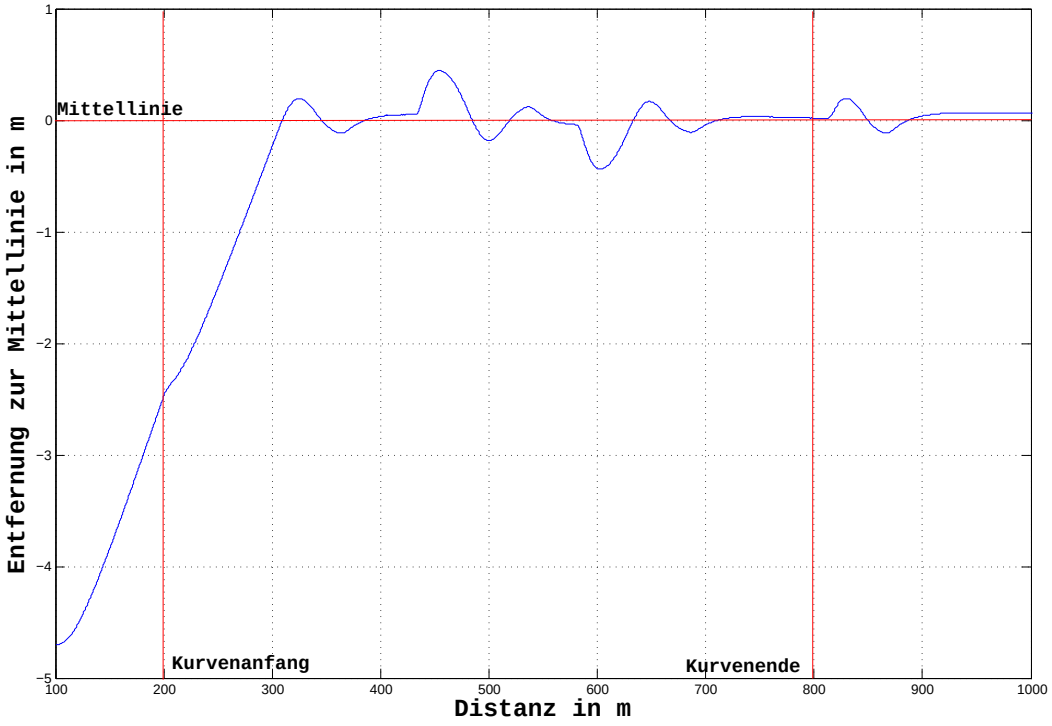

Abbildung 17: Bisektionaler Algorithmus mit Intervallschritt 0.0005

6.3 Test des messenden Algorithmus

Bei dem Test des messenden Algorithmus wird ebenso verfahren, wie bei dem Test des bisektionalen Algorithmus. Es wird die Entfernung zur Mittellinie gemessen sowie die zurückgelegte Distanz seitdem die Startlinie überfahren wurde. Dabei muss ebenfalls beachtet werden, dass von Meter 100 bis ca 720 eine Kurvenfahrt bzw. Meter 200 bis 800 im Test des bisektionalen Algorithmus stattfindet, die das Fahrzeug durch die Fliehkraft nach außen drückt. Durch diese Kraft entsteht in den Diagrammen eine leichte Schwingung um die Mittellinie. Je nach Art der Implementierung kann der Algorithmus darauf reagieren und fälschlicherweise Korrekturen durchführen, die nach der Kurvenfahrt ebenfalls wieder korrigiert werden müssen.

6.3.1 Ergebnis der Funktion measure()

In der Abbildung 18 ist auf der x-Achse der Lenkwinkel in Prozent aufgetragen, auf der y-Achse die Entfernung zur Mittellinie in Zentimetern. Der Messbereich beträgt, wie man der Abbildung entnehmen kann, von -40 bis + 40 Prozent. Dabei ist der Bereich von -40 bis

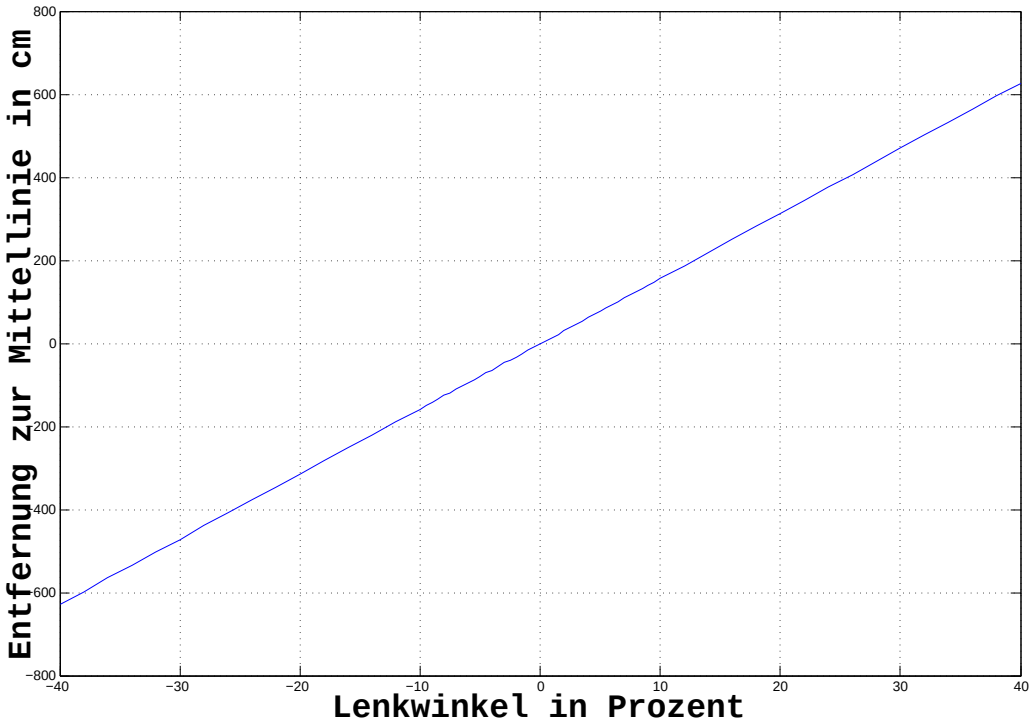

Abbildung 18: Ergebnis der Funktion *measure()*

-10 Prozent in zwei Prozent Intervallen abgedeckt, der Bereich von -10 bis +10 Prozent in 0.5 Prozentschritten und von +10 bis +40 Prozent wieder in zwei Prozent Intervallen.

Wird diese Abbildung betrachtet, so fällt sehr schnell auf, dass der Verlauf des Verhältnisses von Lenkwinkel zu Abweichung sehr linear verläuft. An einigen Stellen gibt es kleinere Schwankungen, allerdings sind diese nicht sehr gravierend. Aus diesem Grund kann bei der Implementierung der Erkennung und Korrektur auf Basis der Polynomapproximation ein Polynom ersten Grades verwendet werden.

6.3.2 Test der Funktion detectAndCorrect()

In Abbildung 19 ist zu erkennen, dass nach relativ kurzer Distanz (genauer ca. 15m) der Algorithmus eine Korrektur vornimmt. Danach benötigt das Lenksystem ca. 85m zur fast vollständigen Korrektur. Diese Wegstrecke von der Erkennung bis zur fast vollständigen Korrektur von 100 Metern ist gleich der Strecke, die der bisektionale Algorithmus benötigt, wenn dieser mit einem Änderungsintervall von 0,1 Prozent pro Aufruf arbeitet.

Während der Kurvenfahrt könnte der Gedanke aufkommen, dass der Algorithmus bei un-

gefähr der Hälfte der Kurve eine weitere Anpassung vornimmt. Dies ist allerdings nicht der Fall, da der Algorithmus bei diesem Test während der Kurvenfahrt ausgeschaltet ist, aufgrund der Schwingung im Test des bisektionalen Algorithmus. Dieser Ausbruch ist da durch entstanden, weil das Fahrzeug zuerst eine Links-, danach eine Rechtskurve und zum Schluss wieder eine Linkskurve fährt. Wenn die Kurvenfahrt zu Ende ist, muss der Algorithmus noch zwei kleinere Anpassungen vornehmen und fährt dann ziemlich exakt auf der geraden Strecke.

