UNIVERSITÄT BAYREUTH

Fakultät für Mathematik, Physik und Informatik
Mathematisches Institut

Analyse, Weiterentwicklung und Optimierung eines Simulationsmodells für ein inverses Pendel

Diplomarbeit

von

Christoph Baumann

30. Oktober 2012

Betreuung:
Prof. Dr. L. Grüne
Dipl.-Math. T. Jahn

Danksagungen

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. Grüne für die interessante Themenstellung und die Bereitstellung der Apparatur, ohne die die Bearbeitung des Themas dieser Arbeit in dieser Form nicht möglich gewesen wäre, bedanken. Des Weiteren möchte ich mich für die hervorragende Betreuung meiner Diplomarbeit herzlichst bei Herrn Thomas Jahn bedanken, der mir bei der Anfertigung der vorliegenden Arbeit jederzeit mit Rat und Tat und sehr viel Engagement hilfreich zur Seite stand.

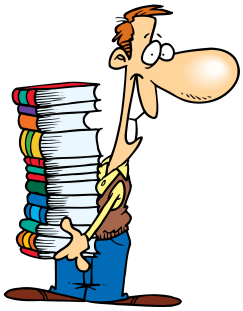
Außerdem möchte ich mich bei all denjenigen bedanken, die mich innerhalb der Entstehungszeit dieser Diplomarbeit begleitet und unterstützt haben. Mein ganz besonderer Dank gilt meiner Freundin und meinen Eltern. Meiner Freundin, weil sie mir immer den nötigen Rückhalt und moralische Unterstützung gegeben hat. Meinen Eltern, weil sie immer versucht haben, mir in meinem Leben alles zu ermöglichen und mir stets den richtigen Weg gezeigt haben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation



„Riskiere ich alles fallen zu lassen
oder gehe ich lieber zweimal?“

Gleichgewicht halten bzw. Gegenstände balancieren – eine Fähigkeit, die Mensch und Tier (man denke an den Seelöwen, der einen Ball balanciert) relativ früh und intuitiv beherrschen. Damit dies gelingt, sind jedoch eine ständige Beobachtung der Situation und Korrekturen durch Ausgleichsbewegungen nötig. Was für Lebewesen recht einfach scheint, ist für Maschinen und Roboter eine große Herausforderung. Auch bei der Stabilisierung eines sogenannten „inversen Pendels“ muss den Computern bzw. Schaltkreisen zunächst die Physik des Systems und die zur Balance nötigen Bewegungen beigebracht werden.

Die genannte Stabilisierungsaufgabe ist eines der bedeutendsten und anschaulichsten klassischen Probleme der Regelungstechnik, da es nicht nur von rein akademischem Interesse ist, sondern reihenweise praktische Anwendungen existieren. Einige davon werden nachfolgend genannt [5]:

- Stabilisierung der vertikalen Position eines Space Shuttles zu Beginn des Fluges
- Balancieren einer Rakete bei der Fahrt zur Startrampe
- Halten eines zweibeinigen Roboters in einer senkrechten Position – auch ein stillstehender Mensch kann so als inverses Pendel betrachtet werden
- Dynamik eines Roboterarms, falls der Kraftangriff unter dem Schwerpunkt des Armes liegt und das System somit instabil ist

- Einachsige, selbststabilisierende Roller, z.B. Segway
- Lastkranregelung[1]

Das inverse Pendel ist von seiner theoretischen Modellierung und seinen Anforderungen an eine Stabilisierung her gesehen durchaus komplex, gleichzeitig aber von seinem einfachen Aufbau und seiner Anschaulichkeit her kaum zu übertreffen. Diese Eigenschaften machen das inverse Pendel weltweit zu beliebten Lehr- und Praktikumsobjekten im Fachbereich der Regelungstechnik. Die Problemstellung beim inversen Pendel ist dementsprechend bereits gut erforscht, aber dennoch nach wie vor spannend, da hier verschiedene Forschungsbereiche wie Robotik, Kontrolltheorie und rechnergestützte Regelung kombiniert werden. Da es sich bei diesem System um ein nichtlineares, instabiles und unteraktuiertes[2] System handelt, eignet es sich hervorragend als Experimentierplattform für das Überprüfen bestehender und dem Entwicklen neuer Regelungsalgorithmen. [11]

Die Realisierung eines geeigneten Reglers kann mittels verschiedener Entwurfsstrategien erfolgen, hier seien PID-Regler, Regler mit Polvorgabe, LQR-Regler, Fuzzy Systeme oder Neuronale Netze genannt. Oft wird auch das Aufschwingen des Pendels aus der Nulllage durch Bewegungen des Wagens realisiert. Dafür sind nichtlineare Herangehensweisen wie Energieansätze erforderlich. [5]

1.2 Zielsetzung

Momentan wird das am Lehrstuhl für Angewandte Mathematik vorliegende inverse Pendel mittels linearer Feedback-Regelung stabilisiert, bei der mittels Zustandsrückführung eine Führungsgrößenaufschaltung erfolgt. Dieses Vorgehen wird in [14] erläutert. Da hierfür eine Linearisierung des nichtlinearen Modells um den Arbeitspunkt nötig ist, funktioniert diese Regelung nur lokal. Das bedeutet, dass sich das Pendel bereits mit geringer Winkelgeschwindigkeit in einem kleinen Winkel um die aufrechte Position herum befinden muss, bevor die Regelung einschreiten kann.

Da diese Situation nicht unbedingt befriedigend ist, sollen auch Regelungstechniken verwendet werden, die mit der Nichtlinearität des Systems zurecht kommen. Dazu zählt die modellprädiktive Regelung (meist Model Predictive Control, MPC). Dabei wird das Simulationsmodell des zu regelnden Prozesses verwendet, um die zukünftigen Zustände in Abhängigkeit der gewählten Steuerung zu berechnen. Die aufgrund dieser Vorhersage berechnete optimale Steuerung wird jedoch nur für einen Zeitschritt verwendet und anschließend die Optimierung wiederholt. [16]

Da diese Art Regelung auf der Vorhersage der zukünftigen Zustände beruht, ist es unablässlich, ein Simulationsmodell zu verwenden, das die Realität möglichst exakt beschreibt, damit dieses Konzept erfolgreich anwendbar ist. Das bisher implementierte Simulationsmodell des vorliegenden inversen Pendels ist aufgrund vieler Vereinfachungen und idealisierter

[1] Dabei soll eine von einem Kran transportierte schwingende Last durch lineare Bewegungen des Kranläufers möglichst schnell zum Stillstand gebracht werden. In gewisser Hinsicht könnte man das inverse Pendel als eine „auf den Kopf gestellte“ Lastkranregelung bezeichnen.

[2] d.h. die Anzahl der Stelleingriffe ist kleiner als die Anzahl der mechanischen Freiheitsgrade

Annahmen zu ungenau, um solch moderne Regelungsalgorithmen darauf erfolgreich anwenden zu können. Aus diesem Grund soll in dieser Diplomarbeit das Simulationsmodell für das inverse Pendel präzisiert werden – insbesondere die Positionsangabe des Schlittens –, sodass die Realität besser beschrieben wird. Darüber hinaus soll die Arbeit auch als Handbuch für zukünftige Studenten dienen, die sich mit der vorliegenden Pendelapparatur von Googol Technology beschäftigen werden.

1.3 Gliederung

Im nächsten Kapitel werden die Grundlagen über das inverse Pendel erläutert. So wird zum einen das bisherige Simulationsmodell eingeführt und zum anderen die Ist-Situation beschrieben. In Kapitel 3 wird dann auf die Pendelapparatur und deren Steuerung eingegangen. Dabei werden die programmiertechnischen Vorarbeiten erklärt, die nötig waren, um die Fahrversuche und Messungen durchführen zu können. Das darauffolgende Kapitel widmet sich den Fahrversuchen und deren Aufzeichnungen, bevor im fünften Kapitel die Messungen analysiert und ausgewertet werden. In Kapitel 6 werden schließlich neue Simulationsmodelle eingeführt und mit dem bisherigen Modell verglichen. Im vorletzten Kapitel werden kurz weitere Ansätze und Ideen zur Modellierung vorgestellt, bevor im achten Kapitel eine Zusammenfassung gegeben und das Fazit gezogen wird.

Kapitel 2

Grundlagen

In diesem Kapitel wird nun genauer auf das inverse Pendel eingegangen – es erfolgt die mathematische Beschreibung der physikalischen Gegebenheiten. Dabei wird das vereinfachte bisherige Simulationsmodell eingeführt und gezeigt, dass dieses die Realität nicht exakt beschreiben kann.

2.1 Das inverse Pendel

Wir beschäftigen uns hier ausschließlich mit der am häufigsten vertretenen Bauform, dem „einachsigen inversen Pendel“, eine einfache Skizze hiervon ist in Abbildung 2.1 dargestellt. Der Aufbau besteht aus einem horizontal frei beweglichen Wagen und einem Pendelgewicht, das über einen Ausleger mit dem Wagen drehbar gelagert verbunden ist. Das inverse Pendel soll nun durch die richtige Ansteuerung des Wagens vertikal nach oben balanciert werden.

Als Beispiele für weitere Versionen von inversen Pendeln seien hier das „räumliche inverse Pendel“, das „rotatorische inverse Pendel“, das „mehrachsige inverse Pendel“ und das „doppelte inverse Pendel“ genannt. Diese alternativen Pendelmodelle können beispielsweise in [5] nachgelesen werden.

2.2 Einführung des bisherigen Simulationsmodells

Für eine detaillierte Herleitung des Pendelmodells ohne Schlitten mittels physikalischer Gesetze sei hier auf [2] und [15] verwiesen. Der Wagen des inversen Pendels lässt sich durch Vorgabe der Beschleunigung steuern, welche durch zweimaliges Integrieren die Position liefert. Durch Vereinfachung der vorhandenen Gleichungen und Überführung in ein MATLAB-konformes Differentialgleichungssystem erster Ordnung erhält man schließlich:

$$\left.\begin{aligned} \dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= -\frac{c}{J}\,x_2(t) + \frac{m\,\bar{l}}{J}\,\left(g\sin x_1(t) + u(t)\cos x_1(t)\right) \\ \dot{x}_3(t) &= x_4(t) \\ \dot{x}_4(t) &= u(t) \end{aligned}\right\} =: f(x(t), u(t)) \tag{2.1}$$

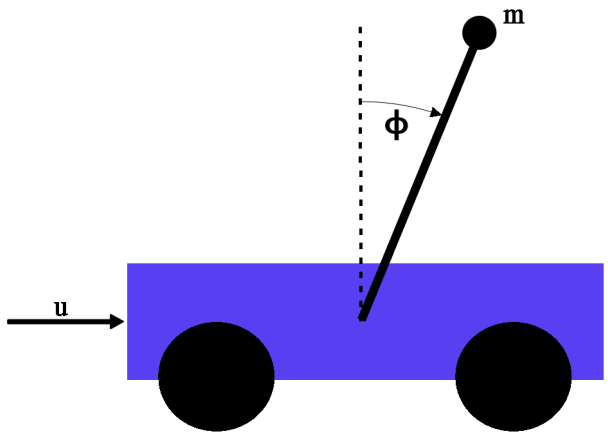


Abbildung 2.1: Schematische Darstellung des inversen Pendels auf einem Wagen

Die Komponenten des Zustandsvektors $x(t) \in \mathbb{R}^4$ und $u(t) \in \mathbb{R}$ beschreiben dabei folgende Zustände:

$x_1(t)$:	Winkel $\Phi(t)$ zwischen Pendel und aufrechter Position, nimmt mit dem Uhrzeigersinn zu
$x_2(t)$:	Winkelgeschwindigkeit des Pendels
$x_3(t)$:	Position des Schlittens
$x_4(t)$:	Geschwindigkeit des Schlittens
$u(t)$:	Beschleunigung des Schlittens (= Eingang des Systems und Kontrollgröße)

In diesem Modell wird das Pendel als starrer Körper mit Masse m und Trägheitsmoment J angenommen. Die Konstante c beschreibt den Reibungskoeffizient. Es gilt dabei: je größer c, desto größer die Reibung. $\bar{l}$ beschreibt den Abstand des Schwerpunkts des Pendels zur Rotationsachse. Diese beiden in der zweiten Gleichung auftretenden Brüche wurden im bisherigen Modell mit den Werten $K_1 := \frac{c}{J} = 0.029787$ und $K_2 := \frac{m\bar{l}}{J} = 1.466434$ implementiert.

Da wir nur zwei Sensoren, zur Messung der Schlittenposition und des Pendelwinkels, zur Verfügung haben, besteht unser Systemausgang $y(t)$ lediglich aus den zwei Komponenten $x_1(t)$ und $x_3(t)$:

$$y(t) = C\,x(t) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} x(t) = \begin{pmatrix} x_1(t) \\ x_3(t) \end{pmatrix} \tag{2.2}$$

Dieses Modell stellt jedoch eine starke Vereinfachung der Realität dar. So wurde zum einen angenommen, dass das Pendel so leicht ist, dass der Einfluss seines Gewichts auf die Bewegung des Wagens vernachlässigbar ist und zum anderen wurden mehrere Konstanten so gewählt, dass sie sich gegenseitig aufheben. Des Weiteren wird die Position des Wagens lediglich durch die zweimalige Integration der Beschleunigung bestimmt und dadurch werden auch hier idealisierte Verhältnisse angenommen, da Aspekte wie Verzögerungen, Reibungen oder sonstige Störungen ignoriert werden.

2.3 Beschreibung der Ist-Situation

Aufgrund der oben angedeuteten Vereinfachungen beim bisherigen Modell kann dieses die Realität nicht exakt beschreiben. Zur Illustration wird hier ein Vergleich zwischen den Simulationswerten des aktuellen vereinfachten Modells und den realen Messwerten der Wagenposition gezeigt. Man sieht deutlich, dass bereits beim denkbar einfachsten Fahrversuch[1] mit lediglich einer konstanten Beschleunigung nach einer halben Sekunde eine Abweichung von über 2 cm vorhanden ist. Bei einem Fahrweg von nur 35 cm entspricht das einem Fehler von knapp 6%.

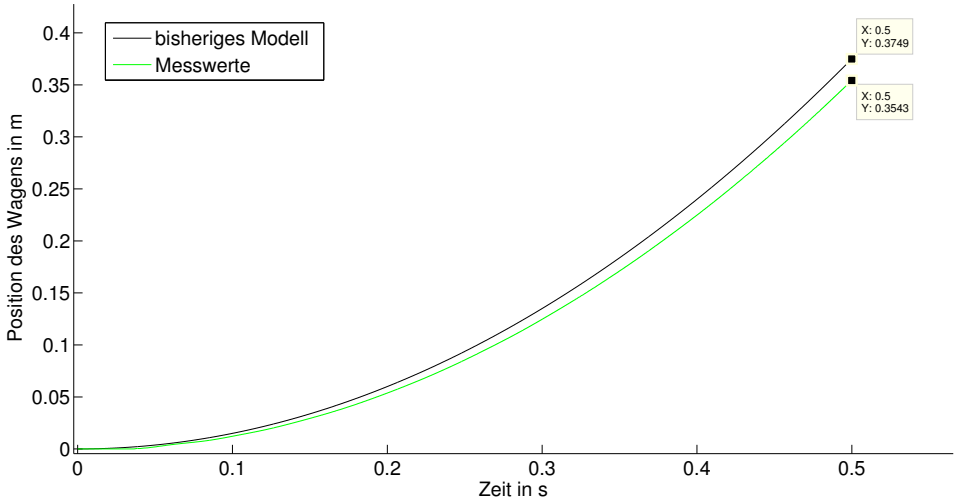


Abbildung 2.2: Vergleich der Positionsmesswerte aus dem Fahrversuch „Step1“ mit den Simulationswerten aus dem bisherigen Simulationsmodell

Aufgrund dieser inkorrekten Wagenposition ist natürlich auch der Pendelwinkel entsprechend verfälscht und nicht realitätsnah. Die folgende Abbildung 2.3 zeigt den Vergleich der gemessenen Winkelwerte mit den Werten aus dem bisherigen Simulationsmodell. Die maximale Differenz beträgt 0.01473 rad, dies entspricht einem Winkel von 0.84°. Dies mag zunächst wenig erscheinen, beachtet man aber die Pendellänge von 80 cm, so beträgt die Differenz der Bogenlänge bereits 1.18 cm. Dieser Fehler kann also durchaus entscheidend dafür verantwortlich sein, ob das Pendel aufrecht stehen bleibt oder nicht.

An dieser Stelle ist anzumerken, dass in der vorliegenden Arbeit die Daten aus Messwerten stets grün, das alte Modell stets schwarz und neue Modelle rot oder blau geplottet werden. Diese einheitliche Gestaltung vereinfacht das schnelle Verständnis der Abbildungen.

[1] Diesen Fahrversuch werden wir später mit „Step1“ bezeichnen.

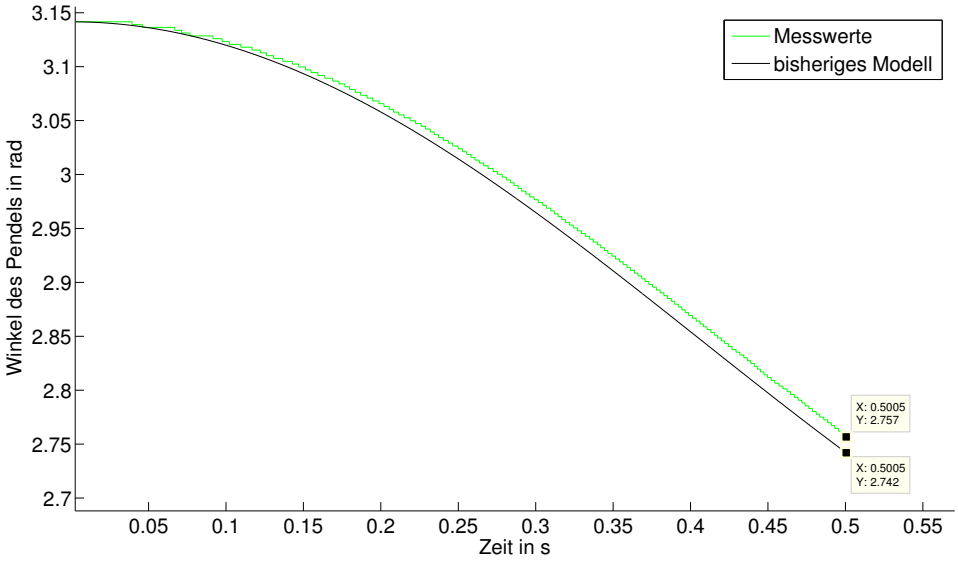


Abbildung 2.3: Vergleich der Winkelmesswerte aus dem Fahrversuch „Step1“ mit den Simulationswerten aus dem bisherigen Simulationsmodell

Kapitel 3

Technische Details

In diesem Kapitel werden die technischen Details des vorliegenden inversen Pendels erläutert. Nachdem die Apparatur erklärt wurde, wird insbesondere auf die Steuerung des Motors eingegangen, um die danach erläuterte Programmierung nachvollziehen zu können. Es wird sich an den Handbüchern des Pendelherstellers „Googol Technology (HK) LTD“ [12] und des Servomotorproduzenten „Panasonic“ [24] orientiert.

3.1 Beschreibung der Apparatur

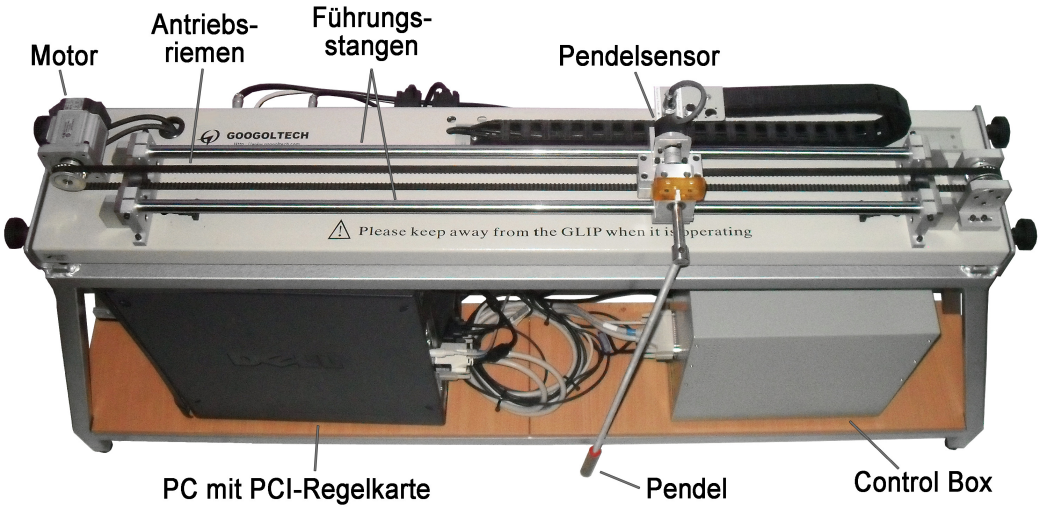


Abbildung 3.1: Foto des inversen Pendels

Bei der am Lehrstuhl vorhandenen Apparatur eines inversen Pendels handelt es sich um das Modell „GLIP2001“ von Googol Technology aus Shenzhen (China) bzw. Hongkong. Es be-

steht aus einem Schlitten, der sich entlang zweier paralleler Schienen bewegen kann und über einen Riemen von einem Motor angetrieben wird, einem DSP-basiertem[1] Kontrollsystem und dem Pendel selbst, welches auf dem Schlitten drehbar montiert ist.