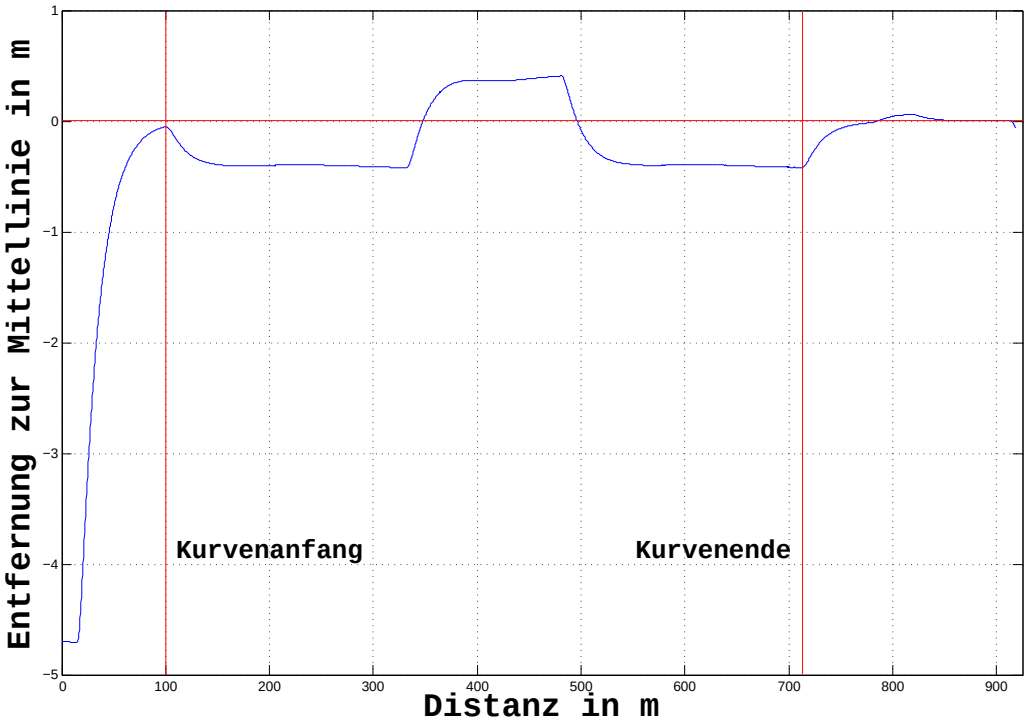

Abbildung 19: Ergebnis der Funktion *detectAndCorrect()* bei einer zurückzulegenden Distanz von 1 Metern mit Stabilisator

Wird der Stabilisator ausgeschaltet, d.h. der Algorithmus überprüft nicht mehr, ob sich die Entfernung zur Mittellinie während des letzten Aufrufes nicht mehr als um einen Millimeter geändert hat, dann ergibt sich ein Ablauf wie in Abbildung 20. In dieser Abbildung ist deutlich zu erkennen, dass das Fahrzeug enorm schwingt, bei dem Versuch eine Korrektur durchzuführen, während der Kurvenfahrt ist der Algorithmus deaktiviert, da hier exakt gleiches Verhalten wie in Abbildung 19 zu erkennen ist. Nach der Kurvenfahrt greift der Algorithmus erneut ein und ist nach fast 300 Metern immer noch nicht fertig mit der Korrektur.

Der Grund für diese enormen Schwingungen liegt darin begründet, dass zu viele Korrekturen mit zu großen Werten durchgeführt werden in einer bestimmten Zeit. Während der bisektionale Algorithmus pro Aufruf den Offset nur um einen sehr geringen Wert von 0,1 Prozent ändert, arbeitet dieser Algorithmus mit einem Änderungswert von bis zu 40 Prozent bei einem Aufruf. Wenn nun der Algorithmus zu oft aufgerufen wird, summieren sich diese Werte sehr schnell zu einem sehr großen Offset und das Fahrzeug kann bei einer zu kurzen Periode auch sehr schnell von der Fahrbahn abkommen und die Korrektur nicht mehr richtig korrigieren.

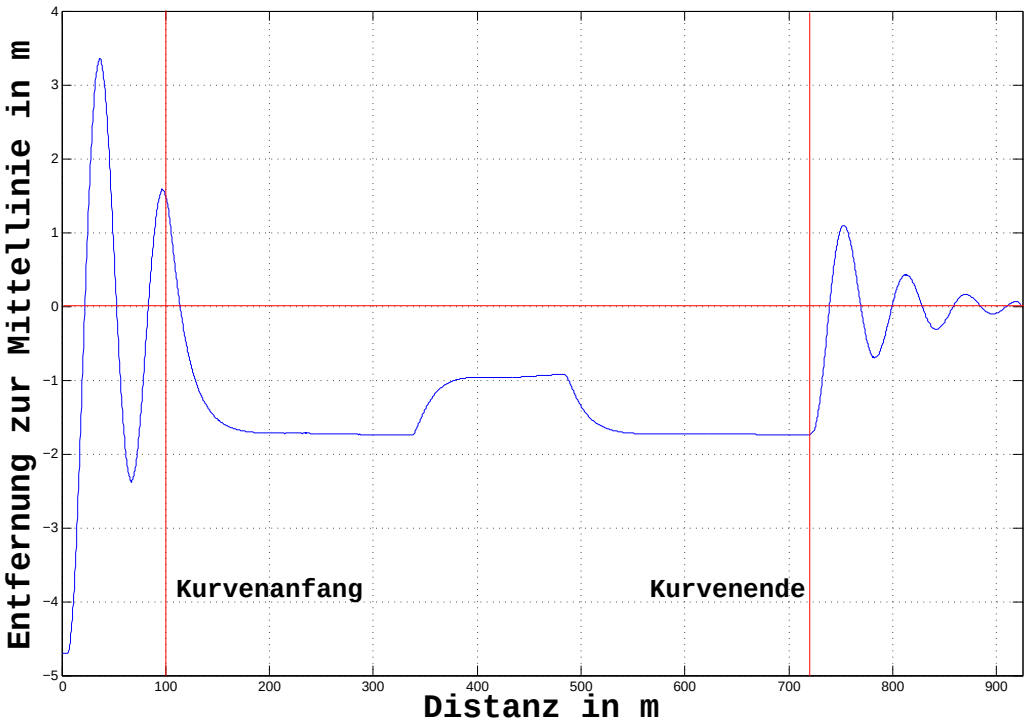

Abbildung 20: Ergebnis der Funktion *detectAndCorrect()* bei einer zurückzulegenden Distanz von 5 Metern ohne Stabilisator

Wird die zurückzulegende Distanz auf 10 Meter erhöht, kann man in der Abbildung 21 sehr gut erkennen, dass sich das Verhalten enorm verbessert und nach gut 200 Metern eine vollständige Korrektur durchgeführt ist und das Fahrzeug stabil auf der gerade Strecke ohne seitliche Versetzung der Ideallinie fahren kann. Allerdings ist dieser Wert nicht so optimal wie der des ersten Testes des Algorithmus.

Man kann dieses Ergebnis noch verbessern, indem man die zurückzulegende Distanz weiter

erhöht, allerdings wird festgestellt werden, dass eine Distanz von 15 Metern sehr gut ist und man dieses Ergebnis nicht bedeutend verbessern kann.

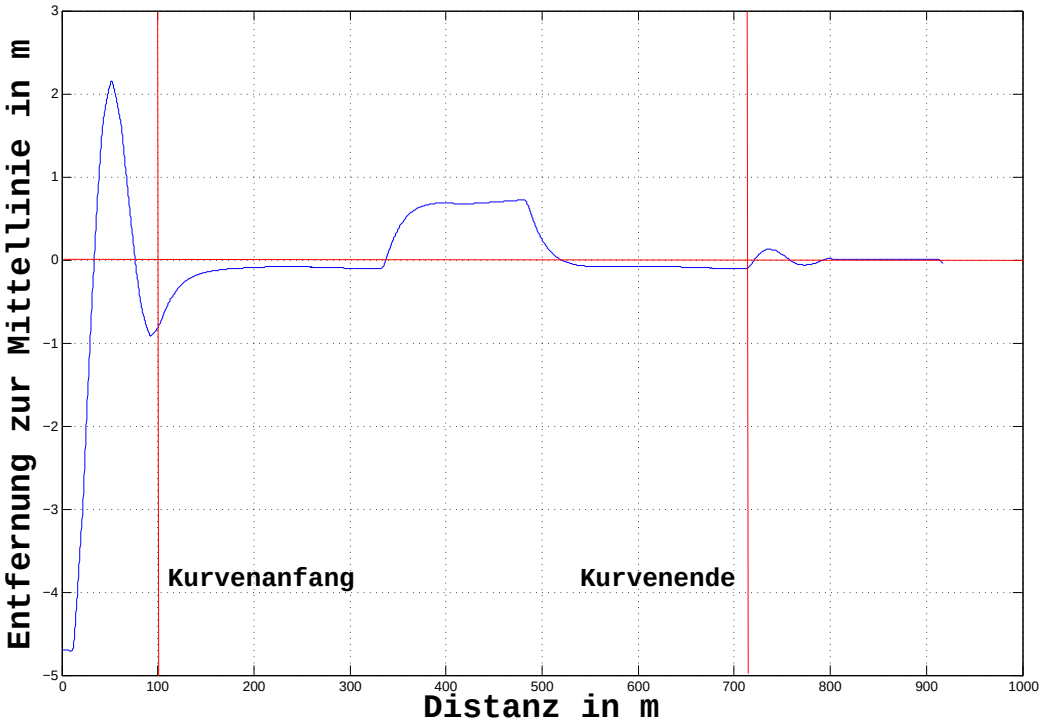

Abbildung 21: Ergebnis der Funktion *detectAndCorrectPoly()* bei einer zurückzulegenden Distanz von 10 Metern ohne Stabilisator

6.3.3 Test der Funktion detectAndCorrectPoly()

In der Abbildung 22 wird eine Änderung des Offsets alle 0.5 Sekunden durchgeführt. Dabei treten ebenfalls große Schwingungen auf, wie es bereits in Abbildung 20 der Fall ist. Der Grund ist in diesem Falle der Gleiche. Es werden zu oft zu große Änderungen am Offset vorgenommen, so dass das Fahrzeug anfängt zu schwingen und der Algorithmus Probleme bekommt, diesen Offset wieder auf eine richtige Größe zu bringen.

Beträgt die Zeit pro Änderung nicht mehr wie 0.5 Sekunden, schafft der Algorithmus diese Änderungen noch vorzunehmen und kann nach ca. 250 Metern fast stabil fahren. Allerdings wird das Fahrzeug mit dieser Einstellung wahrscheinlich immer leicht schwingen und keine richtige stabile Fahrweise erreichen.

Außerdem ist in den nachfolgenden Test keine Deaktivierung des Algorithmus während einer Kurvenfahrt vorgenommen worden. Dies ist ebenfalls ein Grund, weshalb sich der Weg zum Erreichen eines stabilen Zustandes verlängert bzw. nach jeder Kurvenfahrt eine erneute Korrektur vorgenommen werden muss.

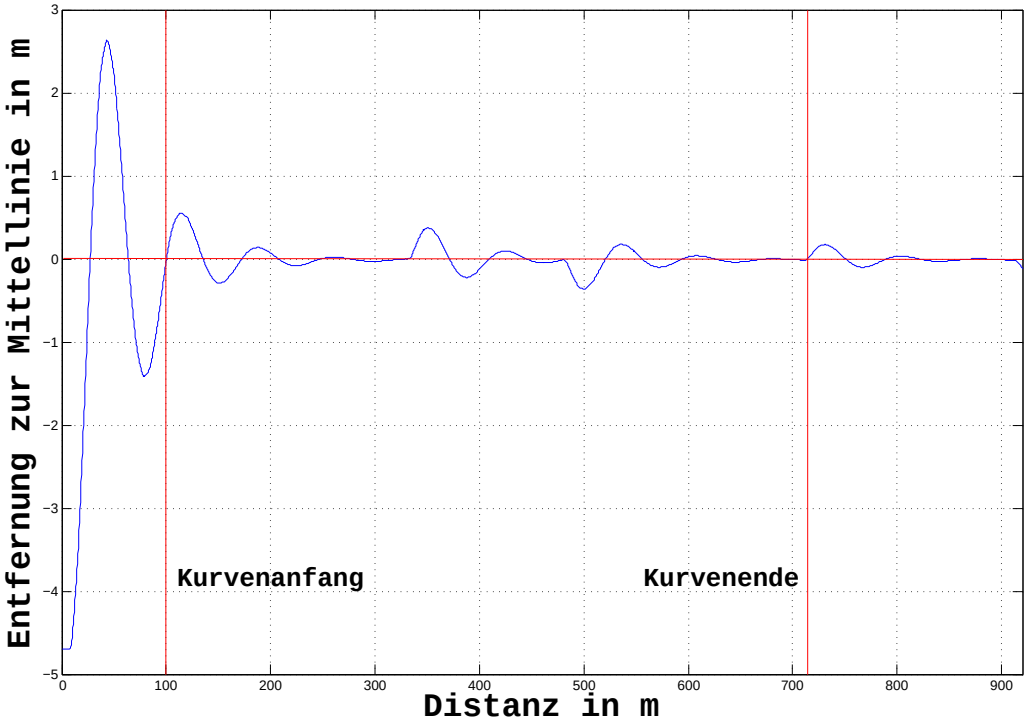

Abbildung 22: Ergebnis der Funktion *detectAndCorrectPoly()* bei einer Zeit pro Änderung von 0.5s ohne Stabilisator

Wird die Zeit pro Änderung vergrößert, so verringert sich die Schwingung des Fahrzeuges, hört allerdings auch nicht komplett auf. In Abbildung 23 wurde die Zeit auf eine Sekunde erhöht. In Abbildung 24 auf 1.5 Sekunden. Dabei fällt allerdings auch auf, dass sich nur die Schwingung zu Begin der Korrektur verändert, also in den ersten 100 Metern. Die Kurvenfahrt sieht bei allen drei Tests fast identisch aus, nur bei der Abbildung 22 sieht man, dass der Algorithmus bei Einfahrt in die Kurve stärkere Korrekturen vornimmt.

Ein weiterer Punkt, der bei direktem Vergleich aller drei Abbildungen auffällt, ist, dass alle drei Tests mit einer Korrektur am Kurvenausgang beendet werden. Dabei spielt die Zeit pro Änderung in sofern eine Rolle, wann keine Korrekturen mehr vorgenommen werden. Je größer diese Zeit ist, desto schneller läuft der Algorithmus stabil.

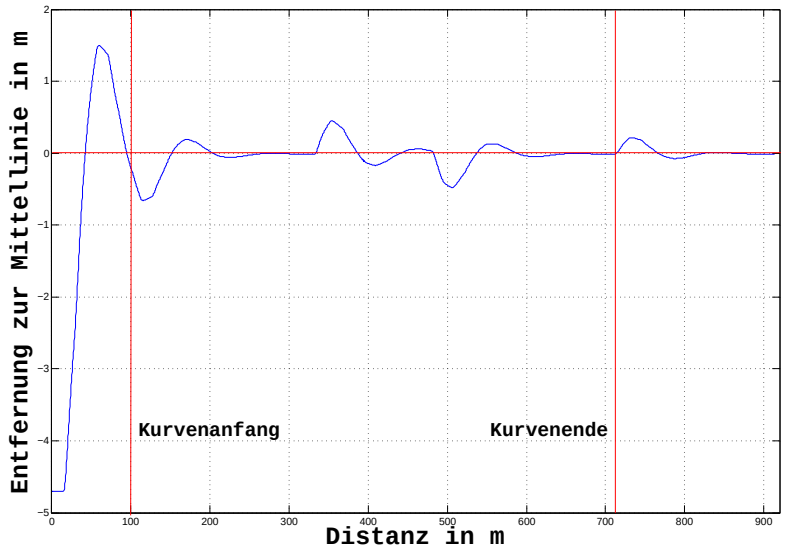

Abbildung 23: Ergebnis der Funktion *detectAndCorrectPoly()* bei einer Zeit pro Änderung von 1.0s ohne Stabilisator