Der hier verwendete Motor ist ein sogenannter **Servomotor**. Im Gegensatz zu einfachen Gleichstrommotoren ist hier bei konstanter Last die Drehzahl nicht proportional zur Motorspannung und kann deshalb nicht einfach durch Veränderung der Spannung leicht und direkt angepasst werden. Dafür ist dieser für Positionierungsaufgaben verwendbar, da er einen geschlossenen Regelkreis und mit dem Antriebsverstärker zusätzliche Elektronik besitzt. Realisiert wird der geschlossene Regelkreis dabei durch einen Drehgeber als Messeinrichtung (z.B. einem Resolver, einem Inkrementalgeber oder einem Absolutwertgeber) und einem Regelschaltkreis, der den Motorstrom anhand von gewünschtem Soll- und aktuellem Istwert reguliert. Dadurch sind beispielsweise gleichmäßige Fahrprofile bei schwankenden Lasten möglich. [24]

Bei unserem Servoantrieb handelt es sich um einen Motor der MINAS A4 Serie von Panasonic mit integrierter Drehzahl-, Drehmomenten- und Positionsregelung. Die Messung der Istwerte erfolgt über einen Inkrementalgeber. Aufgrund von 32bit Technik und 1000 Hz Abtastung zeichnet er sich durch eine hohe Regelgeschwindigkeit und ein niedriges Eigenträgheitsmoment aus. Darüber hinaus führen Eigenfrequenz-Filter und Störmomentüberwachung zu geringen Vibrationen. Außerdem erfolgt eine optimale Anpassung an Lastverhältnisse durch Autotuning während des Betriebs. [24] Diese Eigenschaften, die einen Projektingenieur bei der Realisierung einer Automatisierungslösung erfreuen dürften, da sie für eine gute Dynamik und einen ruhigen Lauf sorgen, erschweren jedoch die Modellierung eines neuen Systems, da nun zwischen dem Simulationsmodell des inversen Pendels und dessen zugehöriger Regelung noch die komplexe integrierte PID-Regelung des Motors steckt.

Üblicherweise werden Positioniersysteme wie diese von Panasonic in den Bereichen der kunststoffverarbeitenden Industrie, im Sondermaschinenbau, bei Fertigungsprozessen in der Bekleidungsindustrie, in Verpackungsmaschinen und bei Zulieferfirmen für den Automobil- und Elektronikanlagenbau eingesetzt. [23]

Bei der späteren Regelung unseres inversen Pendels übernimmt der Motor sowohl die Rolle des Aktors (Antrieb) als auch eines Sensors (zur Positionsbestimmung des Schlittens durch den Inkrementalgeber des Servomotors). Zusätzlich befindet sich noch ein Sensor an der Aufhängung des Pendels zur Bestimmung des Pendelwinkels.

Der Antriebsverstärker des Servomotors befindet sich in der grauen „Control Box“. Darüber hinaus beinhaltet diese ein Netzgerät, einen Stromkreisschutz und die Geräteschnittstelle. Zusätzlich befindet sich ein „An-Aus-Schalter“ auf der Vorderseite des Geräts. Vom Hersteller wird empfohlen, die „Control Box“ erst dann an- bzw. bereits auszuschalten, wenn der angeschlossene PC läuft.

Zur Mechanik kommen noch Kontrollhardware und Kontrollsoftware hinzu, um die Regelung des inversen Pendels mit Hilfe der Sensoren und des Aktors durchführen zu können. Die Hardware besteht aus einem PC als Basis zur Programmentwicklung und als Benutzerschnittstelle, einer DSP-basierten **PCI-Regelkarte** namens GT-400-SV-PCI von Googol Technology sowie mehreren Verbindungskabeln, um den PC mit der „Control Box“ und die „Control Box“ mit dem Pendelsockel zu verbinden. Für die korrekte Installation der

[1] DSP = digital signal processing, zu deutsch: digitale Signalverarbeitung

PCI-Karte inklusive Treiber[2] und den richtigen Anschluss aller Verbindungskabel sei auf das Handbuch [12] verwiesen. Diese PCI-Regelkarte bildet also die Schnittstelle zwischen unserem Programm, d.h. unserer Regelung, und dem mechanischen Aufbau unseres inversen Pendels. Theoretisch könnte dieser Motion Controller bis zu vier Achsen gleichzeitig kontrollieren, es sind hier jedoch nur zwei vorhanden: zum einen der Motor (zum Steuern und Auslesen) und zum anderen das Pendel (nur zum Auslesen). Der schematische Aufbau des Gesamtsystems ist in Abbildung 3.2 zu sehen.

Die Control Software kommt standardmäßig mit zwei Varianten aus dem Werk von Googol Technology: zum einen eine DOS-basierte Version, in der alle Regelalgorithmen in C programmiert sind und zum anderen eine MATLAB-basierte Version als interaktives Tool zum Modellieren, Simulieren und Analysieren des Pendelsystems in Simulink. Wir verwenden hier jedoch lediglich die in der erstgenannten Variante mitgelieferte Windows-Programmbibliothek (DLL) von Googol Technology.

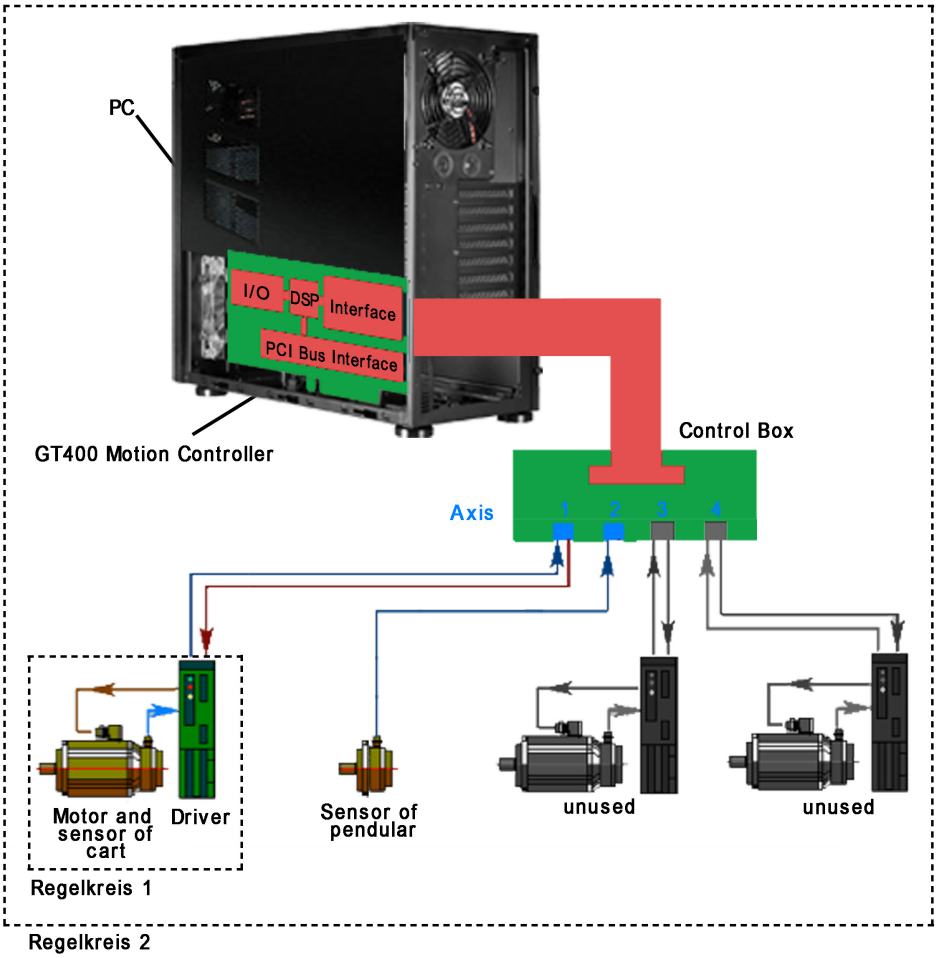


Abbildung 3.2: Schematischer Aufbau - GT Motion Controller und Gesamtsystem

[2]Diese befinden sich auf der von Googol Technology mitgelieferten CD.

3.2 Ansteuerung des Motors

Nach Erläuterung der hardwaretechnischen Gegebenheiten im vorherigen Abschnitt stellt sich nun die Frage, wie man den Motor des Schlittens mit einem Programm ansprechen und steuern kann. Hierzu wurde von Googol Technology eine umfangreiche Bibliothek zur Verfügung gestellt, die in C geschrieben ist und sämtliche Methoden beinhaltet, um mit der PCI-Regelkarte den Motor kontrollieren zu können. So kann man mit den Methoden dieser DLL beispielsweise den Motor konfigurieren, die Sensoren auslesen und den Motor in Bewegung setzen. Da diese Methoden sehr allgemein sind, werden die für unsere Zwecke wichtigen Methoden zur einfachen Handhabe nochmals in eine Klasse gekapselt, die speziell auf das vorhandene Pendel zugeschnitten ist.

3.2.1 Einrichtung der Entwicklungsumgebung und Anbindung der Bibliothek

Es wird als Entwicklungsumgebung Code::Blocks 10.05 verwendet und als Compilerpaket ist MinGW[3] installiert. Nachdem das Projekt angelegt worden ist und man die Headerdatei der Bibliothek `Userlib.h` eingebunden hat, funktioniert das Kompilieren und Anbinden der DLL in der Kommandozeile ganz simpel mit:

```
> gcc -c main.cpp          // Compile or assemble the source file(s), but do not link
> gcc main.o GT400.DLL     // Link the compiled file(s) with library
```

Falls man direkt in Code::Blocks kompilieren und linken möchte, so muss man in der Menüleiste unter „Project“ den Eintrag „Build Options“ auswählen. Anschließend muss im entsprechenden Projekt unter „Linker Settings“ im rechten Feld namens „Other linker options“ lediglich der Name der Bibliothek `GT400.DLL` eingetragen werden.

3.2.2 Ansprechen des Motors

Alle verfügbaren Methoden der Bibliothek können im „GT Programming Manual“ [10] nachgeschlagen werden. Nichtsdestotrotz wird hier zumindest kurz beschrieben, wie man den Wagen steuern und kontrollieren kann. Grundsätzlich beginnt jeder Funktionsaufruf aus der DLL mit dem Präfix „`GT_`“ und sollte mit dem zurückgelieferten Rückgabewert auf erfolgreiche Durchführung überprüft werden. Als erstes muss der Controller durch den Aufruf `GT_Open()` geöffnet und anschließend die verschiedenen Achsen initialisiert werden. Wichtig ist dabei die Tatsache, dass der Controller stets nur eine Achse ansprechen kann. Alle achsspezifischen Befehle beziehen sich dabei immer auf die zuletzt aktivierte Achse. Möchte man also etwa den Motor starten, so muss zunächst `GT_Axis(1)` aufgerufen werden, um die erste Achse zu aktivieren und anschließend durch `GT_AxisOn()` der Motor angeschaltet werden. Möchte man den Pendelsensor auslesen, so muss durch den Aufruf `GT_Axis(2)` die entsprechende Achse aktiviert werden, bevor anschließend der Winkel mittels `GT_GetAtlPos(&pos)` ausgelesen werden kann. Nach dem Beenden des Fahrversuchs wird der Motor durch `GT_AxisOff()` deaktiviert und der Controller durch den Befehl `GT_Close()` wieder geschlossen.

[3] Version vom 18.11.2011

Der Motor lässt sich in vier verschiedenen Modi (auch „Bewegungsprofilen“ oder „Bewegungsmodi“) steuern: „S-Curve“, „T-Curve“, „V-Curve“ und „Electronic Gearing“. Es wird hier allerdings nur der für uns wichtigste Modus beschrieben, der „V-Curve“-Modus. Dieser wird von Googol Technology auch „Independent jogging“-Bewegungsmodus genannt und durch den Befehl `GT_PrflV()` aktiviert. Der Benutzer spezifiziert hierbei Zielgeschwindigkeit und Beschleunigung der aktuellen Achse. Die Bewegungsrichtung wird durch das Vorzeichen von Geschwindigkeit bzw. Beschleunigung bestimmt. In diesem Modus wird der Motor mit der angegebenen Beschleunigung sein Tempo bis zur Zielgeschwindigkeit erhöhen und mit dieser Geschwindigkeit weiterfahren, bis eine neue Geschwindigkeit oder das Stop-Kommando gegeben wird. In diesem Modus muss also keine Zielposition angegeben werden. Darüber hinaus ist das Bewegungsprofil sehr flexibel, da man Geschwindigkeit, Beschleunigung und Richtung während der Bewegung ändern kann. Ein typisches Geschwindigkeitsprofil für diesen Bewegungsmodus wird in Abbildung 3.3 gezeigt, alle vorgegebenen Kommandos sind dabei durch schwarze Pfeile markiert. In Listing 3.1 ist anschließend ein beispielhafter Programmausschnitt zur Anwendung des „Independent Jogging“-Profils zu sehen.

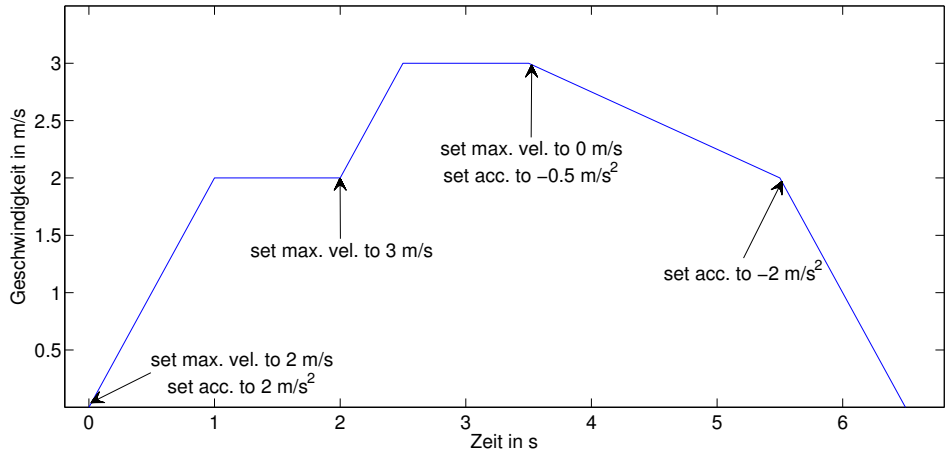


Abbildung 3.3: Geschwindigkeitsprofil im „Independent Jogging“-Bewegungsmodus bei Geschwindigkeits- und Beschleunigungsänderungen

```
void ExampleVMotion()   // Motion function of independent jogging mode
{
    GT_Open();          // Open the motion controller

    // ... initializing ...

    GT_Axis(1);         // Set the first axis as current axis
    GT_AxisOn();        // Activate drive

    GT_PrflV();         // Specify the jogging mode of motion for current axis
    GT_SetAcc(0.01);    // Specify the acceleration (in pulses per squared sampletime)
    GT_SetVel(1);       // Specify the target velocity (in pulses per sampletime)
    GT_Update();        // Update parameter

    // ... other commands ...

    GT_AxisOff();       // Disable drive
```

```
    GT_Close();          // Close the motion controller
}
```

Listing 3.1: Beispielanwendung für den Bewegungsmodus „Independent Jogging“

Da in unserem Modell stets die Beschleunigung des Wagens als Stellgröße zur Stabilisierung des Pendels vorgegeben wird (und nicht etwa die Zielposition) und der „V-Curve“-Bewegungsmodus derjenige ist, mit dem man konstante Beschleunigungen am ehesten umsetzen kann, wird dieser bei den Fahrversuchen in der vorliegenden Arbeit verwendet. Da wir jedoch „nur“ die Beschleunigung und nicht die Endgeschwindigkeit vorgeben wollen, „missbrauchen“ wir diesen Modus, indem wir die Zielgeschwindigkeit so hoch setzen, dass diese nie erreicht wird und ändern anschließend nur noch die Beschleunigung. Dadurch werden die waagrechten Passagen mit konstanter Geschwindigkeit in Abbildung 3.3 vermieden und lediglich die vorgegebene Beschleunigung hat Einfluss auf die Bewegung des Wagens.

Auf eine tiefere Einführung, d.h. insbesondere die Beschreibung der Initialisierung, des „T-Curve“-Bewegungsprofils (für das Finden der Startposition des Wagens) und des Setzens der sogenannten Breakpoints (= virtuelle Sicherheitsgrenzen), wird im Rahmen dieser Arbeit verzichtet. Für die Erläuterung sei jedoch auf das Handbuch [10] verwiesen.

3.2.3 Die Klassen „PendulumBase“ und „AccControl“

Die Reduzierung auf eine sehr handliche Klasse, die auf unsere Zwecke zugeschnitten ist, erfolgt analog einer bereits am Lehrstuhl vorhandenen Implementierung in C#, mit dem das Pendel manuell aufschwingbar und mittels linearer Feedback-Kontrolle lokal stabilisierbar ist. Das Problem bei der Implementierung in C# ist jedoch, dass um die in C geschriebenen Methoden der DLL verwenden zu können, ein sogenanntes „Marshalling“[4] notwendig ist. Das heißt, es wird in die Bibliothek hineingesprungen, die Methode ausgeführt und anschließend wieder zurückgesprungen, was sich zeitlich sehr negativ auswirkt. So schlug sich das beispielsweise in einer maximalen Ausleserate der Sensoren von 3 ms nieder. Bei unserer Implementierung in C/C++ stellt sich dieses Problem nicht, weshalb wir später u.a. von einer wesentlich höheren Ausleserate profitieren werden.

Die Kapselung der für uns wichtigen Methoden erfolgt in zwei Schritten. Zunächst bauen wir die Klasse „PendulumBase“ mit allen für uns wesentlichen Funktionalitäten wie Motor einschalten, Achsen initialisieren und Sensoren auslesen. In Listing 3.2 ist übersichtshalber die Deklaration dieser Klasse inklusive Kommentierungen abgebildet. Die Quellcodes der Klassendeklarationen und der Implementierung der Methoden sind auf der beiliegenden CD enthalten.

```
1  class PendulumBase
2  {
3      public:
4          PendulumBase();                  // Constructor
5          virtual ~PendulumBase();         // Destructor
6
7          void ErrCheck(int result);       // Error check of return value of called function
8          virtual void InitializeAxis();   // Set specified parameters of motor and current
                                               position of axes to zero
```

[4] mit dem Befehl `Marshal.AllocHGlobal`

```
9           void Reset();                    // Reset the motion controller and do an
                                                initializing
10          void EnableMotor(bool enabled); // Enable and disable motor
11          void Update();                   // Update current axis with chosen parameters
12
13          void FindStartingPosition(double percent);  // Drive to desired starting position
14          void SetAccMPSQ(double mpsq);    // Set acceleration (in meter/second^2)
15          void SetVelMPS(double mps);      // Specify the maximum velocity (in meter/second)
16          void SetPosM(double m);          // Define target position (in meter)
17          void SmthStp();                  // Stop the cart smoothly
18
19          double GetPosition();            // Get current position of cart (in meter)
20          double GetAngle();               // Get current angle of pendular (in radiants)
21
22          void WaitUntilMotionDone();      // Wait until motion of cart is completed
23          void WaitUntilMotionStops();     // Wait until cart is not in motion anymore
24          bool IsMotionDone();             // Is motion of current axis completed?
25          bool IsInMotion();               // Is current axis in motion?
26
27      protected:
28          void Axis(unsigned short current_axis);  // Activate specified axis
29          unsigned short GetSts();         // Get status of current axis
30
31          void PrflT();                    // Specify motion mode of current axis as T-curve
32          void PrflS();                    // Specify motion mode of current axis as S-curve
33          void PrflV();                    // Set the motion mode of current axis as velocity
                                                control mode ("independent jogging")
34          void PrflG(unsigned short master);  // Specify motion mode of current axis as
                                                    electronic gearing and which axis is the
                                                    master axis
35          void SetAtlPos(int value);       // Modify the current position of current axis to
                                                a specified value (in pulses)
36          int GetAtlPos();                 // Get the current position of current axis (in
                                                pulses)
37
38          int smpltime;                    // Sample time of motion controller
39          double posfactor;                // Multiply m (meter) with this factor to get pulses
40          double velocfactor;              // Multiply m/s with this factor to get pulses
                                                per sampletime
41          double accelfactor;              // Multiply m/s^2 with this factor to get pulses
                                                per squared sampletime
42          double radfactor;                // Multiply pulses with this factor to get radiants
43          pthread_mutex_t my_mutex;        // Mutex for synchronizing actions that involve
                                                different axes
44  };
```

Listing 3.2: Die Deklaration der Klasse „PendulumBase“

Im zweiten Schritt leiten wir daraus noch eine weitere Klasse „AccControl“ ab, deren einzige Aufgabe es ist, bestimmte Beschleunigungen bzw. Beschleunigungssequenzen an den Motor weiterzugeben. Die aus diesem Grund sehr übersichtliche Deklaration der Klasse ist in Listing 3.3 zu sehen.

```
1   class AccControl : public PendulumBase
2   {
3       public:
4           AccControl();                        // Constructor
5           virtual ~AccControl();               // Destructor
6
7           virtual void InitializeAxis();       // Activate motor axis and jogging mode
8           virtual void FindStartingPosition(double percent);  // Drive to desired starting
                                                                   position
9           void SetAcceleration(double acc);    // Set specified acceleration
10          void StopMotion();                   // Stop current motion of cart
```

```
11
12      protected:
13          double max_vel;                     // Maximum velocity
14          double max_acc;                     // Maximum acceleration
15  };
```

Listing 3.3: Die Deklaration der Klasse „AccControl“

3.3 Programm zur Durchführung der Versuche

Das Programm zur Durchführung der Fahrversuche soll ermöglichen, dass man mittels verschiedener Beschleunigungssequenzen (d.h. es werden für unterschiedlich lange Zeiten verschiedene Beschleunigungen verwendet) den Motor steuern und dabei nebenläufig die Sensoren des Pendels und des Schlittens auslesen kann. Diese Messwerte sollen dabei in zwei Dateien abgespeichert werden. Während man zum Auslesen des Pendelwinkels Achse 2 des Motion Controllers aktivieren muss, ist für das Ansprechen des Motors und Auslesen der Schlittenposition eine Aktivierung von Achse 1 nötig. Da Beschleunigungsgebung und Sensorauslesen möglichst parallel ablaufen sollen, ist hier der Einsatz von Threads[5] und sogenannten Mutexen nötig.