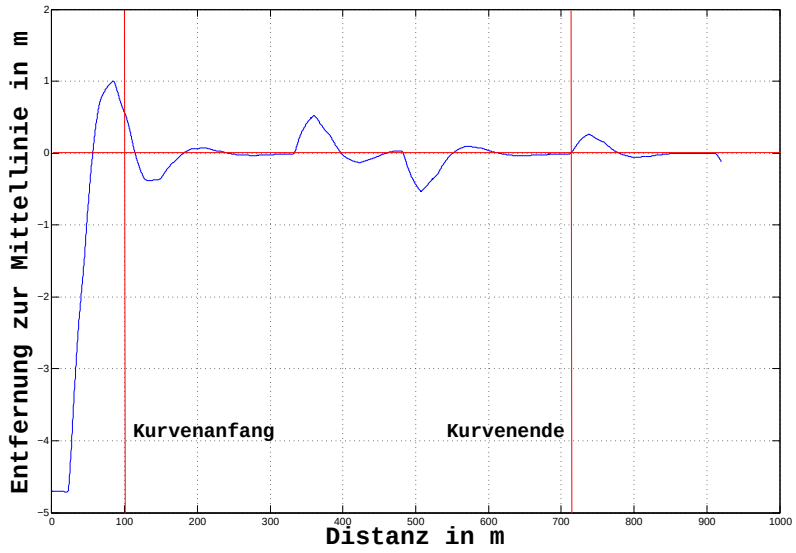

Abbildung 24: Ergebnis der Funktion *detectAndCorrectPoly()* bei einer Zeit pro Änderung von 1.5s ohne Stabilisator

Wird die Zeit noch weiter erhöht, so verbessert sich das Ergebnis nochmals deutlich. In der

Abbildung 25 wird die Zeit pro Änderung auf 3.0 Sekunden erhöht. Die Implementierung mittels Polynomapproximation schafft eine fast vollständige Korrektur innerhalb von 100 Metern und auch bei dem Verlassen einer Kurvenstrecke schafft diese Implementierung nach einer weiteren Korrektur eine sehr stabile Fahrweise.

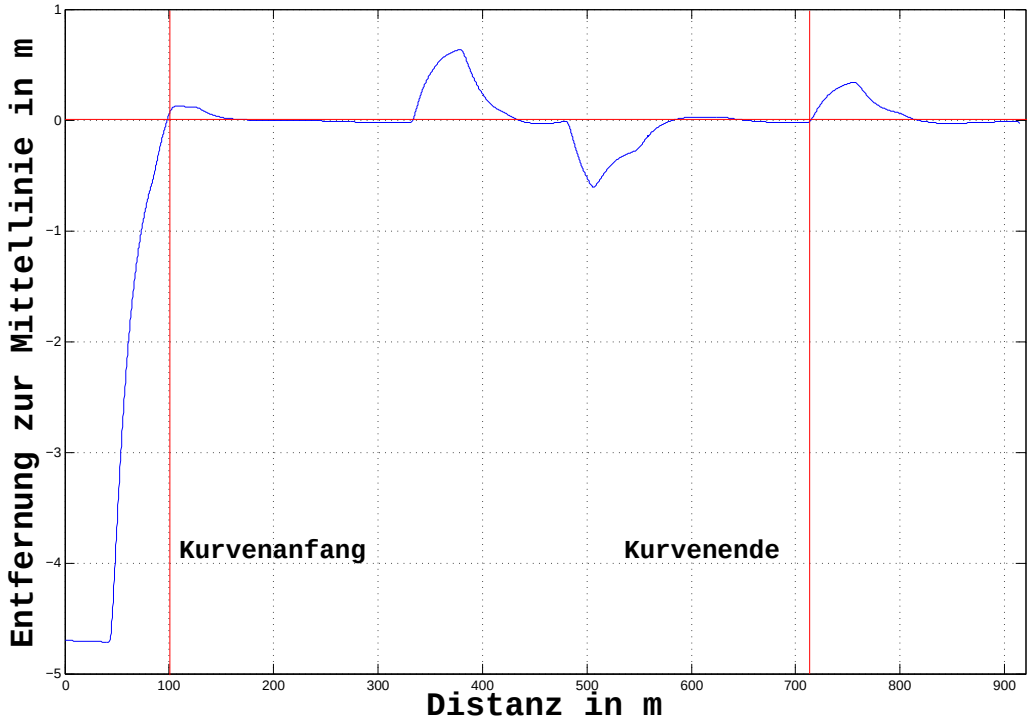

Abbildung 25: Ergebnis der Funktion *detectAndCorrectPoly()* bei einer Zeit pro Änderung von 3.0s ohne Stabilisator

7 Portierung der Algorithmen auf die FAUST Architektur

Ein Grund, warum T.O.R.C.S. als Entwicklungs- und Simulationssoftware eingesetzt wird, ist der, dass die entwickelte Software fast identisch auch auf der Software-Architektur des Projektes FAUST läuft und sich sehr leicht und mit wenig Anpassung implementieren lässt. So ist es möglich mit wenig Aufwand eine neue Task in den FAUSTplugins zu erstellen und man muss die entwickelten Funktionen und Methoden einfach einfügen. Die einzigen Ersetzungen, die vorgenommen werden müssen, sind die T.O.R.C.S.-spezifischen Fahrzeugvariablen durch die des Projektes FAUST zu ersetzen.

Es ist eine Task names *CorrectSteering* erstellt worden, die die Funktionen *double bisectionalAlgorithm(double distanceToMid, double middle)*, *double measure(double distance, double middle, double distanceToMid)*, *double detectAndCorrect(double distance, double middle, double distanceToMid)*, *double detectAndCorrectPoly(double distanceToMid)* enthält. Darüber hinaus enthält die Task eine Variable names *carInCurve*, die vom Typ *bool* ist und den Zustand speichert, ob sich das Fahrzeug in einer Kurve befindet oder nicht. Es gibt 13 Parameter, die über die Weboberfläche vom FAUSTcore eingestellt werden können.

Für die Fahrzeugspezifikation existiert der Parameter *car* mit dem Typ *int*. Um die Priorität der Task festzulegen, ist der Parameter *priority* vorhanden, der ebenfalls den Datentyp *int* besitzt. Um Einstellungen am bisektionalen Algorithmus vorzunehmen, existieren die Parameter *bisecAlgo_distToMid* und *bisecAlgo_intervall*. Beide Parameter sind vom Typ *double*. *bisecAlgo_distToMid* definiert den Idealbereich um die Mittellinie herum und *bisecAlgo_intervall* gibt an, welche Änderung pro Aufruf in Prozent erfolgen darf. Zur Einstellung des messenden Algorithmus werden die Parameter *learnAlgo_driveDistance*, *learnAlgo_intervallMeasure*, *learnAlgo_precisionMeasure*, *learnAlgo_fineMeasureStart*, *learnAlgo_fineMeasureEnd*, *learnAlgo_measureStart*, *learnAlgo_measureEnd* und *learnAlgo_stabi* sowie *learnAlgo_disToMid* und *learnAlgo_timeToWait* genutzt. Alle Parameter sind vom Datentyp *double* und haben dabei die gleiche Bedeutung wie die Defines aus Kapitel 5.3.2 und 5.3.3.

7.1 Der Shared Pointer SteeringAngleData

Für die richtige Funktionsweise der Algorithmen zur Korrektur des Lenkwinkels wird der berechnete Lenkwinkel des Pure Pursuit Algorithmus benötigt. Um diesen zu erlangen, wird ein Shared Pointer (Beman Dawes) mit dem Namen *SteeringAngleData* angelegt, der als einzige Variable *steeringAngle* mit dem Datentyp *int* besitzt. Der Inhalt dieser Variable kann durch den Aufruf der Funktion *getSteeringAngle()* bekommen werden, die den Wert ebenfalls als *int* zurückliefert.

Dabei wird der Shared Pointer in der Task *Steering Control* nach Berechnung des Lenkwinkels mit dem Befehl *SteeringAngleDataPtr steerAngData = SteeringAngleDataPtr(new SteeringAngleData((int) (steeringAngle + 0.5)));* erzeugt. Die Variable *steeringAngle* wird von der Funktion *purePursuit()* beschrieben und mit 0.5 addiert, damit eine richtige Rundung des Wertes stattfindet. Anschließend wird der erzeugte Shared Pointer den restlichen Tasks bekannt gemacht, in dem der Befehl *DataContainer<SteeringAngleData>::instance().addData(steerAngData);* aufgerufen wird.

Der Shared Pointer kann mit dem Befehl *SteeringAngleDataPtr steerAngleData = DataContainer<SteeringAngleData>::instance().getData();* geholt werden und mittels *steerAngleData->getSteeringAngle()* bekommt man den Lenkwinkel, den die Task *SteeringControl* errechnet hat.

7.2 Die execute()-Funktion der Task CorrectSteering

Da die Funktion zur Steuerung des Codes komplexer ist als unter T.O.R.C.S. wird die Implementierung erläutert. Die Funktion *execute()* hat die Aufgabe, die Auswahl des Algorithmus zu steuern, die übergebenen Daten von der Task *PolarisV2* auszuwerten und die benötigten Daten zu extrahieren sowie den mittels Pure Pursuit berechneten und übergebenen Lenkwinkel der Task *SteeringControl* auszuwerten und den neuen Lenkwinkel zu setzen.

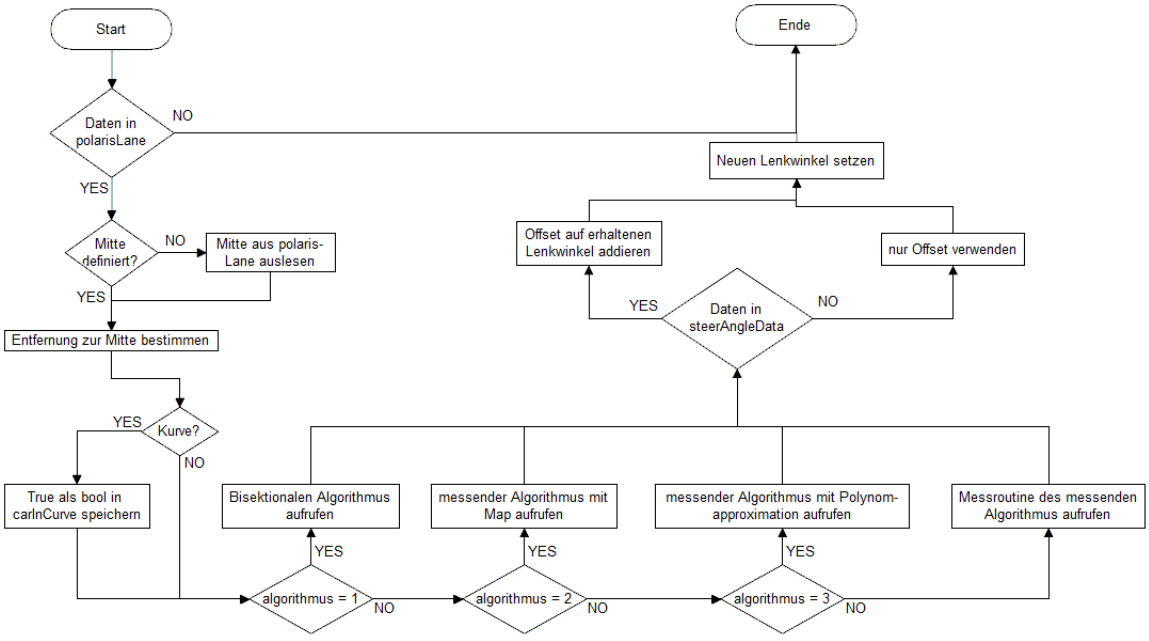

Abbildung 26: Ablaufdiagramm der Funktion *execute()*

Die Funktion *execute()* besitzt fünf Variablen, drei Shared Pointer und ein Singleton. Die

Variablen sind *right_Invalid* und *left_Invalid*, die beide vom Datentyp *int* sind und für die Gültigkeit des Polaris Polynoms stehen. Die Variable *middle* ist vom Datentyp *static double* und speichert die Mittellinie, mit der die Entfernung verglichen wird. *distanceToMiddle* enthält die aktuelle Distanz vom rechten Fahrbahnrand und ist vom Typ *double*. In *steering* wird der von den Korrekturalgorithmen berechnete Offset gespeichert und ist vom Typ *int*.

Als Shared Pointer wird *SensorValuesPtr* mit dem Namen *sValues*, *PolarisLanePtr* mit dem Namen *polarisLane* und *SteeringAngleDataPtr* mit Namen *steerAngleData* verwendet. Dabei stellt *sValues* verschiedene Sensorenwerte zur Verfügung. Es wird aber nur die gefahrene Distanz abgerufen. *polarisLane* stellt die Polynome zur Verfügung, die die Fahrbahn abbilden und haben dabei einen Grad von drei. Die Konstante *d* des Polynoms wird zur Messung der Entfernung vom Fahrbahnrand verwendet. *steerAngleData* stellt den Lenkwinkel zur Verfügung, der von der Task *SteeringControl* berechnet wird.

Das Singleton *ActuatorValues* mit dem Namen *engine* steuert die Aktorik des Fahrzeuges und wird zur Einstellung des Lenkwinkels verwendet.

Als Erstes wird in der Funktion geprüft, ob Polaris wirklich Polynome erzeugt hat und diese im Shared Pointer zum Abrufen bereit sind. Diese Information wird in *right_Invalid* und *left_Invalid* gespeichert. Ist dies nicht der Fall, werden zwar die restlichen Aufgaben abgearbeitet, laufen aber nicht korrekt. Liegen die Daten vor, wird zu nächst überprüft, ob schon die Mittellinie *middle* definiert wurde. Sollte dies nicht der Fall sein, wird sie mit der Konstanten *d* des Polynoms beschrieben. Dabei bezeichnet diese Mittellinie *middle* die optimale Fahrspur auf der Fahrbahn. Die Konstante *d* des Polynoms wird bei jedem neuen Aufruf in die Variable *distanceToMiddle* geschrieben. Außerdem wird bei jedem neuen Aufruf überprüft, ob sich das Fahrzeug in einer Kurve befindet, indem beim zweiten Koeffizienten des Polynoms kontrolliert wird, ob sich der Koeffizient außerhalb des Bereiches von -0.001 bis +0.001 befindet. Wenn es der Fall ist, bekommt die Variable *carInCurve* den Wert *true*, ansonsten *false*.

Enthält der Shared Pointer *sValues* Daten, werden die restlichen Aufgaben abgearbeitet. Dabei wird als Erstes mittels einer Switch-Case-Struktur überprüft, welcher Algorithmus angewendet werden soll. Enthält der Parameter *algorithmus* eine eins, dann wird der bisektionale Algorithmus ausgeführt. Bei einer zwei wird *detectAndCorrect()* vom messenden Algorithmus ausgeführt. Ist eine drei enthalten, soll die Funktion *detectAndCorrectPoly()* vom messenden Algorithmus ausgeführt werden und bei einer vier werden die Messdaten für den messenden Algorithmus erhoben. Alle Funktionen geben einen Offset zurück, der in der Variable *steering* gespeichert wird. Wenn die Task *Steering Control* einen Lenkwinkel berechnet hat und dieser im Shared Pointer *steerAngleData* bereitsteht, wird er zu *steering* hinzuaddiert und danach mittels der Übergabe an *engine* an den Servomotor gesendet.

7.3 Vorgenommene Änderungen an der Funktion bisectionalAlgorithm()

Bei der Funktion *bisectionalAlgorithm()* wurde ein Rückgabewert hinzugefügt sowie die Übergabeparameter verändert. Die Funktion gibt den aktuellen Offset als *double* zurück und erwartet als Übergabeparameter die Entfernung von der Fahrbahnseite (*double distanceToMid*) sowie der Mittellinie (*double middle*). Da die Werte für die Lenkung in T.O.R.C.S. anders sind als in der FAUST Software Architektur, mussten bei den mathematischen Operationen die Vorzeichen vertauscht werden.

7.4 Vorgenommene Änderungen an der Funktion measure()

Auch die Funktion *measure()* erhält einen Rückgabewert vom Typ *double*, womit der aktuelle Offset zurückgegeben wird. Als Übergabeparameter erwartet die Funktion drei Werte: die aktuelle Distanz (*double distance*), die Entfernung zur Mittellinie (*double distanceToMid*) sowie die Mittellinie (*double middle*) selbst. Außerdem wird in der Funktion selbst noch auf die Task Variable *carInCurve* zurückgegriffen, um zu wissen wann sich das Fahrzeug in einer Kurve befindet.

7.5 Vorgenommene Änderungen an der Funktion detectAndCorrect()

Genau wie die Funktion *measure()* erhält die Funktion *detectAndCorrect()* einen Rückgabewert vom Typ *double*, um den aktuellen Offset zurückzugeben. Es werden dieselben drei Übergabeparameter wie bei der Funktion *measure()* erwartet und die Funktion *detectAndCorrect()* greift ebenfalls auf die Task Variable *carInCurve* zurück, um Kurven zu detektieren.

7.6 Vorgenommene Änderungen an der Funktion detectAndCorrectPoly()

Die Funktion *detectAndCorrectPoly()* erwartet als Übergabeparameter die Entfernung zum Fahrbahnrand (*double distanceToMid*) und gibt mit dem Datentyp *double* den Offset zurück. Die vergangene Zeit wird mit der Systemfunktion *gettimeofday()* gemessen, in die Variable *timeSinceEpoche*, welche vom Typ *double* ist, geschrieben und beim Initialisieren der Funktion *startTime* geschrieben, wo sie nur noch reingeschrieben wird, wenn eine Änderung erfolgt ist.