3.3.1 *Exkurs:* Threads und Mutexe

Programmieren mit sogenannten Threads ermöglicht die Implementierung nebenläufiger Handlungen auch innerhalb eines Prozesses. Threads bilden die kleinste Einheit, denen vom Systemkern CPU-Zeit zugewiesen wird. Man kann sich unter einem Thread eine in Ausführung befindliche, sequentielle Folge von Befehlen vorstellen, die den Prozess wie ein roter Faden durchzieht. Zusammengewoben und verknüpft ergeben die einzelnen Threads dann den gesamten Prozess. Mit Hilfe dieser Threads ist der sehr effiziente Entwurf parallelisierter Programme möglich. Des Weiteren schöpfen die Multithreading-Programme die Systemressourcen im Allgemeinen besser aus, was insbesondere bei Multicore-Systemen der Fall ist. [35] Wir beschäftigen uns hier mit den im POSIX-Standard definierten POSIX-Threads, kurz Pthreads.

In dieser Arbeit soll lediglich ganz kurz auf die für uns nötigen und von uns verwendeten Funktionen eingegangen werden, für eine breitere und vor allem tiefere Einführung in das Thema sei auf die Literatur verwiesen, z.B. in [27] und [30]. Dieser Abschnitt orientiert sich hauptsächlich an [35] und [4].

Grundlagen

Innerhalb eines Programms wird ein Pthread mit Hilfe des Datentyps `pthread_t` durch die Thread-ID referenziert. Mit der Funktion `pthread_create()` wird ein neuer Thread gestartet, wobei die ID bei einem erfolgreichen Start im ersten Parameter zurückgeliefert wird. Der neue Thread ist nichts anderes als eine Funktion mit festgelegter Signatur, die von oben genannter Funktion gestartet wird und dann als eine Art `main()`-Funktion des

[5] zu deutsch: Fäden

erstellten Threads abläuft. Die Startfunktion des neuen Threads erwartet dabei einen Zeiger vom Typ `void *` als Parameter und liefert am Ende einen Zeiger vom gleichen Typ als Rückgabewert zurück. Sobald ein Thread aus seiner Startfunktion zurückkehrt, beendet sich der Thread. Möchte man auf das Ende eines bestimmten Threads warten, so benutzt man die Funktion `pthread_join()`. Ist dieser bereits beendet, so kehrt `pthread_join()` unmittelbar zurück, andernfalls blockiert der aufrufende Thread solange, bis der Zielthread beendet ist. Der Rückgabewert des beendeten Threads wird im Parameter `value_ptr` zurückgeliefert.

```
#include <pthread.h>

pthread_t thread_id;
int pthread_create(pthread_t *thread_id,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
int pthread_join(pthread_t thread_id, void **value_ptr);
```

Listing 3.4: Funktionsaufrufe zur Verwendung von Pthreads

Der `main()`-Funktion eines Pthreads-Programms kommt bei POSIX-Threads eine eigene Rolle zu. Beim Start des Programms besitzt der neue Prozess genau einen ausgezeichneten Thread, den Hauptthread, welcher zunächst den C-Startup-Code durchläuft und danach die Startfunktion des Programms, d.h. die `main()`-Funktion, aufruft. Kehrt die `main()`-Funktion zurück, so wird vom Hauptthread die `exit()`-Funktion aufgerufen, was zur Folge hat, dass der Prozess umgehend beendet wird, unabhängig davon, ob in diesem Moment noch weitere Threads aktiv sind oder nicht. Es ist also wichtig, zuvor alle gestarteten Threads mit `pthread_join()` aufzufangen.

Damit Threads miteinander Daten austauschen können, müssen sie auf denselben Speicherbereich zugreifen. Hierdurch können leicht unerwünschte Nebenwirkungen entstehen. So können mehrere Threads durch gleichzeitige Bearbeitung einer statischen oder globalen Variable diese in einen undefinierten Zustand bringen. Gerade bei der Umsetzung größerer Projekte kann dies zu erheblichen Problemen führen. Man stelle sich beispielsweise eine globale Queue von abzuarbeitenden Aufgaben vor, von der sich ein Thread eine Aufgabe holen und diese anschließend daraus entfernen möchte. Falls nun währenddessen ein anderer Thread die Queue umsortiert, so kann dies zu einer korrupten Liste oder einem Speicherzugriffsfehler führen. [4]

Abläufe, deren Endergebnis von der zeitlichen Abfolge der ausgeführten Operationen abhängig ist, nennt man zeitkritische Abläufe oder Race Conditions. Die Codebereiche, in denen die gemeinsam genutzten Ressourcen getrennt bearbeitet werden, werden kritische Bereiche genannt. Das heißt also: Befindet sich ein Thread in seinem kritischen Bereich, darf kein anderer Thread zur gleichen Zeit Code aus seinem kritischen Bereich ausführen. POSIX-Threads stellen hierfür zwei spezielle Synchronisationsmechanismen zur Verfügung: Mutexe und Bedingungsvariablen, wobei wir hier nur auf erstere eingehen und verwenden werden.

Synchronisation mittels Mutexe

Die Lösung des obigen Problems besteht darin, dass ein Thread, der eine globale oder statische Variable bearbeiten und somit in einen kritischen Bereich eintreten möchte, den Zugriff aller anderen Threads darauf blockiert und erst dann wieder freigibt, wenn er fertig ist.

Dies geschieht mit Hilfe sogenannter „Mutexe“. Ein Thread Nr. 1 kann zunächst einen Mutex setzen. Jeder andere Thread, der versucht, diesen Mutex ebenfalls zu setzen, wird solange blockiert, bis Thread Nr. 1 ihn wieder freigegeben hat. Bildlich kann man sich einen Mutex als einen Ball vorstellen, den sich die Threads gegenseitig zuspielen und nur derjenige Thread darf den kritischen Bereich im Code ausführen, der den Ball aktuell besitzt. Der Begriff Mutex ist ein Kunstwort, das als Abkürzung für Mutual Exclusion, also gegenseitigen Ausschluss steht.

Ein Mutex wird innerhalb eines Programms durch eine Variable vom Typ `pthread_mutex_t` repräsentiert. Damit ein Mutex Sinn macht, müssen natürlich alle Threads, die mit diesem Mutex zu tun haben, auf ihn zugreifen können. In der Regel werden Mutexe deshalb globale oder statische Variablen sein. Bevor man einen Mutex benutzen kann, muss man ihn mittels der Funktion `pthread_mutex_init()` initialisieren. Im zweiten Argument können dabei Attribute, die die Eigenschaften des Mutex bestimmen, übergeben werden. Um einen Mutex zu setzen, wird die Funktion `pthread_mutex_lock()` aufgerufen und es wird überprüft, ob der Mutex bereits von einem anderen Thread gesetzt wurde. Falls ja, wird so lange gewartet, bis dieser den Mutex wieder freigibt. Ist der Mutex dagegen frei, so wird er gesetzt und fortgefahren. Mit `pthread_mutex_unlock()` wird ein Mutex schließlich wieder freigegeben. Normalerweise sollte der Thread, der einen Mutex gesetzt hat, diesen auch wieder freigeben. Wird der Mutex nicht mehr gebraucht, wird er durch den Befehl `pthread_mutex_destroy()` wieder beseitigt.

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *mutexattr);

int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Listing 3.5: Funktionsaufrufe zur Verwendung von Mutexen

Thread-Sicherheit

Mutexe werden in erster Linie dazu verwendet, den wechselseitigen Ausschluss beim Zugriff auf globale Datenstrukturen sicherzustellen. Ist dies für eine gesamte Funktion sichergestellt, wird sie als thread-sicher (oder threadsafe) bezeichnet. Die Funktion kann also von mehreren Threads nebenläufig aufgerufen werden kann, ohne dabei Schaden anzurichten. Die nebenläufigen Aufrufe der Funktion dürfen sich hierbei gegenseitig nicht negativ beeinflussen, also insbesondere keine Race Conditions oder einen Deadlock verursachen. Ein Deadlock ist ein Zustand, in dem jeder Thread auf ein Ereignis wartet, das nur von einem anderen Thread ausgelöst werden kann, der aber auch vergeblich auf ein Ereignis wartet. [27]

3.3.2 Schematischer Programmablauf

Das Programm zum Durchführen der Fahrversuche und zum Aufzeichnen der Messdaten lässt sich nun also wie folgt beschreiben: In der `main()`-Funktion, d.h. dem Hauptthread, wird zunächst ein weiterer Thread gestartet, der die Funktion `SensorWriter()` aufruft, die dafür zuständig ist, die beiden Sensoren von Pendel und Schlitten auszulesen. Anschließend wird eine Unterfunktion namens `ApplyAccSequence()` aufgerufen, die die Beschleunigungssequenz an den Motor übergibt. Abbildung 3.4 zeigt den beschriebenen Programmablauf schematisch. Für die exakte Implementierung sei auf die beiliegende CD verwiesen.

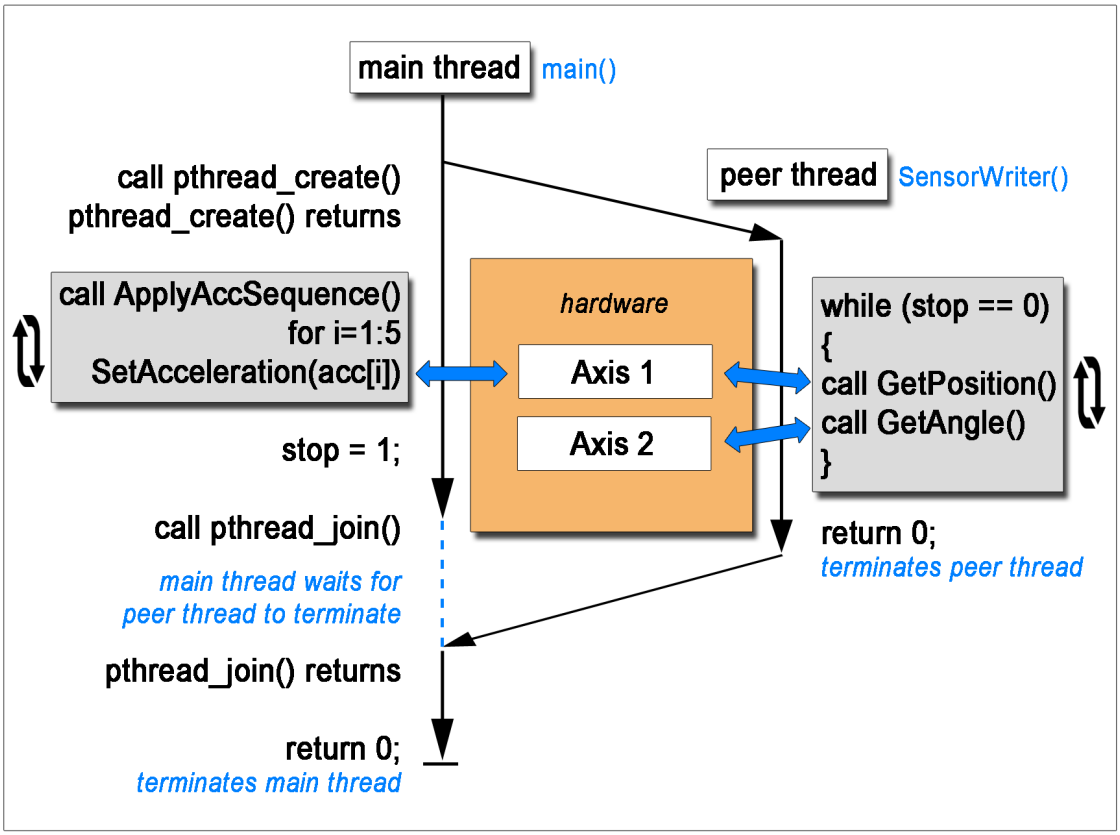


Abbildung 3.4: Schematischer Programmablauf

Da der Nebenthread zum Auslesen der Sensoren nicht nur Achse 1 (für den Schlittensensor), sondern auch Achse 2 (für den Pendelsensor) aktiviert, sind hier Mutexe nötig, um den Ablauf thread-safe zu machen. Der kritische Bereich im Programmablauf ist hierbei jeweils die Zeit zwischen Aktivierung einer Achse und der Durchführung der gewünschten Tätigkeit auf dieser Achse. Es ist also notwendig, diese Bereiche in allen Programmbestandteilen durch einen Mutex abzusichern. Andernfalls könnte es beispielsweise vorkommen, dass der Hauptthread die Beschleunigung auf Achse 2 statt auf Achse 1 gibt, falls der Nebenthread zwischenzeitlich erstere zum Auslesen des Pendelsensors aktiviert hat. Darüber hinaus ist denkbar, dass der

Nebenthread Achse 2 aktiviert, da er im Anschluss den Pendelsensor auslesen möchte. Bevor er jedoch dazu kommt, könnte vom Hauptthread bereits wieder Achse 1 für die nächste Beschleunigungsgebung aktiviert werden. Ruft nun der Nebenthread `GT_GetAtlPos(&pos)` zum Sensorauslesen auf, so erhält dieser die Position des Motors und nicht den Winkel des Pendels. Das Programm würde in diesem Fall sogar fehlerfrei fortfahren, sodass es unbemerkt zu falschen Messwerten kommen könnte.

3.4 Anbindung des impliziten Lösers RADAU5 an MATLAB

Da später Differentialgleichungen mit Massenmatrizen gelöst werden müssen, sollen an dieser Stelle gleich die nötigen Vorbereitungen getroffen werden. Die in MATLAB integrierten Löser kommen leider nicht mit Differentialgleichungen zurecht, die komplexere Massenmatrizen[6] besitzen. MATLAB bietet jedoch die Möglichkeit, separat kompilierten Code auszuführen – die sogenannten MEX-Files[7]. MEX-Files sind Interface-Routinen zwischen MATLAB-Scripts und C- bzw. Fortran-Subroutinen und können im Gegensatz zu „normalen“ C-Funktionen wie m-Funktionen in MATLAB aufgerufen werden. So hat man zum einen den Vorteil, dass bereits vorhandene C- bzw. Fortran-Programme nicht als m-Files komplett neu geschrieben werden müssen und zum anderen ergeben sich bei aufwändigen Berechnungen beträchtliche Geschwindigkeitsvorteile.

In unserem Fall wollen wir den impliziten Löser RADAU5 aus einem MEX-Interface der Technischen Universität München benutzen, der in C und Fortran vorliegt und auch differential-algebraische Gleichungen bis Differentationsindex drei lösen kann, um später für das Lösen dieser Gleichungen universell aufgestellt zu sein. [20]

Nachdem man einen C und einen Fortran Compiler installiert hat, die beide von MATLAB unterstützt werden, kann man diese an MATLAB „ankoppeln“, indem man

```
mex -setup
```

auf MATLABs Kommandozeile eingibt und anschließend die Option mit den entsprechenden Compilern auswählt. Dieser Schritt muss nur ein einziges Mal getätigt werden. Um zu überprüfen, ob die Anbindung erfolgreich war, gibt man

```
mex -v
```

ein. Falls nun eine Liste mit einem C-Compiler (CC) und einem Fortran-Compiler (FC) als Ausgabe erscheint, war der Schritt erfolgreich.

Die Installation des MEX-Interfaces funktioniert anschließend wie folgt:

1. Entpacken des ZIP-Archivs mit dem MEX-Interface in ein eigenes Verzeichnis (Download des ZIP-Archivs unter [20]).

[6] Eine exakte Erläuterung und Einteilung der Löser erfolgt in Abschnitt 6.1.2.
[7] Abkürzung für MATLAB Extension

2. Starten von MATLAB und wechseln in das Verzeichnis aus 1.
3. Eingeben von `compile` auf MATLABs Kommandozeile. Dies führt den Befehl `mex radau5Mex.c options.c radau5.f dc_lapack.f lapack.f lapackc.f` aus.
4. Bei erfolgreicher Installation sollte nun mindestens eine .mexsol- (bei Solaris-Rechnern), eine .mexmaci64- (bei Mac-Rechnern) bzw. eine .dll- (bei Windows-Rechnern, MATLAB<7.1) oder eine .mexw32- (bei Windows-Rechnern, MATLAB≥7.1) Datei entstanden sein.
5. Es gibt nun zwei Möglichkeiten: entweder man setzt MATLABs Suchpfad auf das Verzeichnis unter 1. und kann anschließend das Interface überall verwenden oder man kopiert die Datei aus 4. und die m-Files in das Verzeichnis, in dem man das Interface verwenden will.

Anschließend sollte RADAU5 installiert und verwendbar sein.

Hinweise für Mac-Nutzer:

1. Verwendet man Xcode (von Apples Homepage für Entwicklertools `https://developer.apple.com/xcode`) als C-Compiler, ist zunächst die Installation eines separaten Patches für MATLAB nötig, der hier erhältlich ist: `http://www.mathworks.de/support/solutions/en/data/1-FR6LXJ/`
2. Nach der Installation des Fortran-Compilers (z.B. von `http://gcc.gnu.org/wiki/GFortranBinaries`) muss man beachten, dass gfortran in ‚/usr/local/bin' installiert wird und Bibliotheken in ‚/usr/local/lib' abgelegt werden, sodass man letzteres zum Bash hinzufügen muss, falls man es außerhalb MATLABs aufruft.
3. Anschließend führt man

```
mex -setup
```

   in der MATLAB Kommandozeile aus und wählt die Nummer „1“ für die Auswahl:

```
1: /Applications/MATLAB_R2009b.app/bin/gccopts.sh :
   Template Options file for building gcc MEX-files
```

4. Zuletzt müssen noch folgende Änderungen im ‚maci64'-Abschnitt der Datei ‚~/.matlab/R2011a/mexopts.sh' durchgeführt werden:

```
FC='/usr/local/bin/gfortran'
FC_LIBDIR='/usr/local/lib/'
FC_LIBDIR2='/usr/local/lib/gcc/i686-apple-darwin8
    /4.2.3/x86_64/'
FOPTIMFLAGS='-O5 -funroll-loops -ftree-vectorize'
```

Weitere Erläuterungen sind unter `http://www.mathworks.co.uk/matlabcentral/newsreader/view_thread/262115` zu finden. Danach kann schließlich das MEX-Interface wie oben beschrieben installiert und der implizite Löser RADAU5 verwendet werden.

Kapitel 4

Versuche und Messungen

Um das aktuelle Simulationsmodell mit der Realität vergleichen und anschließend ein besseres Modell entwickeln zu können, werden verschiedene Fahrversuche durchgeführt und die zugehörigen Messdaten aufgezeichnet. In dieser Arbeit werden dabei vier verschiedene Beschleunigungsprofile zum Einsatz kommen. Beginnend bei sehr einfachen Beschleunigungssequenzen - um das Verhalten grundsätzlich kennenlernen zu können - geht es bis zu Sequenzen, die dem Anfang des Aufschwingprozesses für das Pendel entsprechen. Damit die Messwerte auch jederzeit reproduziert werden können, werden hierfür die Versuchsdurchführungen exakt beschrieben.

4.1 Verwendete Beschleunigungsprofile

4.1.1 Step

Den Anfang macht das einfachst mögliche Fahrprofil: es wird eine konstante Beschleunigung für ein paar Hundert Millisekunden vorgegeben. Im regelungstechnischen Sinne entspricht dies einer Sprungfunktion, weshalb das Beschleunigungsprofil in dieser Arbeit auch „Step“ genannt wird. Es werden hier drei unterschiedliche Beschleunigungen verwendet und man erhält somit drei verschiedene Fahrprofile „Step1“-„Step3“, siehe Tabelle 4.1.

Name	Beschleunigung	Dauer
Step1	3.0 m/s^2	500 ms
Step2	1.5 m/s^2	500 ms
Step3	8.0 m/s^2	300 ms

Tabelle 4.1: Beschleunigungsprofile „Step“

Um ein wenig Gefühl für diese Beschleunigungen zu bekommen noch eine kurze Anmerkung: Bei einer konstanten Beschleunigung von 8.0 m/s^2 erreicht man innerhalb von nur 3.47 s eine Geschwindigkeit von 100 km/h. Diesen Wert schafft nicht einmal das Topmodell des Audi R8 mit 550 PS! Entsprechend kann aufgrund der begrenzten Bahnlänge diese Beschleunigung auch nur für 300 ms angewendet werden.

4.1.2 Const

Bei diesen Fahrsequenzen wird nach einer konstanten Beschleunigung von 100 ms nochmals explizit die Beschleunigung 0.0 m/s^2 für 400 ms angewendet, um somit in diesem Zeitbereich eine gleichbleibende Geschwindigkeit zu erhalten. Es werden dabei zwei verschiedene Beschleunigungen verwendet, vgl. Tabelle 4.2.

Name	Beschleunigung	Dauer
Const1	3.0 m/s^2	100 ms
	0.0 m/s^2	400 ms
Const2	5.0 m/s^2	100 ms
	0.0 m/s^2	400 ms

Tabelle 4.2: Beschleunigungsprofile „Const“

4.1.3 Distinctive

Nach den anfänglichen einfachen Profilen kommen nun die ersten markanten Beschleunigungsprofile zum Einsatz und der Schlitten wird etwas hin und her gefahren. Die Dauer der einzelnen Beschleunigungen bleibt hierbei gleich, lediglich die Beschleunigung wird bei diesen beiden Fahrversuchen jeweils variiert, siehe Tabelle 4.3.

Name	Beschleunigung	Dauer
Distinctive1	0.3 m/s^2	1000 ms
	-1.0 m/s^2	1000 ms
	1.5 m/s^2	1000 ms
	-2.0 m/s^2	1000 ms
Distinctive2	0.6 m/s^2	500 ms
	-2.0 m/s^2	500 ms
	3.0 m/s^2	500 ms
	-4.0 m/s^2	500 ms

Tabelle 4.3: Beschleunigungsprofile „Distinctive“

4.1.4 Upswing

Das Beschleunigungsprofil „Upswing“ basiert auf der Optimierung des Aufschwing- und Stabilisationsprozesses des Pendels mittels modellprädiktiver Regelung. Bei diesem Algorithmus wurde eine Samplezeit von 100 ms zugrunde gelegt, d.h. dass alle 100 ms eine neue Beschleunigung vorgegeben wird. Da die Optimierung mit dem „alten“ ungenauen Simulationsmodell durchgeführt wurde, erreicht man mit dieser Beschleunigungssequenz nicht die absolut aufrechte Position, nichtsdestotrotz handelt es sich um ein absolut repräsentatives Beschleunigungsprofil und man erkennt zumindest das Aufschwingen sehr gut. Es werden hier nur

die ersten fünf Sekunden des Upswing-Prozesses betrachtet, bei einem Beschleunigungstakt von 100 ms sind dies aber auch bereits 50 verschiedene Beschleunigungen, die angewendet werden. Da dies nicht mehr übersichtlich in einer Tabelle dargestellt werden kann, sei hier auf die graphische Veranschaulichung als Treppenfunktion in Abbildung 4.1 verwiesen.

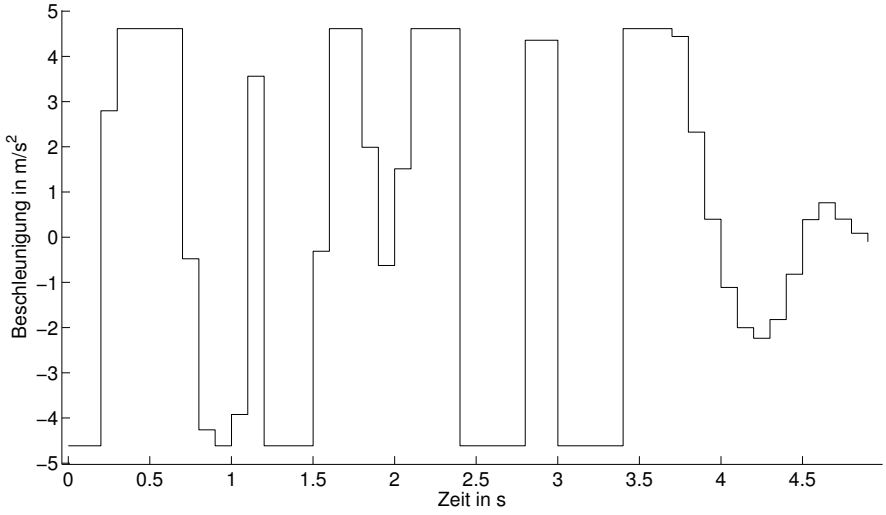


Abbildung 4.1: Beschleunigungsprofil „Upswing“

4.2 Aufzeichnung der Messdaten

Die aufgezeichneten Messdaten bilden die Grundlage für alle Auswertungen, weshalb die exakte Verarbeitung und Speicherung von sehr hoher Bedeutung ist. Um die Fahrversuche auch bzgl. einer Beschleunigungssequenz vergleichen zu können, wurde jeder Fahrversuch sechs Mal durchgeführt.

4.2.1 Formatierung

Bei jeder Messung werden zwei Dateien erzeugt. Zum einen eine Datei mit Namen „data_acc_*Beschleunigungsprofil*_*i*.txt“, wobei *Beschleunigungsprofil* durch einen Namen der oben genannten Beschleunigungssequenzen ersetzt wird und i die Nummer des jeweiligen Versuchs darstellt ($i \in \{1, ..., 6\}$). In dieser Datei wird festgehalten, zu welcher Zeit welche Beschleunigung angelegt wurde und wie lange diese angewendet wird, bevor der nächste Beschleunigungstakt beginnt. Bei uns wird stets eine Taktzeit von 100 ms vorgegeben. Tabelle 4.4 zeigt beispielhaft die Datei „data_acc_Const1_2.txt“. Man sieht bereits hier, dass die Beschleunigungen natürlich nicht exakt mit Beginn der Zeitmessung, sondern etwas (hier 1.5 ms) später starten. Der globale Zeitmesser und Taktgeber berücksichtigt dies jedoch und gibt auch die folgenden Beschleunigungen dementsprechend verzögert. Die Taktzeit von 100

% Time in s	Acc in m/s^2	Duration in s
+0.001560372	+3.000000000	+0.100327669
+0.101888041	+0.000000000	+0.099983962
+0.201872003	+0.000000000	+0.099986149
+0.301858152	+0.000000000	+0.099984190
+0.401842342	+0.000000000	+0.099719147

Tabelle 4.4: Datei „data_acc_Const1_2.txt“

ms kann dabei nicht absolut richtig eingehalten werden. Es hat sich aber herausgestellt, dass - wie auch bei den anderen Fahrversuchen - lediglich kleine Abweichungen von wenigen Zehntel Millisekunden auftreten.

Darüber hinaus wird eine zweite Datei mit Namen „data_pos_angle_*Beschleunigungsprofil_i*.txt“ erstellt, wobei *Beschleunigungsprofil* wiederum durch den Namen einer Beschleunigungssequenz ersetzt wird und i die Nummer des jeweiligen Versuchs darstellt ($i \in \{1, ..., 6\}$). In dieser Datei werden nun alle Messdaten bezüglich des Fahrversuchs gespeichert, d.h. der Zeitpunkt der Messung sowie die Position des Schlittens und der Winkel des Pendels zu diesem Zeitpunkt. Exemplarisch hierfür ist in Tabelle 4.5 der Anfang der Messdaten zum zweiten Versuch des Beschleunigungsprofils „Const1“ zu sehen. Wie man darin erkennen

% Time in s	Position in m	Angle in rad
+0.002351529	-0.000244190	+0.002617994
+0.002535834	-0.000244190	+0.002617994
+0.002708931	-0.000244190	+0.002617994
+0.002825108	-0.000244190	+0.002617994
⋮	⋮	⋮

Tabelle 4.5: Ausschnitt aus Datei „data_pos_angle_Const1_2.txt“

kann, setzt die Messung von Schlittenposition und Pendelwinkel etwas verspätet ein. Darüber hinaus lässt sich feststellen, dass im Durchschnitt (aller Messdaten) jede 0.12-0.13 ms ein Messpunkt erstellt wird. Dies entspricht knapp 8000 Messpunkten pro Sekunde und im Vergleich zur vorherigen Implementierung in C#, bei der man ca. alle 3 ms einen Messpunkt hatte, hat man nun mehr als das 20-fache an Messwerten. Diese hohe Abtastrate lässt bereits jetzt erhoffen, dass man aus den gewonnenen Daten wesentlich mehr Informationen als vorher gewinnen kann.

4.2.2 Aufbereitung der Messdaten

Da die Position motorbedingt ursprünglich ganzzahlig (in sogenannten Ticks) angegeben ist und durch das Programm in Meter umgerechnet wird, kommt es zu kleinen Ungenauigkeiten, weshalb die Positionsaufzeichnung nicht exakt im Nullpunkt startet. Da die Zeitmessung unabhängig von der Beschleunigungssequenz startet, fängt auch die Messdatenaufzeichnung

nicht exakt mit der Zeit Null an. Des Weiteren fällt auf, dass zumeist die erste Beschleunigung früher als die erste Positionsmessung stattfindet. Dies liegt darin begründet, dass nicht festgelegt ist, wann welchem Thread CPU-Zeit zugewiesen wird. In unserem Fall wird also meist zuerst die erste Beschleunigung gegeben, bevor anschließend die Messdatenaufzeichnung startet. Da die erste Positionsveränderung jedoch erst wesentlich später eintritt (siehe Abschnitt 5.1), spielt das keine Rolle.

Um es später beim Verarbeiten der Messdaten leichter zu haben, werden aus oben genannten Gründen mit Hilfe der Routine *preparing_data.m* sowohl die Zeit der Messaufzeichungen als auch die Positionmessungen in die Null geschoben. An diesem Punkt ist natürlich wichtig, diese Zeitkorrektur auch für die Zeitpunkte der Beschleunigungsgebung zu berücksichtigen. Ergebnis dieser Datenmanipulation ist also, dass sowohl Zeit als auch Position stets in der Null beginnen.

Auch die Messwerte des Pendelwinkels werden bearbeitet. Bei den Messwerten ist die untere Ruhelage der Nullpunkt, die aufrechte Position des Pendels entspricht somit dem Winkel π bzw. $-\pi$. Beim verwendeten Differentialgleichungsmodell wird aber Rücksicht auf die Arbeitsweise des Zustandsreglers genommen, der versucht, alle zu regelnden Zustandsgrößen gegen Null zu führen. Demzufolge entspricht in der Simulation die untere Ruhelage des Pendels dem Winkel π und die aufrechte Position dem Nullpunkt. In der bereits erwähnten MATLAB-Routine *preparing_data.m* werden die Winkeldaten deshalb so verändert, dass sie mit der Simulation übereinstimmen.

4.2.3 Heuristische Ermittlung der Messdatengenauigkeit

Um ein Gefühl für die später notwendige Genauigkeit bei der Modellbildung zu bekommen, soll zunächst untersucht werden, inwieweit Störungen das Fahren des Schlittens bzw. auch das Messen beeinflussen. Aus diesem Grund wurde jedes Beschleunigungsprofil sechs Mal angewendet, um somit für jede Beschleunigungssequenz eine Messreihe von sechs Datensätzen zu haben. Schon bei genauer Beobachtung der Durchführung der Messungen fällt auf, dass das Pendel bei schwierigeren Beschleunigungsprofilen bei verschiedenen Versuchen unterschiedlich weit aufschwingt.

Der Vergleich der je sechs Versuche von „Step1“, „Step2“ und „Step3“ zeigt, dass die Messwerte der Position am Ende des Versuchs (nach 500 bzw. 300 ms) alle innerhalb der Messgenauigkeit liegen und maximal 0.3 mm voneinander entfernt sind. In Abbildung 4.2 sind die einzelnen Stufen aufgrund der ganzzahligen[1] Messung der Systemzustände sehr gut zu erkennen.

Neben den „Step“-Versuchen sind auch die Messungen der Profile „Const1“ und „Const2“ sehr identisch und somit reproduzierbar. Hier treten zum Ende der Messung hin Unterschiede bis maximal 0.2 mm auf. Interessant ist jedoch die Tatsache, dass zu Beginn (im Bereich von 0.030-0.080 ms, siehe Zoom in Abbildung 4.3) Unterschiede bis zu 0.65 mm in den Versuchen auftreten, die kurz darauf bereits nicht mehr zu identifizieren sind. Hier sieht man also sehr deutlich das Wirken des PID-Reglers im Antriebsverstärker des Servomotors, der diese aufgrund von Hardwarelatenzen auftretenden Differenzen schon nach sehr kurzer Zeit wieder ausregelt.

[1] durch Auflösung des Motordrehgebers definiert

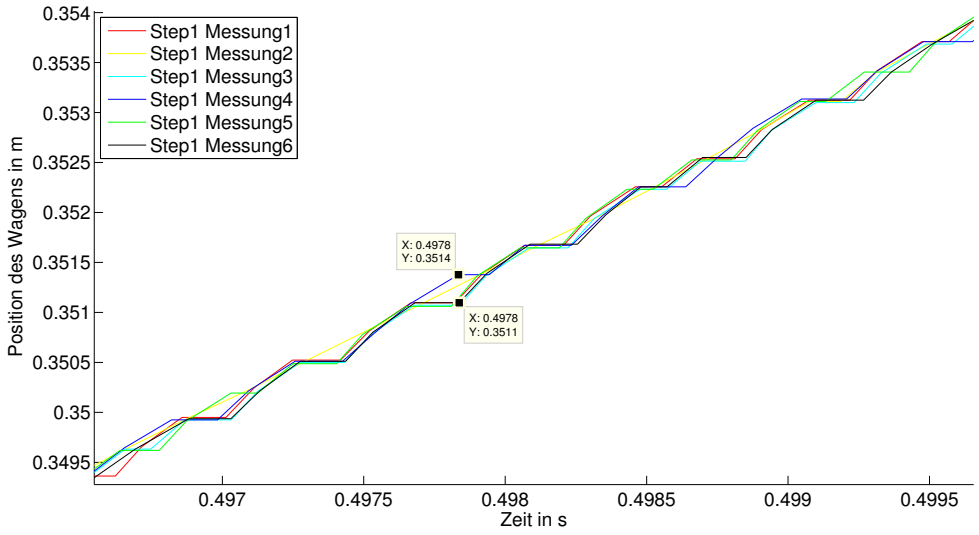


Abbildung 4.2: Vergleich der Schlittenposition bei verschiedenen Messungen von „Step1“

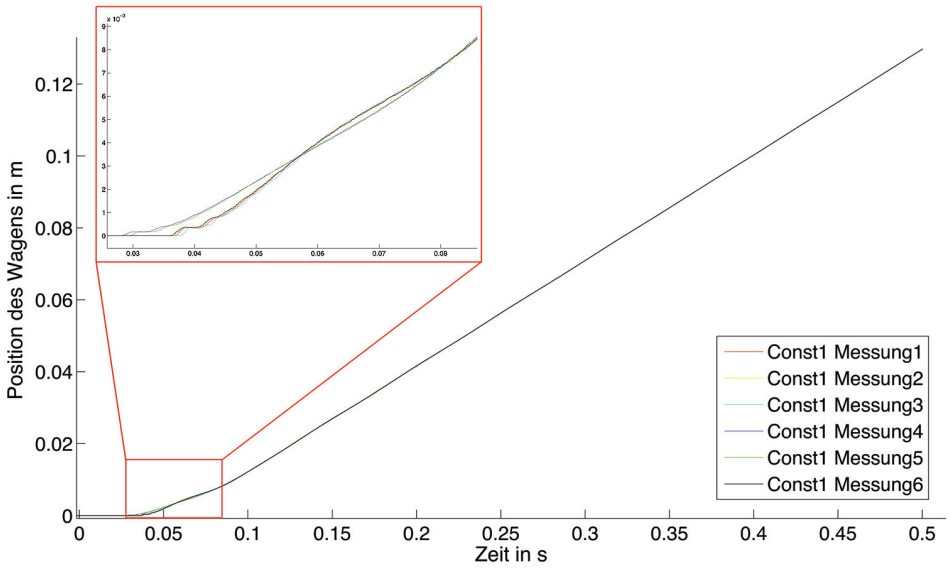


Abbildung 4.3: Vergleich der Schlittenposition bei verschiedenen Messungen von „Const1“

Beim längsten Beschleunigungssprofil „Upswing“ beträgt der maximale Unterschied in der Position des Schlittens zwischen zwei Fahrversuchen am Ende der Steuerungssequenz über 13 mm (vgl. Abbildung 4.4). Erklärung hierfür ist, dass der Beschleunigungstakt von 100 ms nicht exakt eingehalten werden kann, sondern einzelne Beschleunigungen ein paar Zehntel

Millisekunden zu früh bzw. zu spät gegeben werden. Dieser Fehler pflanzt sich entsprechend fort, sodass bei längeren Sequenzen die Unterschiede in der Position deutlicher ausfallen.

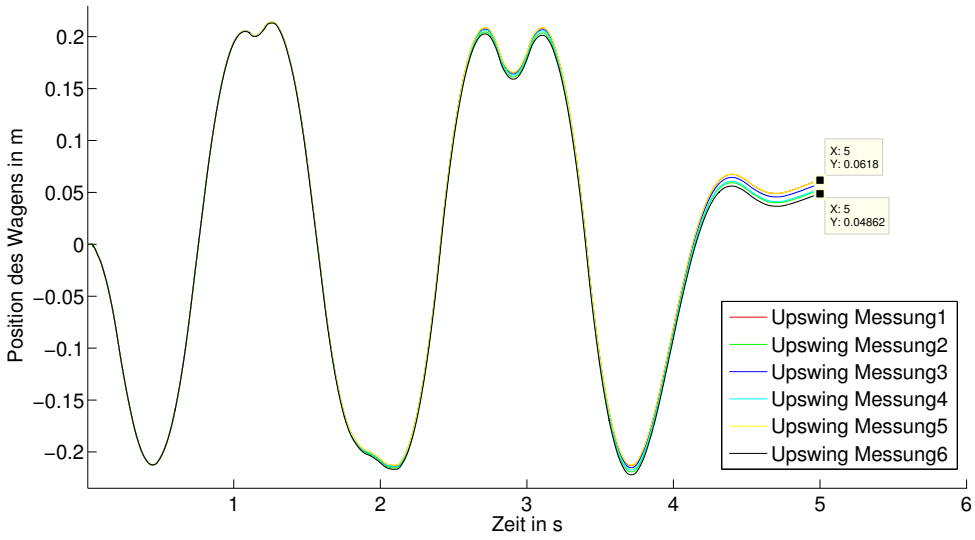


Abbildung 4.4: Vergleich der Schlittenposition bei verschiedenen Messungen von „Upswing“

Dementsprechend kann hier festgehalten werden, dass die einfachen Fahrversuche über längeren Zeitraum gesehen durchaus identisch ablaufen, im Bereich bis 80 ms jedoch unterschiedlich „anfahren“. Da es bei den längeren Fahrprofilen grundsätzlich größere Differenzen aufgrund der unterschiedlichen Taktlänge gibt, empfiehlt es sich bei diesen Fahrprofilen jede Messung als eigenständigen Versuch zu betrachten. Lediglich die Messungen von „Step“ sind nach dem Aufbereiten der Messdaten wie in Abschnitt 4.2.2 beschrieben miteinander vergleichbar. Aufgrund der dort beobachteten geringen Unterschiede kann man davon ausgehen, dass auch bei den längeren Sequenzen wenige Störungen bzw. geringes Rauschen auftritt und es deshalb durchaus legitim ist, eine einzelne Messung einer Datenreihe als Referenz für das neue Modell herzunehmen. Unter Rauschen versteht man dabei die Erscheinung, dass sich selbst einem statischen Messsignal zeitlich fluktuierende Störungen überlagern, die wiederum vielfältige Ursachen haben können. [8]

4.2.4 Hinweise zur Implementierung der Beschleunigungsprofile im Simulationsmodell

Wie im letzten Abschnitt beschrieben, kann bei den Fahrversuchen die vorgegebene Taktzeit von 100 ms nicht absolut exakt eingehalten werden. Aus diesem Grund ist es wichtig, der Simulation den Zeitpunkt mitzuteilen, zu der in der Messung eine neue Beschleunigung beginnt. Deshalb wird eine Matrix mit den Daten aus Tabelle 4.4 mit an die Differentialgleichungsfunktion in MATLAB übergeben. Durch Abfragen der Zeit wird dann die zu diesem Zeitpunkt aktive Beschleunigung verwendet.

Auch wenn diese pragmatische und einfach zu realisierende Lösung, die Parameter in Abhängigkeit von der Zeit (oder auch Ort und Geschwindigkeit) direkt im ode-File jeweils an die aktuellen Werte anzupassen, nicht „ganz sauber“ ist, funktioniert sie in den meisten Fällen. So konnten in dieser Arbeit diesbezüglich in keinem Fall Probleme festgestellt werden. Probleme könnten theoretisch dadurch auftauchen, dass die Lösungsalgorithmen in einem Zeitschritt mehrere Funktionswertberechnungen zu verschiedenen Zeitpunkten abfordern (beim Runge-Kutta-Verfahren 4. Ordnung z. B. am Anfang, in der Mitte und am Ende des Zeitschritts). Die Parameteränderung kann (und wird in der Regel) innerhalb eines Zeitschritts liegen, so dass die Funktionswertberechnungen mit unterschiedlichen Parametern ausgeführt werden. Das hat zur Folge, dass die MATLAB-odefunction die Schrittweite an dieser Stelle ständig verfeinert und dies somit zu einem „Festfahren“ der Rechnung führt. Sollten solche Probleme auftreten, könnte man diese mit der MATLAB-Event-Location der ode-Solver abfangen. Weitere Details können unter `http://www.mathworks.de/de/help/matlab/ref/odeset.html#f92-1017470` nachgesehen werden.

Kapitel 5

Auswertungen

Nach Durchführung der im vorherigen Kapitel vorgestellten Fahrversuche werden nun die zugehörigen Messdaten analysiert und untersucht, um Rückschlüsse auf das reale Modell ziehen zu können. In diesem Kapitel wird dabei ausschließlich der Wagen betrachtet. Die Auswertungen in dieser Arbeit werden dabei stets am Fahrprofil „Step1“ erläutert und demonstriert. Um den Rahmen der vorliegenden Arbeit nicht zu sprengen, soll auf Ausführungen der äquivalenten Ergebnisse der anderen Fahrversuche verzichtet werden.

5.1 Verzögerungszeit

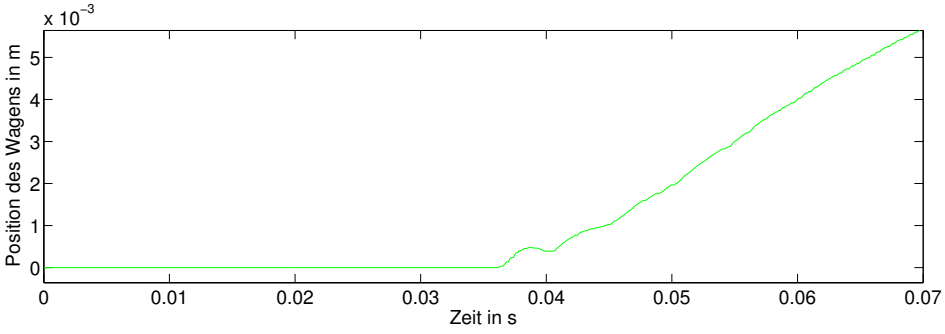


Abbildung 5.1: Ausschnitt der Messwerte des Fahrversuchs „Step1“

Schon beim ersten Betrachten der aufgenommenen Messdaten fällt auf, dass es relativ lange dauert, bis sich der Wagen das erste Mal bewegt, siehe Abbildung 5.1. Bei den vorliegenden 48 Datensätzen (8 Fahrprofile à 6 Messungen) liegt der arithmetische Mittelwert dieser Verzögerung bei

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} t_i^{\text{verzögerung}} = 34.527 \text{ ms}. \tag{5.1}$$

Man erhält eine Standardabweichung von

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2} = 4.420 \text{ ms}. \tag{5.2}$$

Die besondere Normierung mit $n-1$ sorgt dafür, dass die Stichprobenvarianz erwartungstreu ist. Diese Eigenschaft verliert man, wenn stattdessen mit n normiert wird. [6] Die minimale Verzögerung ist 22.268 ms und die maximale Verzögerung 37.882 ms. Die meisten Verzögerungen liegen im Bereich von 36 ms, es gibt jedoch einige Ausreißer nach unten (vgl. Abbildung 4.3), die deshalb den Mittelwert etwas vermindern.