8 Test der Algorithmen auf FAUST Architektur

Die Tests werden auf dem Fahrzeug Onyx aus dem Projekt FAUST durchgeführt. Dabei haben die Algorithmen mit einer Ausführungszeit von 25 Millisekunden gearbeitet, während eine Änderung an der Aktorik ungefähr 200 Millisekunden dauert. Das Fahrzeug ist mit einer Geschwindigkeit von 20 Prozent der maximal möglichen Geschwindigkeit gefahren, sowohl in Kurven als auch auf geraden Strecken.

8.1 Test der Entfernungsmessung zur Mittellinie anhand der Polynome

Bei diesem Test wird von der rechten Fahrbahnseite beginnend die Distanz des linken als auch rechten Polynoms von Polaris zum Fahrbahnrand bzw. mittleren Leitlinie gemessen.

reale Entfernung	**gemessen linkes Polynom**	**gemessen rechtes Polynom**
-2cm	-65.2cm	1.5cm
0cm	-64.5cm	2.2cm
2cm	-32.3cm	4.4 / -8.5cm
4cm	-31.2cm	4.1cm
6cm	-69.5cm	4.5cm
8cm	-27.6cm	6.3cm
10cm	-24.6cm	7.8cm
12cm	-23.9cm	9.2cm
14cm	-22.4cm	10.7cm
16cm	-21.7cm	11.9cm
18cm	-19.1cm	16.0cm
20cm	-18.4cm	16.7cm

Tabelle 1: Messung der Entfernung zur Mittellinie mittels Polaris Teil 1

Betrachtet man den ersten Teil der Messung in Tabelle 1, so fällt auf, dass das rechte Polynom wesentlich genauer arbeitet als das linke Polynom. Der Ausreißer bei der Messung des rechten Polynoms bei einer Entfernung von 2 Zentimeter kommt dadurch zustande, weil Polaris teilweise eine falsche Erkennung der Fahrbahnlinien hat.

Wird nun der zweite Abschnitt in Tabelle 2 betrachtet, kann festgestellt werden, dass keines der beiden Polynom sehr exakte Werte liefert bezüglich der Entfernung zum rechten Fahrbahnrand bzw. der mittleren Leitlinie. Allerdings arbeitet das rechte Polynom um einiges besser als das linke Polynom. Aus diesem Grund wird in den Algorithmen zur Entfernungsmessung zum Fahrbahnrand das rechte Polynom herangezogen.

reale Entfernung	gemessen linkes Polynom	gemessen rechtes Polynom
22cm	-16.9cm	18.3cm
24cm	-14.9cm	19.5cm
26cm	-12.7cm	20.6cm
28cm	-11.5cm	22.1cm
30cm	-9.8cm	23.5cm
32cm	-8.2cm	24.9cm
34cm	-6.5cm	26.4cm
36cm	-5.3cm	27.9cm
38cm	-5.7cm	29.1cm
40cm	-6.3cm	30.5cm
42cm	-6.4cm	31.8cm

Tabelle 2: Messung der Entfernung zur Mittellinie mittels Polaris Teil 2

8.2 Test der portierten Funktion bisectionalAlgorithm()

In Abbildung 27 kann man erkennen, dass der bisektionale Algorithmus auf dem Fahrzeug des FAUST Projektes funktioniert. Die Bereiche, die im roten Rahmen mit einem roten Querstrich markiert sind, sollen nicht gewertet werden. An diesen Stellen hat Polaris die Fahrbahnlinien verloren und musste per Hand wieder auf die Fahrbahn gesteuert werden. Während einer Fahrt ohne den Algorithmus ist das Fahrzeug deutlich auf der rechten Fahrbahnmarkierung gefahren und hatte dabei eine Entfernung zum rechten Fahrbahnrand laut Polaris von einem bis vier Zentimeter.

Die Funktion benötigt einen gewissen Weg, wie bereits im Test unter T.O.R.C.S (Kapitel 6.2) gezeigt wurde, um den richtigen Offset zu finden. Der Weg liegt in diesem Test bei ca. 25 Zentimetern und der optimale Offset bei ca -16 Prozent. Mit diesem Offset ist es dem Fahrzeug möglich, fast immer in der Mitte der Fahrbahn zu bleiben.

Im dritten Abschnitt sieht man, dass die Entfernung zur Mittellinie zunächst konstant steigt, nachdem Polaris die Fahrspur verloren hat. Während dieser Zeit hat der bisektionale Algorithmus allerdings eine enorme Verstellung des Offsets bewirkt, so dass eine erneute Korrektur durchgeführt werden muss, damit anschließend wieder die Fahrspur innerhalb der Fahrbahn gehalten werden kann.

Es treten manchmal auch enorme Sprünge in der Entfernung zum rechten Fahrbahnrand auf, wo der Wert von z.B. 20 Zentimeter auf null Zentimeter im nächsten Augenblick fällt und kurz darauf wieder auf 20 Zentimeter steigt. Dieses Phänomen ist ebenfalls Polaris geschuldet, dass kurzzeitig keine Linien erkennen konnte.

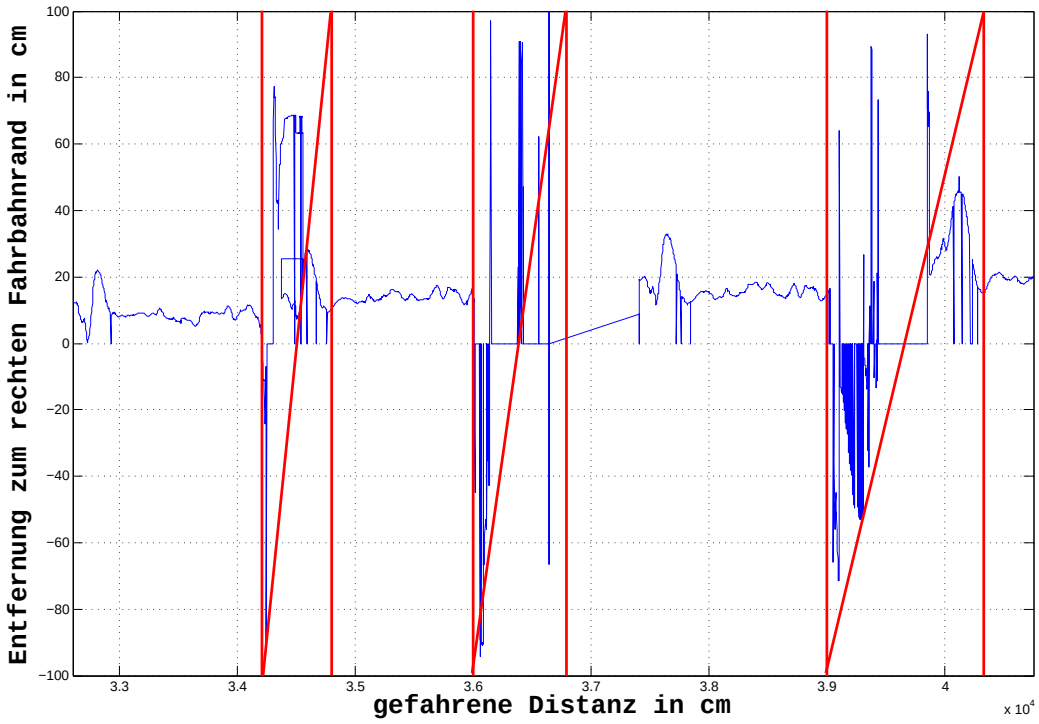

Abbildung 27: Ergebnis der Funktion *bisectionalAlgorithm()* bei einer Änderungsrate von 0.015 Prozent pro Aufruf

8.3 Test der portierten Funktion measure()

Der Test der Funktion *measure()* zeigt in Abbildung 28, dass der messende Algorithmus zurzeit nicht auf den FAUST Fahrzeugen implementiert werden kann. Durch die zum Teil ungenaue, aber auch fehlerhafte Messung der Entfernung zum rechten Fahrbahnrand mittels Polaris, können keine stabilen Werte erzielt werden, selbst wenn das Fahrzeug zum Zeitpunkt der Messung stabil geradeaus gefahren ist und auch den Offset richtig eingestellt hat.