Darüber hinaus fällt auf, dass die Ortskurve anschließend nicht – wie anzunehmen wäre – quadratisch zunimmt, sondern sprungartig ansteigt und etwas hin- und herschwingt. Das deutet daraufhin, dass der integrierte Controller merkt, wie weit er inzwischen von der Solltrajektorie abgewichen ist und deshalb durch einen Sprung versucht, diese Differenz wieder wettzumachen.

Diese doch sehr lange Verzögerung von durchschnittlich 34 ms zu Beginn ist demnach nicht nur durch Hardwarelatenzen zu erklären, sondern ist vielmehr auch einer Regelungenauigkeit und Trägheit des PID-Reglers geschuldet. Ist anfangs etwa noch die Regeldifferenz für ein Einschreiten des Reglers zu klein, so vergrößert sich diese so schnell (d.h. quadratisch), dass der Controller dies nur durch einen extremen Sprung einholen kann. Dazu kommt noch die Haftreibung des Wagens, die es anfangs zu überwinden gilt. All diese Punkte, insbesondere die Ungewissheit über den Zeitpunkt, an dem der Controller die Beschleunigungsvorgabe erhält, werden die Modellierung dieser Latenz schwierig machen. Eine schematische Darstellung der Zusammensetzung dieser Gesamtverzögerung befindet sich in Abbildung 5.2. Da es sich beim benutzten Servoantrieb von Panasonic jedoch um eine Standard-Industrielösung handelt, gehen wir davon aus, dass die Zeit x, bis der Controller den Kontrollwert erhält, vergleichsweise gering ist. Aus diesem Grund wird diese Hardwarelatenz im Weiteren vernachlässigt[1], während versucht wird, die Systemträgheit durch das später eingeführte Modell aufzufangen.

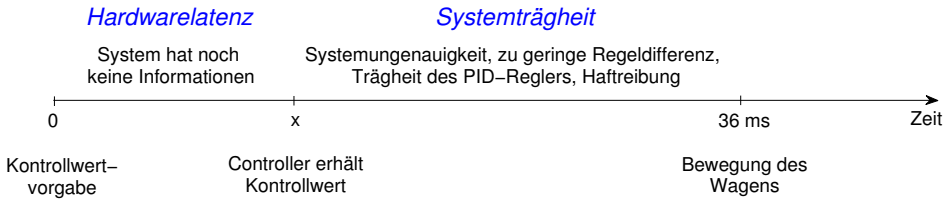


Abbildung 5.2: Schematische Darstellung der Verzögerung des Wagens

[1] Alternativ wäre eine Berücksichtigung mittels sogenannter „Retardierter Differentialgleichungen“ möglich.

5.2 *Exkurs:* Filter

In vielen Anwendungen misst man Variablen, die sowohl wenig variieren als auch durch eine Störgröße beeinflusst werden. Es ist dann oftmals wünschenswert ein glättendes Filter auf die Messdaten anzuwenden, um die zugrundeliegende glatte Funktion rekonstruieren zu können. Deshalb werden in diesem Abschnitt zwei Filter vorgestellt, die später zum Einsatz kommen werden. Diese sind auch ohne Kenntnisse der digitalen Signalverarbeitung leicht zu verstehen, mathematisch interessant und leisten für unsere Zwecke sehr gute Dienste. Dieser Abschnitt orientiert sich dabei an [26] und [9].

Die Messwerte werden in dieser Arbeit mit $x_i = x(t_i)$ und die geglätteten Daten zur Zeit t_i mit g_i bezeichnet. Darüber hinaus wird angenommen, dass das Rauschen unabhängig von der untersuchten Variable ist. Vor Anwendung der Filter ist es außerdem notwendig, die vorhandenen (nicht äquidistanten) Messwerte zu bearbeiten, um konstante Abstände in der Zeitmessung zu erhalten. Andernsfalls wäre die Gewichtung der einzelnen Messwerte beim Filtern verzerrt. Es wird hierfür die MATLAB-Funktion `interp1` verwendet, die an den Stützstellen des neuen äquidistanten Gitters eine lineare Interpolation der vorhandenen Messdaten durchführt. Aufgrund der sehr hohen Leserate und sowie bereits fast gleichverteilten Messpunkte treten hier nur Abweichungen von weniger als 0.1 mm von den ursprünglichen Messdaten auf. Man kann sich also trotz dieser vermeintlich ungenauen Methode mehr als zufrieden geben. Man betrachte hierzu Abbildung 5.3 und beachte insbesondere die Skala der y-Achse.

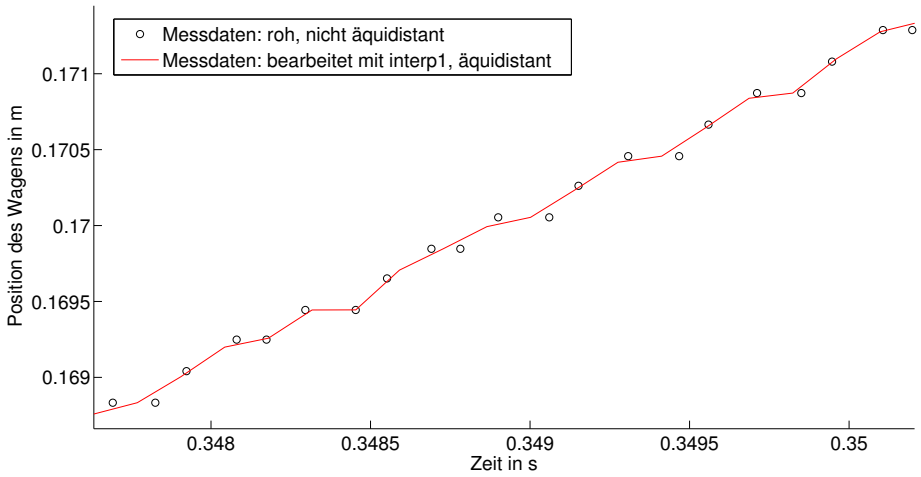


Abbildung 5.3: Äquidistante Messdaten nach Manipulation mit `interp1`

5.2.1 Moving-Average-Filter

Der einfachste Ansatz um Datensätze zu glätten, ist das Anwenden des „Moving Average“-Filters[2]. Die Idee ist, den (evtl. gewichteten) Durchschnitt der benachbarten Daten von

[2]auch „gleitendes Mittel“ oder „gleitender Durchschnitt“

Datenpunkt x_i als geglätteten Wert g_i zu wählen. Nach der Berechnung eines geglätteten Werts wird anschließend das „Fenster“ der betrachteten Daten um einen Zeitschritt weitergeschoben und der nächste gefilterte Wert berechnet.

Hierbei handelt es sich um einen Tiefpassfilter mit endlicher Impulsantwort[3]. Durch seine Tiefpasscharakteristik werden kurzfristige (d.h. hochfrequente) Veränderungen in einer Zeitreihe geglättet und längerfristige (d.h. niederfrequente) Trends und Zyklen hervorgehoben. Der gleitende Mittelwert $g(t)$ der Ordnung n (n ungerade, d.h. $n = 2m+1$ mit $m \in \mathbb{N}$) einer Zeitreihe $x(t)$ ist definiert als:

$$g_x^n(t_i) = \sum_{k=0}^{n-1} w_k \, x_{i-m+k} \tag{5.3}$$

wobei w_k die Gewichtung der jeweiligen Datenpunkte darstellt und für diese gilt $\sum_{k=0}^{n-1} w_k = 1$. Mathematisch kann die gleitende Mittelwertbildung als Faltung der Messdaten mit den Filterkoeffizienten verstanden werden.

Wir verwenden den einfachen gleitenden Durchschnitt, der alle Datenpunkte gleich gewichtet, d.h. es gilt

$$w_k = \frac{1}{n} \quad \forall\, k = 1, ..., n$$

und somit

$$g_x^n(t_i) = \frac{1}{n} \sum_{i=0}^{n-1} x_{i-m+k}.$$

Dieser gleitende Mittelwert hat eine Verzögerung von $m = \frac{1}{2}(n-1)$ Zeiteinheiten, d.h. die gemittelten Werte „hinken“ um diese Anzahl Zeiteinheiten hinterher. Die Verzögerung kann korrigiert werden, indem man den gleitenden Mittelwert um diese Anzahl Zeiteinheiten nach vorne verschiebt. Entsprechend fehlen anschließend jedoch die Werte für die letzten Zeiteinheiten. [18]

In MATLAB heißt der Befehl zum Anwenden des Filters

```
> g_filtered = filter(w,1,x)
```

wobei die Ordnung n des Filters durch die Länge des Gewichtungsvektors w bestimmt wird.

5.2.2 Savitzky-Golay-Glättungsfilter

Das Savitzky-Golay-Filter ist ein mathematisches Glättungsfilter in der Signalverarbeitung, welches erstmals 1964 von Abraham Savitzky and Marcel J. E. Golay beschrieben wurde. Diese Veröffentlichung wird von einigen Autoren als eine der wichtigsten und meistzitierten Grundlagenveröffentlichung im Bereich der computergestützten Numerik eingeschätzt. [28]

Die Idee ist, im Gegensatz zum gleitenden Mittelwert nicht einfach den Durchschnitt der benachbarten Daten als geglätteten Wert g_i im Punkt t_i zu nehmen, sondern eine bestimmte

[3] auch „FIR-Filter“, englisch für finite impulse response filter

Anzahl von Nachbar-Messwerten x_i mittels eines Polynoms eines gewissen Grades M (d.h. es sind mindestens $M+1$ äquidistante Stützstellen nötig) zu fitten. Dann ergibt der Wert des Polynoms an der Stelle t_i den geglätteten Wert g_i. Anschließend wird das „Fenster“ der betrachteten Messwerte um einen Zeitschritt nach rechts verschoben und ein neues Polynom gesucht. Diese Idee ist in Abbildung 5.4 mit einem Polynom 2. Grades ersichtlich, wobei n_L die Anzahl der Punkte zur Linken von t_i und n_R die Anzahl der Punkte zur Rechten von t_i bezeichnet. Mit $p_i(t)$ bezeichnen wir das Polynom vom Grad M, das den Kleinste-Quadrate-Fehler durch die $n_L + n_R + 1$ verrauschten Datenpunkte um den Zeitpunkt t_i minimiert. Es gilt somit $g_i = p_i(t_i)$.

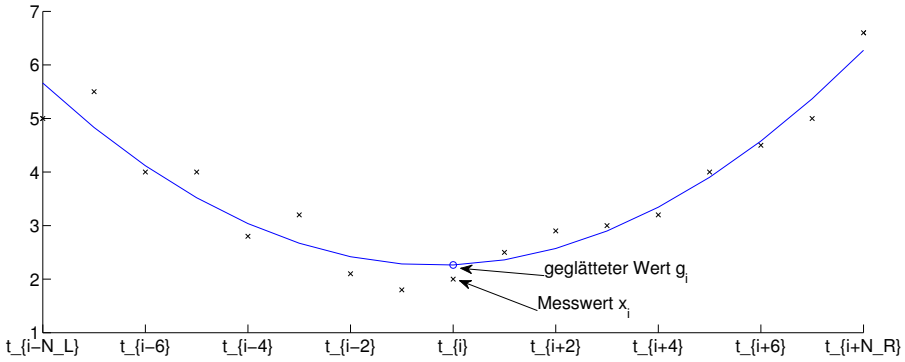


Abbildung 5.4: Kleinste-Quadrate-Polynom eines Savitzky-Golay-Filters zum Zeitpunkt t_i

Ein Vorteil des Savitzky-Golay-Filters ist, dass anders als bei anderen Glättungsfiltern Anteile von hohen Frequenzen nicht einfach abgeschnitten werden, sondern in die Berechnung mit einfließen. Dadurch neigt das Filter dazu, ausgezeichnete Eigenschaften der Verteilung wie relative Maxima, Minima und Streuung zu erhalten, die von herkömmlichen Methoden wie der Bildung des gleitenden Mittels gewöhnlich durch Abflachung oder Verschiebung verfälscht werden.

Das Verfahren scheint sehr aufwändig zu sein, da für jeden Datenpunkt ein Ausgleichspolynom berechnet werden muss. Man kann jedoch zeigen, dass dies nicht nötig ist und dass stattdessen eine geeignete Filterfunktion verwendet werden kann. Aus diesem Grund soll hierauf etwas näher eingegangen werden. Das gefittete Polynom $p_i(t)$ vom Grad M durch die Messdaten x_i kann zunächst geschrieben werden als

$$p_i(t) = \sum_{k=0}^{M} b_k \left(\frac{t - t_i}{\Delta t} \right)^k . \tag{5.4}$$

Wir nehmen an, dass die Stützstellen t_i äquidistant sind mit $t_{i+1} - t_i = \Delta t$. Um das Polynom im Kleinste-Quadrate-Sinn durch die Messdaten zu legen, muss man die Koeffizienten b_k so bestimmen, dass gilt

$$\sum_{j=i-n_L}^{i+n_R} (p_i(t_j) - x_j)^2 = \min. \tag{5.5}$$

Dazu definieren wir die Matrix

$$A := \begin{pmatrix} (-n_L)^M & \dots & -n_L & 1 \\ \vdots & & \vdots & \vdots \\ 0 & \dots & 0 & 1 \\ \vdots & & \vdots & \vdots \\ n_R^M & \dots & n_R & 1 \end{pmatrix} \in \mathbb{R}^{(n_L+n_R+1)\times(M+1)} \tag{5.6}$$

und die Vektoren

$$\mathbf{b} := \begin{pmatrix} b_M \\ \vdots \\ b_1 \\ b_0 \end{pmatrix} \in \mathbb{R}^{M+1} \quad \text{und} \quad \mathbf{x} := \begin{pmatrix} x_{i-n_L} \\ \vdots \\ x_i \\ \vdots \\ x_{i+n_R} \end{pmatrix} \in \mathbb{R}^{n_L+n_R+1}. \tag{5.7}$$

Man sieht, dass die Matrix A weder von der Abszisse t_i noch vom Abstand Δt abhängt.

Mit diesen Definitionen können wir (5.5) reformulieren als

$$\|A\mathbf{b} - \mathbf{x}\|_2 = \min. \tag{5.8}$$

Mittels QR-Zerlegung von A wäre es nun möglich, diese Gleichung für $\mathbf{b}$ zu lösen. Für unser Vorhaben, die Daten zu glätten, brauchen wir jedoch nur $g_i = p_i(t_i) = b_0$ zu kennen. Die Lösung $\mathbf{b}$ von (5.8) kann auch als Lösung der Normalengleichung

$$A^T A\mathbf{b} = A^T\mathbf{x} \tag{5.9}$$

betrachtet werden. Somit erhalten wir

$$g_i = \mathbf{e}_{M+1}^T (A^T A)^{-1} A^T \mathbf{x}, \tag{5.10}$$

wobei $\mathbf{e}_{M+1}^T$ den transponierten $(M+1)$-sten Einheitsvektor bezeichnet.

Offensichtlich können wir g_i als Linearkombination der x_i schreiben. Wir definieren den Vektor

$$\mathbf{c} := A(A^T A)^{-1}\mathbf{e}_{M+1}, \tag{5.11}$$

der die Filterkoeffizienten $c_{-n_L}, ..., c_{n_R}$ enthält. Da $\mathbf{c}$ nicht von t_i und Δt abhängt, muss es lediglich einmal ausgewertet werden. Anschließend können alle geglätteten Werte g_i einfach durch das Skalarprodukt

$$g_i = \mathbf{c}^T\mathbf{x} = \sum_{j=i-n_L}^{i+n_R} c_{j-i}x_j \tag{5.12}$$

berechnet werden.

Die Berechnung des Vektors $\mathbf{c}$ in (5.11) kann für einen großen Polynomgrad M jedoch Probleme bereiten, da sich die Kondition beim Bilden von $A^T A$ verschlechtert. Stattdessen

verwendet man für Gleichung (5.8) üblicherweise die QR-Zerlegung $A = QR$. Setzt man dies nun in (5.11) ein, so erhält man

$$\mathbf{c} = \frac{1}{r_{M+1,M+1}} Q\mathbf{e}_{M+1} \tag{5.13}$$

und der Polynomwert am Punkt t_i ergibt sich schließlich durch

$$b_0 = \frac{1}{r_{M+1,M+1}} \mathbf{e}_{M+1}^T Q^T x = \frac{1}{r_{M+1,M+1}} \sum_{j=i-n_L}^{i+n_R} q_{j-i,m+1} x_j. \tag{5.14}$$

Ein weiterer großer Vorteil des Savitzky-Golay-Filters ist somit seine Geschwindigkeit. Für gegebenes n_L, n_R und M müssen die Filterparameter $\mathbf{c}$ nur ein einziges Mal ausgerechnet werden. Anschließend können alle geglätteten Werte g_i durch das einfache Skalarprodukt (5.12) der Länge $n_L + n_R + 1$ berechnet werden. Nachteilig ist, dass es nicht offensichtlich ist, wie die Filterparameter n_L, n_R und M zu wählen sind. Hier ist in den meisten Fällen etwas visuelle Optimierung von Nöten, um die besten Resultate zu erhalten.

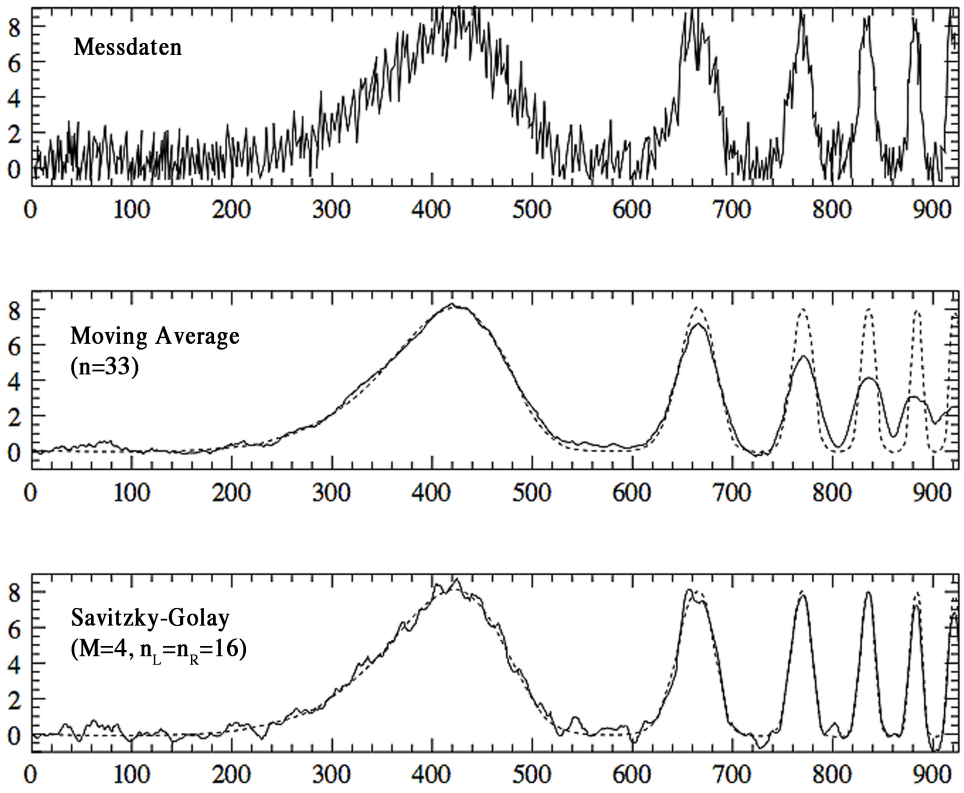


Abbildung 5.5: Oben: Messdaten. Mitte: Ergebnis bei Anwendung des Moving-Average-Filters mit 33 Datenpunkten. Unten: Ergebnis bei Anwendung des Savitzky-Golay-Filters (vom Grad 4 und mit ebenfalls 33 Datenpunkten). Die gestrichelte Linie stellt das ursprüngliche glatte Signal dar. Quelle: [26]

Das Glättungsfilter, das auch digitales polynomiales Glättungsfilter oder Kleinste-Quadrate Glättungsfilter genannt wird, wird typischerweise benutzt um ein verrauschtes Signal zu glätten, dessen Frequenzbandbreite (ohne Rauschen) hoch ist. Da wir die Messwerte des Wagens nicht nur auf die niederfrequenten „groben“ Positionsveränderungen reduzieren wollen, sondern uns insbesondere für die kleinen, aussagekräftigen, hochfrequenten „Ruckler“ interessieren, ist dieser Filter für uns hilfreich. In diesem Fall beweist sich die Vorgehensweise von Savitzky-Golay als viel besser im Vergleich zu Standard-FIR-Filter, die dazu tendieren neben der Störung auch einen großen Anteil des hochfrequenten Signals mit auszulöschen. Jedoch sind Savitzky-Golay-Filter entsprechend beim Aussondern des Rauschens weniger erfolgreich. Aus diesem Grund werden wir später eine Kombination aus beiden Filtern verwenden. In Abbildung 5.5 erkennt man deutlich, dass bei Anwendung des Moving-Average-Filters auf Basis von 32 benachbarten Datenpunkten die Spitzen des Messsignals deutlich abgeflacht sind und an Amplitude verlieren, wohingegen bei Anwendung des Savitzky-Golay-Filters die Amplituden in Breite und Höhe erhalten bleiben, die Kurve jedoch insgesamt auch weniger geglättet wurde. Darüber hinaus zeichnet sich das Savitzky-Golay-Filter dadurch aus, dass es keine Latenz verursacht und somit die Messdaten zeitlich nicht verfälscht.