Da in diesem Test, der mehrmals durchgeführt wurde und niemals ein gutes Ergebnis lieferte trotz des Versuches einer langsameren Geschwindigkeit, keine zufriedenstellenden Werte erzeugt werden können, ist ein vollständiger Test der Portierung der Funktionen *detectAndCorrect()* und *detectAndCorrect()* nicht möglich gewesen.

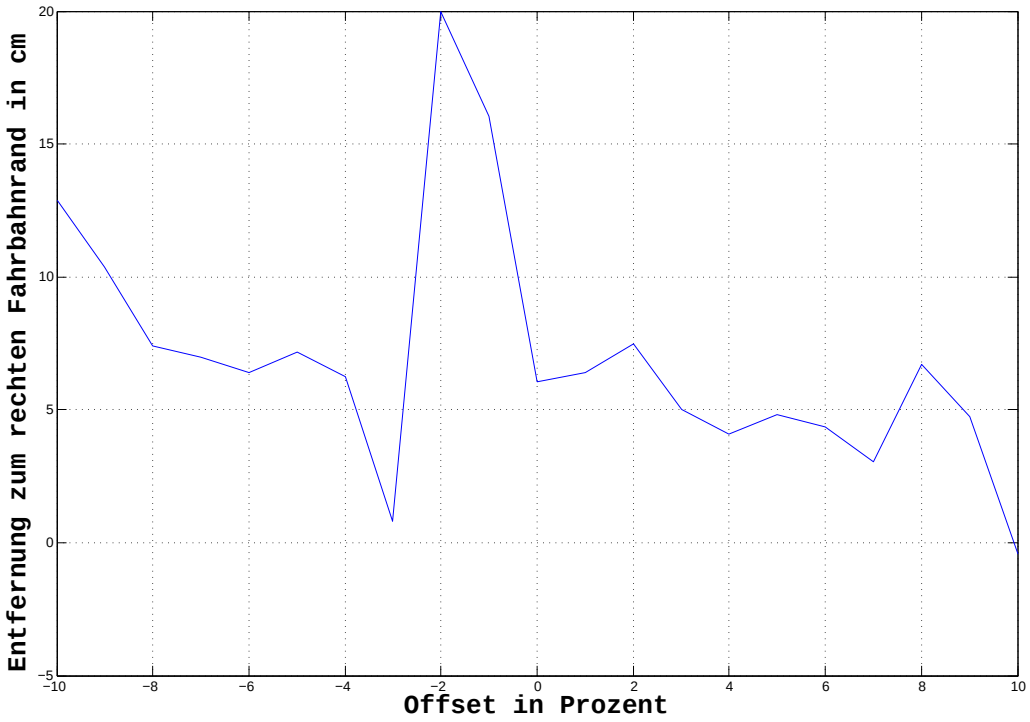

Abbildung 28: Ergebnis der Funktion *measure()*

9 Fazit

9.1 T.O.R.C.S. als Simulator

T.O.R.C.S. ist durchaus ein sehr mächtiges Programm für die Simulation von Fahreigenschaften und darauf aufbauenden Algorithmen. T.O.R.C.S. stellt dem Simulator eine Vielfalt an Daten über das Fahrzeug und dessen Umgebung zur Verfügung. Allerdings hat T.O.R.C.S. auch einige Nachteile im Bezug auf physikalische Fahrzeugplattformen. Während im Simulator alle Daten sehr einfach per Variablen abgefragt werden können, müssen diese im physikalischen Fahrzeug durch Kameras und Umgebungssensoren erfasst werden. Diese sind teilweise nicht immer präzise oder es müssen zusätzliche Algorithmen implementiert werden, die über das Kamerabild Distanzmessungen der Fahrlinien zum vorherigen Bild durchführen.

Wenn bestehende Algorithmen von physikalischen Fahrzeugen in T.O.R.C.S. implementiert werden, um diese weiter zu optimieren ohne die Abhängigkeit von Akkus, einer gut eingestellten Lenkung oder einer gut arbeiten Sensorik, gibt es ebenfalls Probleme. Zum einen muss das gesamte Programm auf die T.O.R.C.S. spezifischen Schnittstellen angepasst werden bzw. müssen kleine Funktionen entworfen werden, die diese Kompatibilität ermöglicht, zum Anderen kann es erforderlich sein, dass Parameter erst einmal angepasst werden müssen, damit der Algorithmus auch unter T.O.R.C.S. einwandfrei läuft.

Ist aber diese Arbeit einmal erledigt, so bietet T.O.R.C.S. eine gute Alternative zur Entwicklung und Optimierung der Algorithmen auf physikalischen Fahrzeugen, da T.O.R.C.S. jeden Unfall und Rempler mit der Leitplanke verzeiht und man sehr schnell und einfach Anpassungen am Programmcode vornehmen kann, ohne besondere Rücksicht zu nehmen auf die Einschaltreihenfolge der Komponenten sowie der sich daraus ergebenden richtigen Initialisierung der Sensorik.

9.2 Die Algorithmen unter T.O.R.C.S.

9.2.1 Die Zusatzfunktionen

Wie der Test in Kapitel 6.1 gezeigt hat, funktionieren die selbst implementierten Funktionen zur Distanzmessung und zum Halten von Geschwindigkeiten sehr gut. Auch die Implementierung der Schaltung von Bernhard Wymann funktioniert sehr gut, was seine Dokumentation belegt.

Während die Distanzmessung sehr genaue Werte liefert und immer wieder die gleichen Werte bei mehreren Rundenfahrten herauskommen, ist der Geschwindigkeitsregler kein ganz optimaler Regler. Er schwingt ziemlich deutlich in einem Bereich von -0.25 bzw. +1.5 vom

eingestellten Sollwert. Da bei der speziellen Anwendung der Algorithmen zur Lenkwinkelkorrektur die Geschwindigkeit eher unnötig ist, weil die Algorithmen mit Distanzen und Entfernungen arbeiten, ist dies ein vertretbares Ergebnis des Reglers.

9.2.2 Der bisektionale Algorithmus

Der bisektionale Algorithmus ist ein Algorithmus, der immer funktioniert, da er nur zwei Kenngrößen hat. Zum einen die Änderungsrate pro Aufruf der Funktion und zum anderen den Idealbereich um die Mittellinie herum, in dem er keine Änderung vornimmt. Der Algorithmus ist enorm robust und leicht zu implementieren und verbraucht so gut wie keine Ressourcen durch zwei kleine if-Abfragen.

Den einzigen offensichtlichen Nachteil, den der bisektionale Algorithmus besitzt, ist die optimale Änderungsrate pro Aufruf. Entweder man entscheidet sich für eine große Änderungsrate pro Aufruf, nimmt damit allerdings große Schwingungen in Kauf, die in einem Wettbewerb wie dem Carolo-Cup viele Punkte kosten könnten durch das Überfahren der seitlichen Fahrbahnbegrenzungsmarkierung oder es wird eine sehr kleine Änderungsrate eingestellt, womit die Schwingungen zwar deutlich reduziert werden, der Algorithmus allerdings auch viel mehr Zeit benötigt, bis die Korrektur durchgeführt ist.

Wie sich im Test (Kapitel 6.2) herausgestellt hat, arbeitet der Algorithmus auch enorm gegen die wirkenden Fliehkräfte, die der Algorithmus nach der Kurvenfahrt wieder korrigieren muss, da sich der Offset auf einen falschen Wert eingestellt hat. Dieses Verhalten kann man zwar unterbinden, indem man die Korrektur in Kurven abschaltet, allerdings nimmt man in Kauf, dass eine größere Abweichung, die kurz vor der Kurve aufgetreten ist, erst nach der Kurvendurchfahrt korrigiert werden kann und somit eventuell das Fahrzeug die komplette Kurve über der Fahrbahnbegrenzungsmarkierung fährt. Dies ist eine kritische Frage und sollte für jede Situation neu beurteilt werden.

9.2.3 Der messende Algorithmus

Einer der größten Nachteile des messenden Algorithmus ist der, dass dieser Algorithmus, um überhaupt eine Korrektur vornehmen zu können, erst einmal Messwerte benötigt. Diese Messwerte müssen mit einer gut eingestellten Lenkung erzeugt werden, um wirklich verwertbar zu sein. Ist dieser Schritt einmal erledigt, kann der Algorithmus durchaus schnell und effektiv eine Verstellung des Lenkwinkels korrigieren innerhalb weniger Schritte und je nach Einstellung sogar schneller als der bisektionale Algorithmus.