In MATLAB wird das Savitzky-Golay-Filter bei den eingeführten Variablenbezeichnungen mit dem Befehl

```
> g_filtered = sgolayfilt(x,M,n_L+n_R+1)
```

angewendet.

5.3 Geschwindigkeit des Schlittens

Da wir in unserem System nur die Position des Schlittens messen können, aber keinerlei Informationen über die Geschwindigkeit bzw. die Beschleunigung des Wagens haben, müssen wir uns der Numerik bedienen, um diese interessanten Informationen zu erhalten.

5.3.1 Ermittlung durch Differenzenquotient

Naheliegendste und einfachste Option für die näherungsweise Bestimmung der zeitlichen Ableitung einer Funktion ist der Differenzenquotient. Für eine etwas bessere Näherung verwenden wir den zentralen Differenzenquotienten

$$\frac{\Delta x}{\Delta t} = \frac{x(t_i + \Delta t) - x(t_i - \Delta t)}{2\Delta t} = x'(t_i) + \mathcal{O}(\Delta t^2), \tag{5.15}$$

d.h. die Steigung der Sekante zwischen den beiden Nachbar-Messpunkten. Δt bezeichnet die äquidistante Schrittweite $t_i - t_{i-1}$.

Eine Bemerkung zur Genauigkeit des Differenzenquotienten: Definieren wir für unsere Messwerte eine universelle Messfehlerschranke ε, d.h. $|\varepsilon_i| \leq \varepsilon \; \forall$ Messfehler ε_i, so erhält man

$$\begin{aligned} x'(t) &= \frac{x(t_i + \Delta t) + \varepsilon_{i+1} - x(t_i - \Delta t) - \varepsilon_{i-1}}{2\Delta t} + \mathcal{O}(\Delta t^2) \\ &= \frac{x_{i+1} - x_{i-1}}{2\Delta t} + \frac{\varepsilon_{i+1} - \varepsilon_{i-1}}{2\Delta t} + \mathcal{O}(\Delta t^2). \end{aligned} \tag{5.16}$$

Der Fehler der zentralen Differenzenformel wäre somit maximal für $\varepsilon_{i+1} = \varepsilon$ und $\varepsilon_{i-1} = -\varepsilon$, er wäre dann

$$\frac{\varepsilon}{\Delta t} + \mathcal{O}(\Delta t^2). \tag{5.17}$$

Betrachtet man diese Fehlersumme betragsmäßig, so stellt man fest, dass der zweite Summand mit Verringerung der Schrittweite Δt abnimmt, der erste aber zunimmt. Ihre Summe besitzt deshalb ein positives Minimum in einem gewissen $\Delta t =: h_0$ und damit kann man den theoretisch beliebig verkleinerbaren Fehler in diesem praktischen Fall nicht unter eine gewisse Grenze senken. [29] Referenzversuch „Step1“ besitzt beispielsweise eine Schrittweite von $1.3697 \cdot 10^{-4}$ s, d.h. der zweite Summand (der eigentliche Fehlerterm) spielt auch bei sehr hohen zweiten Ableitungen (d.h. sehr hohen Beschleunigungen) keine tragende Rolle. Der Gesamtfehler der Näherung wird also vom Messfehler, dem ersten Summanden, dominiert. Aus diesem Grund kommt man auch um ein starkes Filtern der Messdaten nicht herum, damit sich die Messwerte gegenseitig etwas angleichen und somit der Messfehler relativiert wird.

Um die Fähigkeiten der Filter später noch besser einschätzen zu können, soll hier dennoch auch kurz das Ergebnis des Differenzenquotienten bei Anwendung auf die rohen Messdaten gezeigt werden. In Abbildung 5.6 sieht man wie erwartet das starke Messrauschen und somit nicht hilfreiche Bild bei Anwendung des Differenzenquotienten ohne vorheriges Filtern.

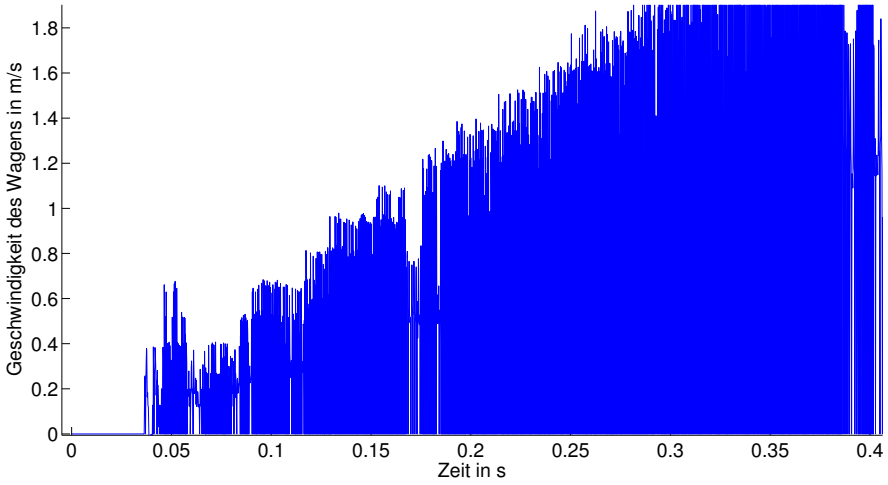


Abbildung 5.6: Ermittelte Geschwindigkeit mittels Differenzenquotient mit den ungefilterten Messdaten aus „Step1“

Aus diesem Grund wird zunächst das Moving-Average-Filter aus dem vorherigen Abschnitt angewendet, siehe Abbildung 5.7. Auch wenn man bei einer kleinen Filterlänge von zehn Werten immer noch einiges an Rauschen hat, kann man schon erkennen, dass das Zentrum des Rauschens einer linearen Asymptote folgt, was grundsätzlich schon einmal korrekt ist

(Beschleunigung konstant ⇒ Geschwindigkeit steigt linear). Zudem sieht man, dass das Rauschen umso größer wird, je größer die Geschwindigkeit ist. Das lässt sich dadurch erklären, dass die Messungen umso ungenauer werden, je größer die Geschwindigkeit des Schlittens ist. Mit der Fülle an vorliegenden Messdaten ist es möglich, die Daten auch etwas gröber zu filtern. Erhöht man die Filterlänge, so erhält man eine durchaus „schöne“ Gerade. Man beachte jedoch den Verlust des charakteristischen Einschwingens zu Beginn. Wie bereits in Abschnitt 5.1 gesehen, fährt der Schlitten zwar verspätet los, versucht dann aber durch erhöhte Geschwindigkeit den Zeitverlust wieder einzuholen, um schlussendlich zumindest in der Nähe der richtigen Geschwindigkeitstrajektorie zu landen.

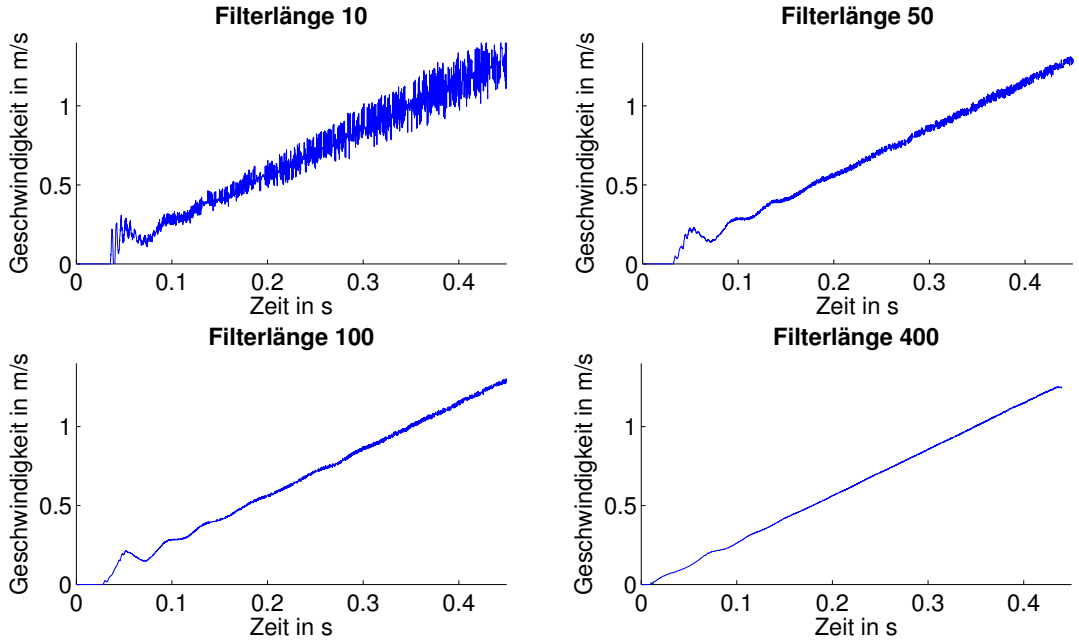


Abbildung 5.7: Ermittelte Geschwindigkeit mittels Differenzenquotient mit Messdaten aus „Step1“ nach Anwendung des Moving-Average-Filters mit verschiedenen Filterlängen

Bei einer Filterlänge von 400 Datenpunkten liegen die betrachteten Messwerte jedoch schon über 50 ms auseinander, d.h. die Daten werden bereits enorm geglättet und das Ergebnis muss nicht mehr unbedingt der Realität entsprechen. Aus diesem Grund wird die Filterlänge des Moving-Average-Filters möglichst gering gehalten und zusätzlich das Glättungsfilter von Savitzky-Golay angewendet. Wie oben bereits erwähnt, erhält dieses die charakteristischen (d.h. höherfrequentigen) Kurven der Messdaten. Bei Anwendung dieses zusätzlichen Glättungsfilters hat man zwei Stellschrauben, an denen man drehen kann: zum einen die „Fenstergröße“, d.h. die Anzahl an Datenpunkten, die bei der Berechnung mit einbezogen werden, zum anderen den Grad des angepassten Polynoms. Durch Anwenden des Filters kann das Rauschen stark reduziert werden. Aufgrund des Polynomfittings sind die geglätteten Daten umso „wackeliger“, je höher der Polynomgrad ist und je weniger Datenpunkte

man betrachtet, da sich in diesem Fall das Polynom den verrauschten Daten am ehesten anschmiegen kann. Wie im letzten Abschnitt beschrieben, ist an dieser Stelle visuelles Optimieren nötig, um ein akzeptables Ergebnisbild zu erhalten. Als gute Werte haben sich dabei ein Polynom dritten Grades sowie eine Fensterbreite von etwa 61 Datenpunkten herausgestellt.

Das beste Ergebnis wurde dabei erzielt, wenn man vorher das Moving-Average-Filter der Länge 50 angewendet hat. Vergleicht man das Ergebnis, das in Abbildung 5.8 zusehen ist, nun mit dem rechten oberen Bild in Abbildung 5.7, so ist deutlich zu erkennen, wie sehr das Rauschen abgenommen hat, das charakteristische Einschwingen zu Beginn jedoch erhalten bleibt. Auf den vorgeschalteten gleitenden Mittelwert kann hierbei nicht verzichtet werden. Wendet man lediglich das Savitzky-Golay-Filter an, so ist das Ergebnis ebenso wenig zufriedenstellend.

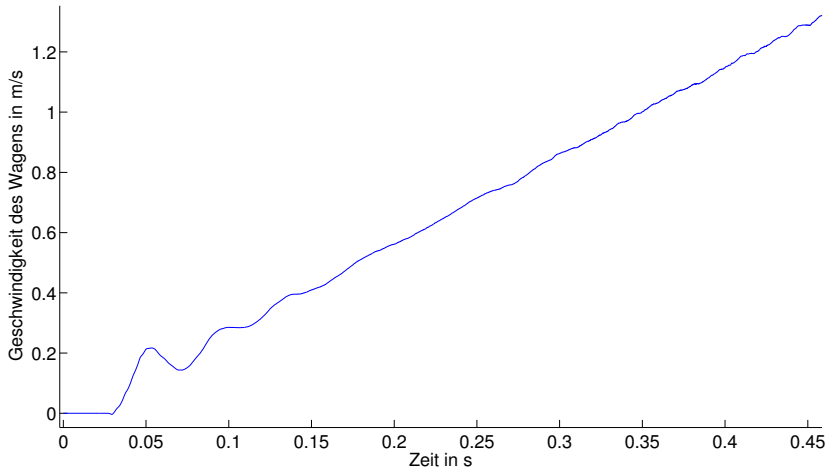


Abbildung 5.8: Ermittelte Geschwindigkeit mittels Differenzenquotient nach Anwendung des Moving-Average- und des Savitzky-Golay-Filters

5.3.2 Ermittlung durch Kurvenanpassung mittels Savitzky-Golay-Filter

Da man bei Benutzung des Savitzky-Golay-Filters in jedem Messpunkt implizit eine Kurvenanpassung mit Polynomen durchführt, bietet es sich natürlich an, gleich dieses Polynom zu verwenden, um durch algebraisches Ableiten die Differentiation durchzuführen und entsprechend die Geschwindigkeit zu erhalten. Die MATLAB-Routine `sgolay` bietet die benötigten Polynomkoeffizienten der Ableitungen direkt als Ausgabe mit an. Abbildung 5.9 zeigt demnach in jedem Messpunkt die erste Ableitung des gefitteten Polynoms. Dadurch erhält man qualitativ das gleiche Bild wie in Abbildung 5.8, lediglich im Detail erkennt man die etwas rundere Kurve bei der Ableitung mit Hilfe des Polynomfittens. Dieser geringe Unterschied lässt sich durch die vorherige Anwendung des Moving-Average-Filters erklären. Dieser hat

die Messwerte bereits soweit aneinander angepasst, dass sich die Steigung des Polynoms nur noch geringfügig vom zentralen Differenzenquotient unterscheidet.

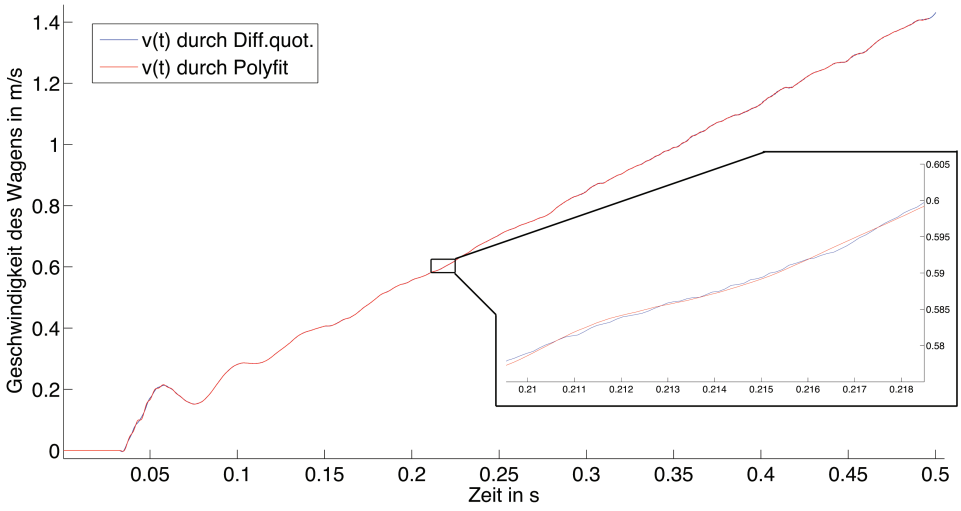


Abbildung 5.9: Vergleich der ermittelten Geschwindigkeit durch Differenzenquotient und durch Polynomfitting (im Savitzky-Golay-Filter)

5.3.3 Vergleich mit bisherigem Modell

Vergleicht man die im Abschnitt zuvor per Differenzenquotient ermittelte Geschwindigkeit mit der Geschwindigkeit aus dem alten Simulationsmodell (vgl. Abbildung 5.10), so erkennt man, dass sich die „reale“ Geschwindigkeit nach dem anfänglichen Einpendeln stets etwas unterhalb der simulierten Geschwindigkeit befindet, der Wagen somit etwas langsamer als im bisherigen Modell angenommen fährt. Es lässt sich aber zumindest feststellen, dass die gemessene Geschwindigkeit ab 150 ms linear ansteigt und dann somit dem erwarteten Verhalten für eine konstante Beschleunigung, die bei „Step1“ vorliegt, erfüllt. Man erkennt zudem, dass die gefundene Gerade eine etwas geringere Steigung als das bisherige Modell besitzt. Man kann hieraus schließen, dass die in diesem Fahrprofil geforderte Beschleunigung von 3 m/s^2 nicht ganz erreicht wird.

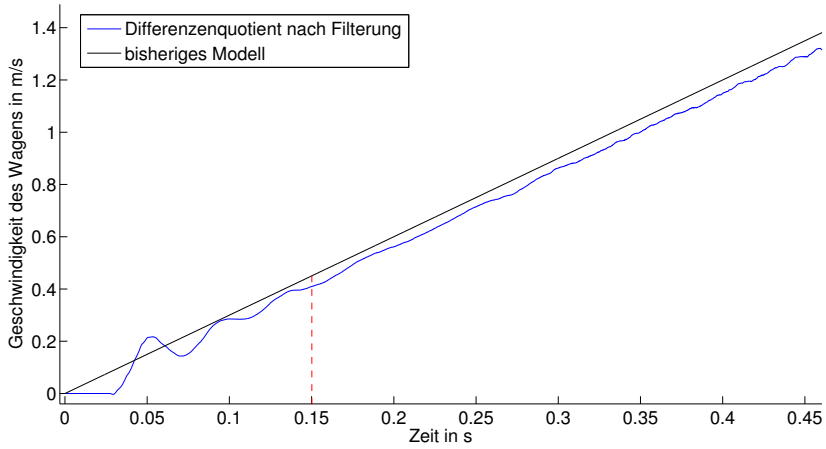


Abbildung 5.10: Vergleich der ermittelten Geschwindigkeit mit der Geschwindigkeit aus dem alten Simulationsmodell

5.4 Beschleunigung des Schlittens

Nach der numerischen Ermittlung der Geschwindigkeit des Schlittens liegt unser Augenmerk nun auf der Bestimmung der Beschleunigung, die als spätere Stellgröße für uns besonders interessant ist.

5.4.1 Ermittlung durch Differenzenquotient

Nachdem die numerische Differentiation der Positionsdaten mittels Differenzenquotient nach starkem Filtern bei der ersten Ableitung noch funktioniert hat, stößt sie bei der zweiten Ableitung an ihre (numerischen) Grenzen. Der Differenzenquotient für die zweite Ableitung

$$\frac{\Delta^2 x}{\Delta t^2} = \frac{x(t+\Delta t) + x(t-\Delta t) - 2x(t)}{\Delta t^2} = \ddot{x}(t) + \mathcal{O}(\Delta t^2). \tag{5.18}$$

zeigt nur noch ein sehr grobes Gezacke, das abhängig von der verwendeten Stärke des Filters stark differiert. Auch nur geringe Veränderungen des Moving-Average- oder des Savitzky-Golay-Filters haben beträchtlichen Einfluss auf das Ergebnis. Aus diesem Grund lässt sich über den Wahrheitsgehalt dieser Resultate wenig aussagen. Gemein haben jedoch alle Ergebnisse, dass die Beschleunigung in den ersten 100-150 ms besonders stark schwankt und dies im Anschluss nachlässt, siehe Abbildung 5.11.

Um zu aussagekräftigen Ergebnissen zu kommen, ist es bei verrauschten Messdaten oftmals ein probates Mittel, die Daten etwas auszudünnen (sogenanntes „Downsampling“). Das Ergebnis, das entsteht, wenn man lediglich jeden fünften Messwert benutzt, zeigt Abbildung 5.12. Da wir eine Schrittweite von etwa 0.13 ms haben, hat man auch nach Durchführung des Downsamplings immer noch ausreichend viele Stützstellen. Man sieht nun zwar das Erreichen des Zielwerts 3 m/s^2, jedoch ist das starke Überschwingen zu Beginn sehr abgeschwächt, was wiederum auch nicht zufriedenstellend ist.

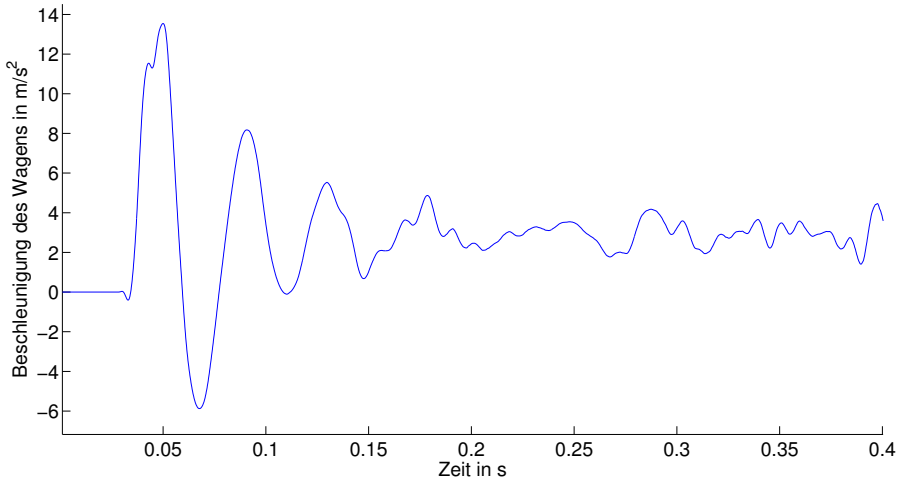


Abbildung 5.11: Ermittelte Beschleunigung mittels Differenzenquotient aus Messdaten

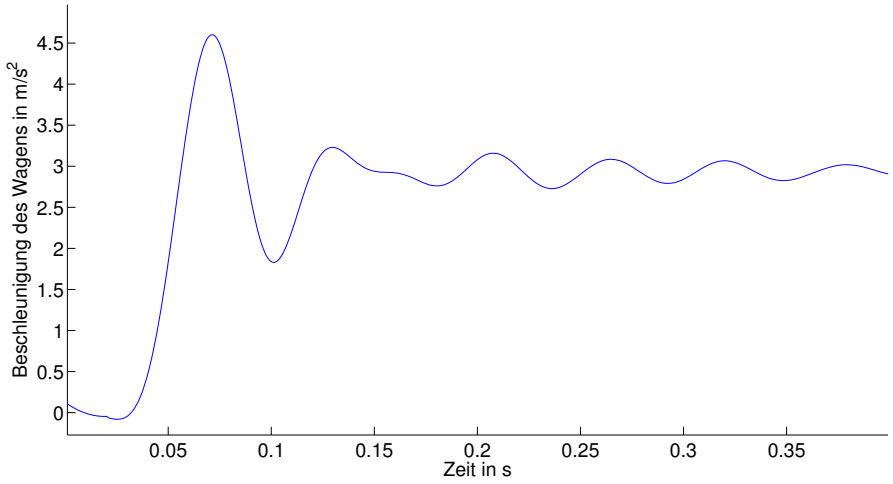


Abbildung 5.12: Ermittelte Beschleunigung mittels Differenzenquotient aus ausgedünnten Messdaten

Man kann also an dieser Stelle festhalten, dass sich die Resultate, abhängig von der angewandten Methodik (d.h. Filterart, Filterlänge und Downsampling), sehr stark unterscheiden und außer einem kräftigen Überschwingen zu Beginn kein gemeinsames Muster zu erkennen ist.

5.4.2 Ermittlung durch Kurvenanpassung

An dieser Stelle stellt sich die Frage, wie stark das Überschwingen der Beschleunigung zu Beginn des Losfahrens tatsächlich ist. Aus diesem Grund möchten wir nun mittels Kurvenanpassung die Beschleunigung bestimmen. Das Anpassen von Kurven an Daten stellt eine der

wichtigsten Methoden zur Bestimmung funktionaler Zusammenhänge verschiedener Größen dar. Nach Glymour ist die Kurvenanpassung ein zweistufiges Verfahren. Die erste Stufe, die Bestimmung des wahren Kurventyps, ist zumeist von Intuitionen gelenkt, wohingegen die zweite Stufe, die Berechnung der Bestapproximation innerhalb der Kurven des zuvor gewählten Typs, ein wohlverstandener Schritt ist. Dieser zweite Schritt basiert zumeist auf der Kleinste-Quadrate-Methode. Die erste Stufe hängt hingegen von zwei konkurrierenden Ansprüchen ab: Einfachheit versus Genauigkeit. Einerseits soll eine Kurve die Daten hinreichend gut approximieren, andererseits dabei aber möglichst einfach sein. Es stellt sich bei Verwendung der Kurvenanpassung daher stets die Frage, was ein geeignetes Trade-Off-Kriterium dieser beiden Ansprüche ist. [25]

Da die Größen Beschleunigung, Geschwindigkeit und Position des Wagens polynomial zusammenhängen, liegt es in unserem Fall nahe, Polynome als Kurventyp für die Anpassung zu verwenden. Aufgrund des betrachteten enormen Überschwingens werden wir jedoch auch Polynome verschiedenen Grades verwenden, um diese an unsere Positionsmesswerte anzupassen. Zur Bestimmung der Beschleunigung braucht das Polynom anschließend lediglich zweimal algebraisch abgeleitet werden. Wir haben allgemein ein Polynom vom Grad k der Form

$$f(t) = a_1\, t^k + a_2\, t^{k-1} + ... + a_{k-1}\, t^2 + a_k\, t + a_{k+1} \quad \text{mit } a_i \in \mathbb{R}, k \in \mathbb{N}, \tag{5.19}$$

das wir an eine bestimmte Anzahl N von Messpunkten anpassen möchten. Dies geschieht mit dem MATLAB-Befehl

```
poly_koeff = polyfit(t_temp,x_temp,k),
```

der an die Messdaten in `t_temp` und `x_temp` ein Polynom vom Grad k anpasst. In `poly_koeff` stehen anschließend die Koeffizienten des gefitteten Polynoms. Mit

```
poly_eval = polyval(poly_koeff,t_temp)
```

wird das gefundene Polynom dann an den Zeiten in `t_temp` ausgewertet. Wir werden hier natürlich nicht alle Messdaten auf einmal betrachten, womit man nur ein Polynom erhalten würde. Stattdessen betrachten wir stets ein „Fenster“ von N Messwerten, erhalten daraus ein Polynom, das wir auswerten können, schieben das Fenster anschließend um einen Zeitschritt (oder auch mehrere) nach rechts und beginnen von vorne.

Die zweite Ableitung $f''(t)$ von (5.19) lässt sich somit sehr leicht bestimmen und lautet

$$f''(t) = a_1 k(k-1)t^{k-2} + a_2(k-1)(k-2)t^{k-3} + ... + 2a_{k-1}. \tag{5.20}$$

Die beiden „großen Unbekannten“, die es nun zu bestimmen bzw. durch visuelles Optimieren zu ermitteln gilt, sind die Fenstergröße N, d.h. die Anzahl an Datenpunkten, an die das Polynom angepasst werden soll, sowie dessen Grad k. Auf das vorherige Filtern der Messdaten wird hier verzichtet, um mit unverfälschten Daten zu arbeiten.

Beschleunigung in den ersten Millisekunden

Zunächst wird sich der Beschleunigung des Wagens zum Zeitpunkt des Losfahrens gewidmet, in Abbildung 5.13 als t_1 bezeichnet. Es wird erneut das Fahrprofil „Step1“ betrachtet. Wir

fitten nun ein Polynom zweiten Grades durch die Messpunkte zwischen t_1 und t_2, d.h. durch die erste „Rampe“der Positionskurve.

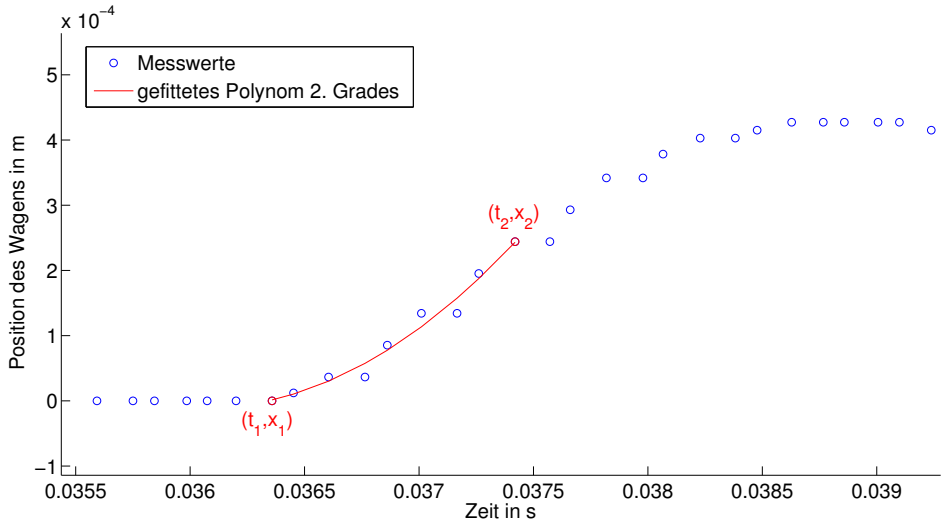


Abbildung 5.13: Gefittetes Polynom 2. Grades durch Anfangsbewegung des Fahrprofils „Step1“

Man erhält als Lösung

$$p(t) = 136.77440\,t^2 - 9.86310\,t + 0.17780. \tag{5.21}$$

Die zweite Ableitung ist konstant und beträgt für das gefundene Polynom 273.55. Das bedeutet, dass in diesem Abschnitt auch eine Beschleunigung von 273.55 $\mathrm{m/s^2}$ vorhanden ist.

Wendet man auf (t_1, x_1) und (t_2, x_2) den einfachen Differenzenquotienten an, so kommt man zunächst auf eine Geschwindigkeit von

$$v_2 = \frac{x_2 - x_1}{t_2 - t_1} = \frac{2.4419 \cdot 10^{-4}\ \mathrm{m} - 0\ \mathrm{m}}{0.03741967\ \mathrm{s} - 0.03635809\ \mathrm{s}} = 0.23002635\,\frac{\mathrm{m}}{\mathrm{s}}. \tag{5.22}$$

Da der Wagen vorher die Geschwindigkeit $v_1 = 0$ hatte, beträgt seine Geschwindigkeit am Ende des Intervalls den in (5.22) berechneten Wert. Daraus errechnet sich eine durchnittliche Beschleunigung von

$$a = \frac{v_2 - v_1}{t_2 - t_1} = \frac{0.23002635\,\frac{\mathrm{m}}{\mathrm{s}} - 0.0\,\frac{\mathrm{m}}{\mathrm{s}}}{0.03741967\ \mathrm{s} - 0.036358096\ \mathrm{s}} = 216.684241\,\frac{\mathrm{m}}{\mathrm{s}^2}. \tag{5.23}$$

Da diese lineare Approximation die Messdaten nicht so exakt annähert wie das vorher gefundene Polynom, ist das Ergebnis entsprechend geringer. Man beachte jedoch stets die Größenordnung, in der man sich hier bewegt (10^{-4} m) und dementsprechend auch die Empfindlichkeit der Messwerte: Ändert man einen Messpunkt um nur $\frac{1}{20}$ mm $= 5 \cdot 10^{-5}$ m ab, so ändert sich die Beschleunigung bereits um 44.367 $\mathrm{m/s^2}$! Betrachtet man die anderen Messungen des Fahrprofils „Step1“, so treten bereits hier große Unterschiede auf. Es lässt sich

jedoch festhalten, dass die Steigung der ersten „Rampe“, d.h. die erste Beschleunigung, stets über 100 m/s^2 liegt.

Würde man also die Messdaten absolut realistisch nachbilden wollen, so bräuchte man nach den 36 ms Latenz zu Beginn ein starkes Aufschwingen auf etwa 100 bis knapp 300 m/s^2, obwohl als Zielwert bei „Step1“ nur 3 m/s^2 vorgegeben waren. Dieser Fakt ist mehr als verwundernd, doch auch die anderen Fahrprofile bestätigen dieses Verhalten.

Ermittlung der richtigen Fitting-Parameter

Nun sollen auch die Beschleunigungen im weiteren Verlauf mittels Kurvenanpassung approximiert werden. Es stellt sich die Frage, welchen Grad man für das Polynom wählt und wieviel Datenpunkte man für das Polynomfitten betrachtet.

Es lässt sich schnell herausstellen, dass das Fitten mit nur wenigen Datenpunkten nicht erfolgreich ist. Beispielhaft ist in Abbildung 5.14 das Ergebnis der Kurvenanpassung mit jeweils zehn Datenpunkten zu sehen.

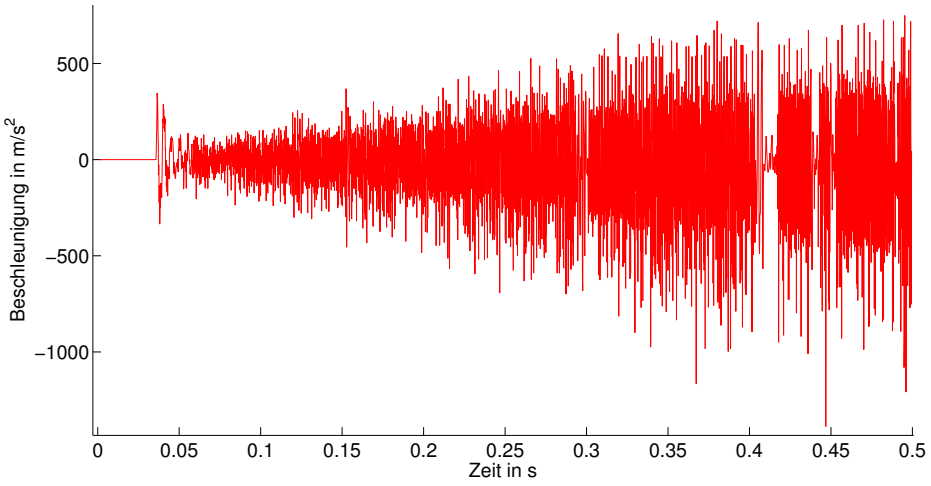


Abbildung 5.14: Fitting mit einem Polynom dritten Grades und zehn Datenpunkten

Auch das Ausdünnen der Messdaten hilft nicht weiter. Hierzu sieht man ein beispielhaftes Ergebnis in Abbildung 5.15, in der nur jeder dritte Messpunkt verwendet wurde. Man erkennt zwar, dass das Überschwingen zu Beginn etwa dem oben festgestellten entspricht und auch nach kurzer Zeit abnimmt, dem weiteren Verlauf nach jedoch wieder stark zunimmt. Dies kann man auch hier wieder mit der zunehmenden Ungenauigkeit (d.h. größer werdendem Messfehler) bei zunehmender Geschwindigkeit erklären.

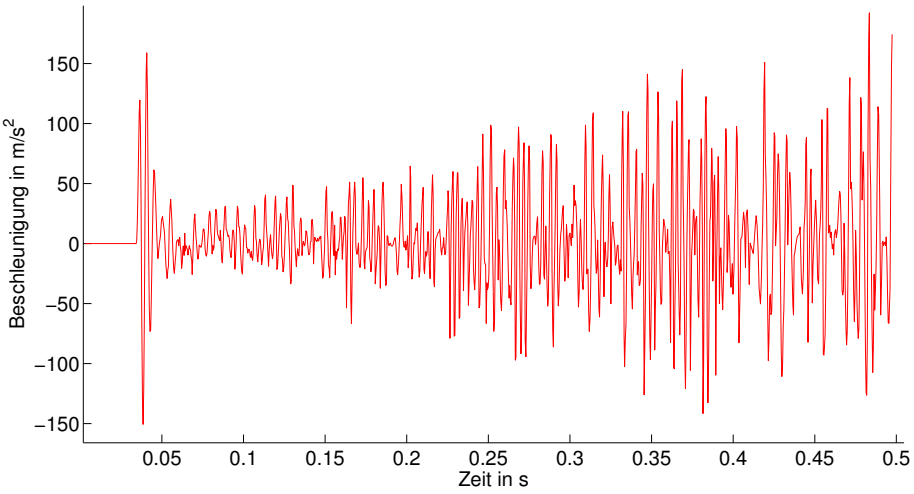


Abbildung 5.15: Fitting mit einem Polynom dritten Grades und zehn Datenpunkten nachdem die Messdaten ausgedünnt wurden (nur jeder 30. Wert)

Auch das Fitten mit der in MATLAB integrierten *Curve Fitting Toolbox* bringt keine Erfolge. Hierbei ist das Problem, dass MATLAB durch den gesamten Input-Vektor ein einziges Polynom legt. Die zweite Ableitung zeigt anschließend ein sofortiges Einfinden der Beschleunigung bei 3 m/s^2 ohne vorheriges Überschwingen, entspricht also ebenso wenig der Realität.

Ermittlung der tatsächlichen Beschleunigung mittels „Spline-Interpolation“

Man kommt schließlich zum Ziel, wenn man die Messdaten in relativ große, unterschiedlich lange Intervalle, d.h. ca. 6-10 ms pro Intervall, aufteilt und auf diesem Intervall mit einem Polynom mit hohem Grad interpoliert. Die Intervalle wurden dabei überlappend gewählt, damit die Polynome auch im Randbereich stabil bleiben und nicht nach oben oder unten „abhauen“. Aus Einfachheitsgründen wurden hier keine Randbedingungen – wie sonst in der Splineinterpolation üblich – definiert. Aber auch so ist das Prinzip und die Funktionsweise gut zu erkennen. Würde man einen niedrigeren Polynomgrad wählen, so wären für ein originalgetreues (messdatentreues) Fitting kleinere Zeitintervalle nötig. Das hätte jedoch wiederum zur Folge, dass Messfehler bzw. Störungen stärker zum Tragen kommen. Zudem wurde die „Verzögerungszeit“ von etwa 34 ms (vgl. Rechnung (5.1)), in der der Wagen im Nullpunkt verharrt, abgeschnitten.

Die zweiten Ableitungen der Polynome aus den einzelnen Intervallen (in Abbildung 5.16 rot und blau gefärbt) werden anschließend zu einer Funktion zusammengesetzt (in grün gestrichelt). Die auf diese Weise erhaltene Beschleunigung wollen wir nun zweimal integrieren, um deren Richtigkeit zu überprüfen. Sollte die gefundene Beschleunigung korrekt sein, so müsste die zweimalige Integration also in etwa dem Bild aus der Messung entsprechen. Da das alte Modell ja genau diesen idealen physikalischen Zusammenhang der zweimaligen Integration zwischen Beschleunigung und Position repräsentiert hat, können wir die gefundene Beschleunigung zur Überprüfung einfach in das bisherige Modell einsetzen und das System damit simulieren. Der Vergleich mit den Messdaten in Abbildung 5.17 zeigt, wie gut

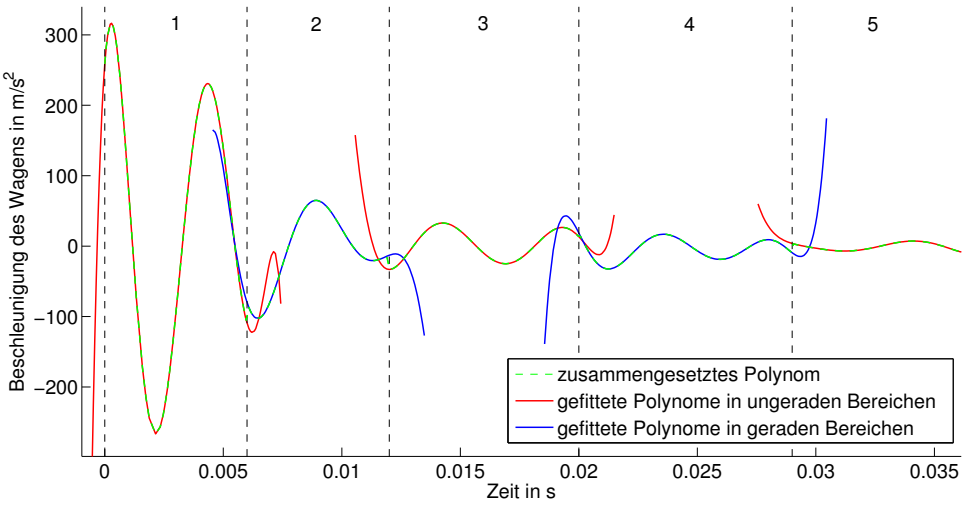


Abbildung 5.16: Erhaltene Beschleunigung durch stückweises Polynomfitting

die auf diese Weise ermittelte Beschleunigung der in dem Versuch tatsächlich aufgetretenen Beschleunigung entspricht: Sogar die kleinen „Rucker“ zu Beginn wurden exakt nachgebildet. Man beachte: Es handelt sich hier um die Bewegung des Wagens im Millimeterbereich innerhalb eines Zeitfenster von nur 35 ms.

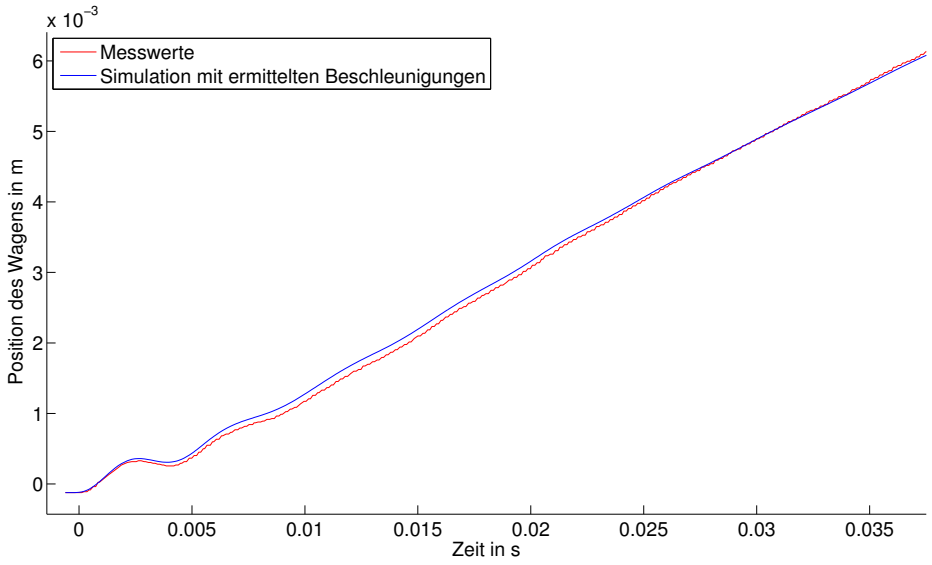


Abbildung 5.17: Vergleich der Messdaten mit der Simulation der erhaltenen Beschleunigung

Bei dieser Auswertung wurde ein Polynom neunten Grades verwendet und darüber hinaus folgende Intervallgrenzen für das Polynomfitting gewählt: 0.000 ms, 0.006 ms, 0.012 ms, 0.020 ms, 0.029 ms, 0.039 ms, 0.048 ms. Die einzelnen Bereiche haben sich jeweils um 3 ms überschnitten. Für andere Fahrprofile sind natürlich andere Intervalle zu wählen, um eine gute

Approximation zu erhalten. Weitere Stellvariablen sind die Überlappungszeit sowie der Polynomgrad. Selbstverständlich lässt sich auch in diesem Beispiel die Abweichung noch weiter minimieren, indem man eine Feinanpassung der Intervalle und des Überlappens durchführt und eventuell auch den Polynomgrad anpasst. Dies war hier jedoch nicht das Ziel, es sollte nur gezeigt werden, wie die reale Beschleunigung zu Beginn in etwa aussieht, was man in Abbildung 5.16 gut erkennen kann.