Bemerkenswert an den Einstellungen ist im Test gewesen, dass die besten Einstellungen, wenn man von einer Distanz und nicht der Zeit ausgeht, bei ungefähr der gefahrenen Geschwindigkeit liegen. Als gute Distanz hat sich eine Strecke von 15 Metern herausgestellt bei einer Geschwindigkeit von circa 13 $\frac{m}{s}$ in T.O.R.C.S.. Bei der Implementierung mit der

Polynomapproximation hat sich das etwas anders verhalten, allerdings wird hier auf die Zeit gesetzt. Je höher die Zeit ist, in der eine Änderung durchgeführt werden kann, umso besser wird das Ergebnis. Dabei gibt es im direkten Vergleich keinen unbedingt besseren Algorithmus. Beide Algorithmen schaffen eine fast vollständige Korrektur innerhalb von 100 Metern und beide brauchen nach einer Kurvenfahrt ca. weitere 100 bis 125 Meter zum Ausgleich der Kurvenfahrt. Es ist eher die Frage, worauf der Entwickler Wert legt.

Legt der Entwickler mehr Wert auf eine schlanke Implementierung und eine hohe Geschwindigkeit, so ist die Implementierung mit der Polynomapproximation ein klarer Favorit. Denn im Wesentlichen besteht diese Implementierung aus zwei if-Abfragen und einer Rechnung. Einmal wird zum Start des Algorithmus das Polynom approximiert, wobei dieser Teil eigentlich nicht ins Gewicht fällt, da es ein einmaliger Ablauf ist. Die andere Implementierung mittels Map hat den enormen Nachteil, dass diese Implementierung nach dem Lokalisieren einer Verstellung in der Lenkung zunächst in der Map den passenden Wert suchen und diesen Wert dann einstellen muss. Sollte allerdings die gemessene Abweichung außerhalb des Messbereiches liegen, so muss eine weitere Korrektur durchgeführt werden. Dies würde bei der Polynomapproximation nicht passieren, sofern das Polynom die gemessenen Werte gut approximiert und der Verlauf darüber hinaus ebenfalls gleich ist.

Im Prinzip ist es egal, welches Messkriterium für die Implementierungen verwendet wird, um festzustellen, wann eine neue Änderung durchgeführt werden muss. Bei der Map Implementierung sollte der Wert bei etwas über der gefahrenen Geschwindigkeit liegen. Ob dies nun in Metern oder Sekunden gemessen wird, ist nicht relevant. Beides liefert die gleichen Ergebnisse. So verhält es sich auch bei der Polynom Implementierung. Je länger die Distanz ist, bis eine Korrektur durchgeführt wird bzw. je größer die Zeitspanne für eine Korrektur ist, desto besser wird das Ergebnis. Genau wie beim bisektionalen Algorithmus kommt es vor allem auf die Wahl der richtigen Größe an.

9.3 Die Algorithmen unter FAUST

Bei dem Test der Algorithmen auf der Fahrzeugplattform des Projektes FAUST, die für den Wettbewerb Carolo-Cup entwickelt werden, hat sich herausgestellt, dass der messende Algorithmus zurzeit keine Alternative zum bisektionalen Algorithmus darstellt. Der bisektionale Algorithmus schafft es, einen physikalischen Offset in der Lenkung zu beheben und somit eine Ideallinie zu fahren und dies bereits nach einer recht kurzen Strecke.

Allerdings darf man nicht vergessen, dass im Wesentlichen die Entfernungsmessung mittels Polaris die Schwachstelle ist, die sowohl dem bisektionalen Algorithmus als auch dem messenden Algorithmus Schwierigkeiten bereiten. Während diese Schwierigkeiten beim messenden Algorithmus zu keinem verwertbaren Ergebnis bereits bei der Messung der Abweichung führt, weil zu große Schwankungen zwischen zwei Messungen auftreten, schafft der

bisektionale Algorithmus damit umzugehen. Allerdings kann bei einer längeren Fehlphase von Polaris auch hier der Offset sich erneut verstellen und somit eine neue Korrekturen nötig sein.

Um dieses Problem zu beseitigen, kann der Aufruf des bisektionalen Algorithmus direkt an die Gültigkeit der Polaris Polynome gekoppelt werden. Damit sollte eine leicht verbesserte Korrektur möglich sein.

9.4 Ausblick

Als Zusatz zu den eigentlichen Algorithmen beinhaltete die Aufgabenstellung die Frage, ob es möglich ist, einen Algorithmus zu entwickeln, der angibt, wie viele Umdrehungen die Lenkstangen benötigen, um die Verstellung auch mechanisch zu korrigieren. Nach reiflicher Überlegung bin ich zu dem Schluss gekommen, dass dies nicht möglich ist.

Der Grund ist, dass jede Lenkstange individuell eingestellt werden muss, um ein möglichst gutes Ergebnis zu erzielen. Denn es bedeutet nur einen geringfügigen Nutzen, wenn das Fahrzeug halbwegs gerade ausfährt, allerdings die Vorderräder beide schräg nach außen oder innen zeigen und damit kein gutes Kurvenverhalten mehr besitzen.

Als einziger möglicher Algorithmus, der aber eben dies nicht verhindert, wäre ebenfalls die Auswirkungen auf die Lenkung zu messen, wenn man an den Lenkstangen dreht. Aber dies kann nur parallel passieren, denn es kann nicht berechnet werden, wie jede Stange individuell gedreht werden muss.

Literatur

[Beman Dawes] BEMAN DAWES, David Abrahams (1998-2005); Rene Rivera (2004-2007): *Shared Pointer class template*. – URL `http://www.boost.org/doc/libs/1_47_0/libs/smart_ptr/shared_ptr.htm`

[Carolo-Cup] CAROLO-CUP: *Homepage des Carolo-Cup Wettbewerbs*. – URL `http://www.carolo-cup.de/`

[Carolo-Cup 2011] CAROLO-CUP: *Carolo-Cup Regelwerk 2011*. 2011. – URL `http://www.carolo-cup.de/uploads/media/Regelwerk2011.pdf`

[Coulter 1992] COULTER, R. C.: Implementation of the Pure Pursuit Path Tracking Algorithm / Robotics Institute, Carnegie Mellon University. URL `http://www.ri.cmu.edu/pub_files/pub3/coulter_r_craig_1992_1/coulter_r_craig_1992_1.pdf`, 1992. – Forschungsbericht

[FAUST] FAUST: *Homepage des Projektes FAUST*. – URL `http://www.informatik.haw-hamburg.de/faust.html`

[Jenning 2008] JENNING, Eike: Systemidentifikation eines autonomen Fahrzeugs mit einer robusten, kamerabasierten Fahrspurerkennung in Echtzeit / Hochschule für Angewandte Wissenschaften Hamburg. 2008. – Masterarbeit

[Markelic 2004] MARKELIC, Irene: Reinforcement Learning als Methode zur Entscheidungsfindung beim simulierten Roboterfußball / Universität Koblenz-Landau - Fachbereich Informatik - Arbeitsgruppe Künstliche Intelligenz. URL `http://www.uni-koblenz.de/~fruit/ftp/teaching/mar04-studienarbeit.pdf`, 2004. – Studienarbeit

[Montemerlo u. a. 2007] MONTEMERLO, M. ; DAHLKAMP, H. ; RIEDMILLER, M.: *Learning to Drive a Real Car in 20 Minutes*. 2007. – URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.3532&rep=rep1&type=pdf`

[Nikolov 2010] NIKOLOV, Ivo: NFQ zur Optimierung eines Lenkungsreglers / Hochschule für Angewandte Wissenschaften Hamburg. 2010. – Projekt 1 Ausarbeitung

[T.O.R.C.S.] T.O.R.C.S.: *The Open Racing Car Simulator - Homepage des Projektes*. – URL `http://sourceforge.net/projects/torcs/`

[Wymann 2005a] WYMANN, Bernhard: *T.O.R.C.S. Manual installation and Robot tutorial*. 2005. – URL `http://www.berniw.org/aboutme/publications/torcs.pdf`

[Wymann 2005b] WYMANN, Bernhard: *T.O.R.C.S. Manual installation and Robot tutorial (Online Version)*. 2005. – URL `http://www.berniw.org/torcs/robot/ch3/gears.html`

Abbildungsverzeichnis

Tabellenverzeichnis

*Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 31. August 2011	
Ort, Datum	Unterschrift