5.4.3 Gleichmäßige Beschleunigung

Bei einer Anfangsgeschwindigkeit v_0 und einer anschließenden gleichmäßigen Beschleunigung a lässt sich die Position x zur Zeit t bestimmen durch

$$x(t) = \frac{1}{2}at^2 + v_0 t. \tag{5.24}$$

Stellt man diese Gleichung nun nach der Beschleunigung um, so erhält man für jeden Zeitpunkt diejenige gleichmäßige Beschleunigung, die nötig ist, um die aktuelle Position zu erreichen:

$$a(t) = \frac{2(x(t) - v_0 t)}{t^2}. \tag{5.25}$$

Errechnet man mit dieser Formel die durchschnittlich benötigte gleichmäßige Beschleunigung für jeden Zeit-Positions-Messwert des Fahrprofils „Step1“, so erhält man Abbildung 5.18.

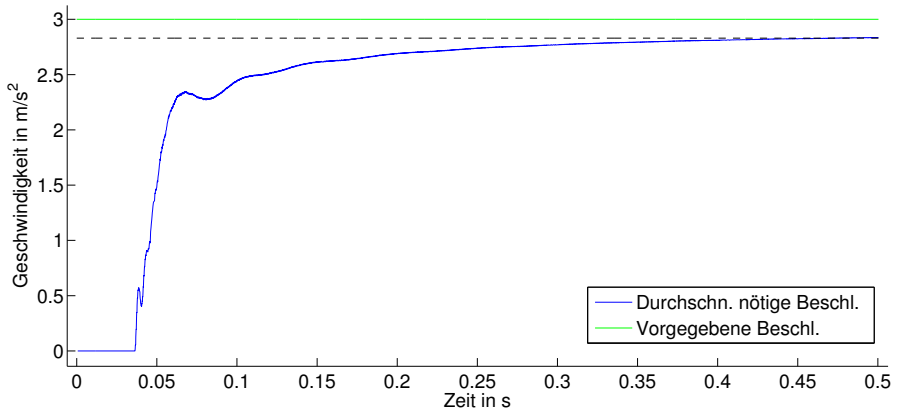


Abbildung 5.18: Durchschnittlich benötigte Beschleunigung für die Position des Wagens zur Zeit t beim Fahrprofil „Step1“

Durch das kräftige Überschwingen der Beschleunigung nach knapp 40 ms geht hier auch der Durchschnitt erwartungsgemäß sprunghaft nach oben. Nach etwa 210 ms hat man mit 2.7 m/s^2 eine durchschnittliche Beschleunigung von 90% des vorgegebenen Wertes erreicht, bevor letztendlich ein Wert von approximativ 2.83 m/s^2 (entspricht knapp 95% des vorgegebenen Wertes) erreicht wird. Ebenso stellt sich bei den beiden anderen „Step“-Fahrprofilen[4]

[4]Für die restlichen Fahrprofile lässt sich diese Berechnung nicht durchführen, da es sich stets um eine Durchschnittsbeschleunigung handelt und diese somit nur bei Anwendung einer einzigen Beschleunigung aussagekräftig ist.

zum Ende der Messung ein Wert von knapp 95% des vorgegebenen Wertes ein. Aufgrund der begrenzten Bahnlänge lässt sich hier jedoch nicht feststellen, ob dies tatsächlich der Sättigungswert ist oder ob dieser noch weiter steigen würde. Nichtsdestotrotz lässt sich zumindest erkennen, dass auch durch den Sprung der Beschleunigung zu Beginn der Rückstand nicht komplett wettgemacht werden kann und somit nicht die Position erreicht wird, die normalerweise mit dieser Beschleunigung erreicht hätte werden müssen.

5.5 Erkenntnisse

Aus den vorherigen Abschnitten dieses Kapitels lässt sich für die Bewegung des Schlittens zusammenfassend festhalten:

- Große Latenz zu Beginn, diese fällt jedoch zuweilen etwas unterschiedlich aus, siehe Abbildung 4.3.
- Anschließend extremer Sprung der Beschleunigung und Versuch des „Einholens“ der eigentlichen Fahrtrajektorie.
- Einpendeln der Beschleunigungs- und Geschwindigkeitskurve etwas unterhalb der ursprünglich vorgegebenen Werte.

Wenn der Motor aktiv ist und man eine Beschleunigung von 0 m/s^2 wirken lässt, so kann man feststellen, dass der Wagen absolut festzusitzen scheint und auch per Hand nicht zu verschieben ist. Der integrierte Regler ist also schnell genug und der Motor stark genug, um sich von diesen äußeren Bedingungen nicht beeinflussen zu lassen bzw. ihnen entgegenzuwirken. Da der Regler aber zumindest eine Regeldifferenz wahrnehmen muss, bevor er einschreitet, hat man zumindest ein marginales „Spiel“ von mindestens einem Tick im Motor. Da dies aber minimal ist und deshalb keinen merklichen Einfluss auf das restliche Verhalten des Systems hat, kann man davon ausgehen, dass auch das Gewicht des Pendels keinen Einfluss auf die Position des Wagens nimmt bzw. nehmen kann. Das rechtfertigt im nächsten Kapitel weiterhin ein Motormodell zu betrachten, das unabhängig von der Pendelposition ist.

- Keine Abhängigkeit der Wagenposition von der Pendelposition und -geschwindigkeit.

Kapitel 6

Einführung neuer Modelle

Wie bereits zu Beginn zu sehen war und wie das letzte Kapitel gezeigt hat, kann das bisherige Simulationsmodell die Realität nicht ausreichend exakt darstellen. Mit Hilfe der erworbenen Erkenntnisse werden im folgenden Kapitel zunächst neue Modelle für die Beschleunigung des Wagens eingeführt. Nachdem diese parametrisiert und verifiziert wurden, wird abschließend ein kurzer Vergleich zeigen, inwieweit es Verbesserungen zur vorherigen Situation gibt. Jegliche regelungstechnische Aspekte in diesem Kapitel stammen aus [33], [22] und [21].

6.1 Motor als PID-Glied

In Abschnitt 3.1 wurde bereits beschrieben, dass im Servomotor ein integrierter PID-Regler arbeitet, der die Steuerung des Wagens kontrolliert. In Kapitel 4 ist das Wirken dieses Reglers sehr deutlich ersichtlich, wenn sich der Motor nach der unterschiedlichen zeitlichen Verzögerung zu Beginn schließlich trotzdem bei der nahezu richtigen Beschleunigung bzw. Geschwindigkeit einpendelt. Aus diesem Grund ist es naheliegend den Motor als PID-Übertragungsglied zu modellieren. Sogenannte Übertragungsglieder sind Verknüpfungen, die auf eine Ursache (= Eingangsgröße) eine bestimmte Wirkung (= Ausgangsgröße) zeigen. Beim PID-Glied handelt es sich somit um ein sogenanntes Übertragungsglied mit Proportional-, Integral- und Differenzieranteil. Für eine Einführung in das Themengebiet der Regelungstechnik wird [33] und [21] empfohlen.

Bei Verwendung des PID-Glieds hätte man somit eine Kaskadenregelung, d.h. eine Ineinanderschachtelung mehrerer Regler. Zum einen kann der PID-Regler zur Regelung des Motors als innerer Regelkreis und zum anderen kann die Stabilisierung des ganzen Systems, d.h. des Pendels, als äußerer Regelkreis betrachtet werden. Dieser Aufbau wurde bereits in Abbildung 3.2 schematisch dargestellt. Die Reglerausgangsgröße des äußeren Reglers (auch Führungsregler genannt) dient dann als Führungsgröße für den inneren Regler (Folgeregler).

Die Funktionalbeziehung im Zeitbereich für den PID-Regler in Parallelstruktur lautet:

$$\begin{gathered} y(t) = K_p\, e(t) + K_i \int_{t_0}^{t} e(\tau) d\tau + K_d\, \dot{e}(t) \\ \text{mit } e(t) = u_{\text{Soll}}(t) - u_{\text{Ist}}(t). \end{gathered} \tag{6.1}$$

$e(t)$ ist die sogenannte Regelabweichung, d.h. die Differenz zwischen gewünschter Beschleunigung und aktueller Beschleunigung und somit Eingangsgröße des Reglers. u_{Soll} ist demnach die Sollgröße, die der äußere Regler (z.B. MPC) zur Stabilisierung des Pendels vorgibt. $y(t)$ ist die Stellgröße und somit Ausgangsgröße des Reglers.

Das P-Glied, d.h. der erste Summand in (6.1), sorgt dafür, dass die Regelabweichung mit dem Verstärkungsfaktor K_p multipliziert und unverzögert weitergegeben wird, was zu einer schnellen Regelung führt. Das Problem des alleinigen P-Reglers ist die bleibende Regelabweichung. Aus diesem Grund wird das I-Glied (siehe zweiter Summand in (6.1)) hinzugenommen: Es summiert die Regelabweichung über der Zeit auf und multipliziert das Integral mit dem Faktor K_i. Je länger also eine Regelabweichung besteht, desto größer wird die Stellgröße des I-Reglers. Das Stellglied wird demnach solange verändert, bis die Regelabweichung null ist. Ein integrierender Anteil verbessert im Regler zwar die stationäre Genauigkeit, zugleich wirkt er aber verzögernd. Zudem kann der Regelkreis durch die integrierende Wirkung instabil werden. Aus diesen Gründen (Trägheit und Schwingneigung) werden reine I-Regler normalerweise auch nicht eingesetzt. Zur Beschleunigung der Regelung wird der D-Regler eingeführt: Der differenzierende Anteil bewertet die Änderung der Regelabweichung und berechnet so deren Änderungsgeschwindigkeit, welche dann mit dem Faktor K_d multipliziert wird. Der D-Anteil wirkt somit beschleunigend und unterdrückt Schwingneigungen, führt aber auch wieder zu einer schlechteren stationären Genauigkeit.

In Abbildung 6.1 ist die typische Sprungantwort der verschiedenen Regler zu sehen, die sich aus den oben genannten einfachen Übertragungsgliedern zusammensetzen lassen.

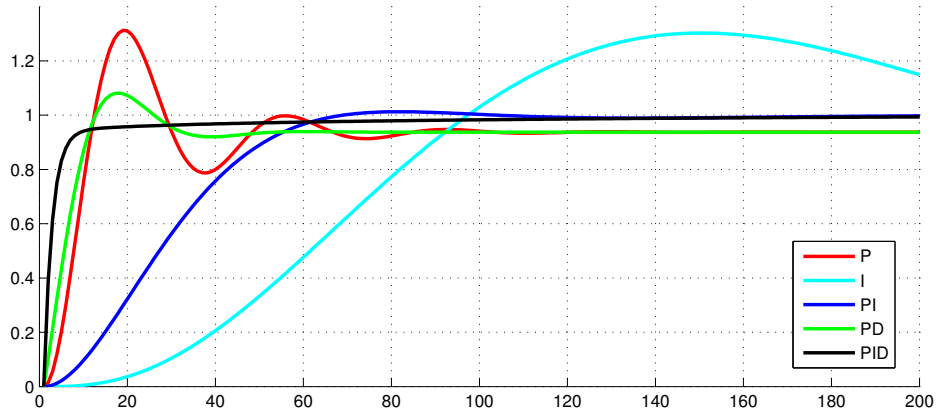


Abbildung 6.1: Typische Sprungantworten für verschiedene Regler

6.1.1 Herleitung des Simulationsmodell

Nun soll der erläuterte PID-Regler in das bisherige Simulationsmodell integriert werden. Dazu leitet man zunächst (6.1) nach der Zeit ab, wodurch man

$$\begin{gathered}\dot{y}(t) = K_p\,\dot{e}(t) + K_i\,e(t) + K_d\,\ddot{e}(t) \\ \text{mit } e(t) = u_{\text{Soll}}(t) - u_{\text{Ist}}(t).\end{gathered} \tag{6.2}$$

erhält. Setzt man jetzt den Ausgang $y(t) =: x_5(t)$, so lässt sich dies als Steuerung wie folgt an das bisherige Modell (2.1) ankoppeln:

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\,(g \sin\, x_1(t) + x_5(t) \cos\, x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= x_5(t) \\
\dot{x}_5(t) &= K_p\, \dot{e}(t) + K_i\, e(t) + K_d\, \ddot{e}(t) \\
e(t) &= u(t) - x_5(t)
\end{aligned} \tag{6.3}$$

Man beachte auch die Änderung von $u(t)$ zu $x_5(t)$ in der zweiten Zeile. Anschließend erhält man mit der Substitution $x_6(t) := e(t)$ und $x_7(t) := \dot{e}(t)$ die zusätzliche Differentialgleichung $\dot{x}_6(t) = x_7(t)$ und somit:

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\,(g \sin\, x_1(t) + x_5(t) \cos\, x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= x_5(t) \\
\dot{x}_5(t) &= K_p\, x_7(t) + K_i\, x_6(t) + K_d\, \dot{x}_7(t) \\
\dot{x}_6(t) &= x_7(t) \\
x_6(t) &= u(t) - x_5(t)
\end{aligned} \tag{6.4}$$

Umstellen liefert

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\,(g \sin\, x_1(t) + x_5(t) \cos\, x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= x_5(t) \\
\dot{x}_5(t) - K_d\, \dot{x}_7(t) &= K_p\, x_7(t) + K_i\, x_6(t) \\
\dot{x}_6(t) &= x_7(t) \\
0 &= u(t) - x_5(t) - x_6(t).
\end{aligned} \tag{6.5}$$

Wir bezeichnen die rechte Seite dieses Differentialgleichungssystems als $f(t, x, u)$ und stellen zusätzlich auf der linken Seite eine sogenannte Massenmatrix M auf, die hier folgendermaßen aussieht:

$$M = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & -K_d \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} \tag{6.6}$$

Man erhält also schließlich eine Differentialgleichung der Form

$$M\dot{x}(t) = f(t, x, u). \tag{6.7}$$

6.1.2 *Exkurs:* Massenmatrix und Differential-Algebraische Gleichung

Da das Gebiet der Differential-Algebraischen Gleichungen ein sehr breites, vielschichtiges und komplexes Anwendungsgebiet ist, ist es schwierig, es knapp zu beschreiben. Demnach wird sich hier auf die Erläuterung der Grundbegriffe beschränkt, die nötig sind, um das weitere Vorgehen in dieser Arbeit nachvollziehen zu können. Dieser Abschnitt richtet sich dabei nach [17], [1] und [19].

Wie wir gesehen haben, führt das neue Simulationsmodell aus (6.5) auf ein Differentialgleichungssystem der Form

$$M(t,x)\,\dot{x}(t) = f(t,x); \quad x(t_0) = x_0 \tag{6.8}$$

mit sogenannter Massenmatrix $M(t,x)$.

Ist die Massenmatrix invertierbar, so lässt sich die Differentialgleichung umstellen zu $\dot{x}(t) = M^{-1}f(t,x)$ und hat für jeden Anfangswert x_0 zur Zeit t_0 eine Lösung. Doch auch wenn eine Invertierung möglich wäre, ist diese nicht immer sinnvoll. So könnte die Matrix beispielsweise vorteilhafte Eigenschaften für die numerische Behandlung besitzen. Man denke dabei an dünn-besetzte Matrizen, bei denen die Inverse in der Regel nicht dünn-besetzt ist. Zum anderen kann (6.8) die naheliegende Darstellung des zugrundeliegenden Systems sein, das man durch Invertierung nicht „unkenntlich“ machen möchte, wie etwa bei großen elektrischen Netzwerken. [34]

Ist die Massenmatrix singulär, hat man ein System von differential-algebraischen Gleichungen (auch Algebro-Differentialgleichungen). Dabei handelt es sich also um ein System, in dem gewöhnliche Differentialgleichungen mit algebraischen (d.h. hier ableitungsfreien) Nebenbedingungen gekoppelt sind. Im Differentialgleichungssystem (6.5) ist diese Struktur sofort ersichtlich, da die letzte Zeile keine Ableitung enthält und somit rein algebraisch ist. Eine differential-algebraische Gleichung hat nur dann eine Lösung, wenn der Startwert x_0 konsistent ist, d.h. wenn das zugehörige Anfangswertproblem mindestens eine Lösung besitzt. Das bedeutet, es existiert eine Anfangssteigung x_{p_0}, sodass gilt $M(t_0,x_0)x_{p_0} = f(t_0,x_0)$.

Die allgemeinste Form einer differential-algebraischen Gleichung ist hingegen eine implizite Differentialgleichung der Form

$$F(t,x(t),\dot{x}(t)) = 0, \quad F : \mathbb{R}\times\mathbb{R}^n\times\mathbb{R}^n \to \mathbb{R}^m \tag{6.9}$$

für eine Funktion $x : I \to \mathbb{R}^m$ mit $I \subset \mathbb{R}$. Aus dem klassischen Satz über implizite Funktionen folgt, dass eine Gleichung in dieser impliziten Form (lokal) nach $\dot{x}$ auflösbar ist, wenn die partielle Ableitung $F_{\dot{x}}$ regulär ist. In diesem Fall kann man die implizite Gleichung (6.9) umschreiben in die Form $\dot{x}(t) = G(t,x(t))$ und hat damit wieder eine explizite gewöhnliche Differentialgleichung.

Eine echte differential-algebraische Gleichung liegt dann vor, wenn die partielle Ableitung $F_{\dot{x}}$ singulär ist. Man betrachtet somit eine Differentialgleichung auf einer Mannigfaltigkeit, diese ist jedoch bei praktischen Problemen meist zunächst nicht explizit bekannt.

Wird bei gewöhnlichen Differentialgleichungen die Lösung durch Integration bestimmt, so ergeben sich im Gegensatz dazu Teile der Lösung einer differential-algebraischen Gleichung

durch Differentiation. Aus diesem Grund reicht nicht wie bei gewöhnlichen Differentialgleichungen die Forderung, dass die Systemfunktion F nur stetig bzw. stetig differenzierbar sein muss, um die Lösbarkeit zu garantieren, sondern es werden nun auch höhere Ableitungen für die Lösung benötigt. Die genaue Ordnung der benötigten Ableitung wird allgemein als (Differentiations-)Index der differential-algebraischen Gleichung bezeichnet.

Der Differentiationsindex eines differential-algebraischen Gleichungssystems

$$F(t, x(t), \dot{x}(t)) = 0 \tag{6.10}$$

ist definiert als die Anzahl $m > 0$ an Zeitableitungen, die nötig sind, um aus dem entstehenden Gleichungssystem

$$\begin{aligned} F(t, x, \dot{x}) &= 0 \\ \frac{d}{dt}(F(t, x, \dot{x})) &= 0 \\ &\vdots \\ \frac{d^m}{dt^m}(F(t, x, \dot{x})) &= 0 \end{aligned} \tag{6.11}$$

durch algebraische Umformungen ein gewöhnliches Differentialgleichungssystem

$$\dot{x}(t) = G(t, x(t)) \tag{6.12}$$

extrahieren zu können. Dieser Index bestimmt auch die in Frage kommenden numerischen Löser für das Problem.

Numerisches Lösen in MATLAB

Mehrere MATLAB-Solver können Problemstellungen mit Massenmatrix lösen, wobei es allerdings Unterschiede darin gibt, welche Eigenschaften von der Massenmatrix gefordert werden. Hat man ein Problem mit Massenmatrix M vorliegen, muss dies dem Integrator mit Hilfe der Mass-Option mitgeteilt werden, indem diese dem Solver mit

```
options = odeset('Mass',M)
```

übergeben wird.

Für die Integration der Systeme in semi-expliziter Form (6.8) in MATLAB gibt es folgendes zu beachten:

- ode23s erlaubt nur konstante Massenmatrizen
- alle Integratoren außer ode15s und ode23t erlauben nur reguläre, d.h. invertierbare Massenmatrizen
- ode15s und ode23t erlauben zwar nicht invertierbare Massenmatrizen, diese dürfen dann aber nicht von der Form $M(t, x)$ sein, also nicht gleichzeitig zeit- und zustandsabhängig sein und zudem keinen Differentiationsindex größer 1 haben.

Die zuletzt genannte Tatsache schränkt die Verwendung dieser stark ein. Auch im vorliegenden Fall können wir die oben genannten Solver nicht verwenden, denn diese geben bei einem Lösungsversuch direkt den Hinweis aus, dass es sich um eine Gleichung mit höheren Index handelt und diese nicht gelöst werden können.

In Bezug auf den numerischen Aufwand sind explizite Differentialgleichungsysteme einfacher zu lösen als solche mit Massenmatrizen. Bei letzteren wiederum erfordern Systeme mit einer diagonalen Massenmatrix weniger Aufwand als solche mit vollbesetzter Massenmatrix. [34]

Wie oben beschrieben, hängt die Lösbarkeit der differential-algebraischen Gleichung von der Wahl der Startwerte ab. Sind diese nicht konsistent, behandelt der Solver die Startwerte jedoch als Schätzungen und versucht konsistente Werte in der Nähe der Schätzungen zu finden, bevor er anschließend das Problem löst. [32]

Numerisches Lösen mittels RADAU5

Da wir mit den in MATLAB integrierten Lösern in unserem Fall nicht weiterkommen, bedienen wir uns mittels MEX-Interface einem externen Löser. Es kommt hierbei RADAU5 von E. Hairer und G. Wanner zum Einsatz, der auch in [17] beschrieben wird. Dieser benutzt ein implizites Runge-Kutta-Verfahren der Ordnung 5 mit Schrittweitensteuerung und kann differential-algebraische Gleichungen mit Differentiationsindex kleiner gleich 3 lösen. Nach Anbindung des impliziten Lösers wie in Abschnitt 3.4 beschrieben, kann dieser durch den Aufruf

```
[tGitter,xGitter,stats,taupred]=radau5Mex(f,t,x0,opt)
```

benutzt werden. Die Massenmatrix M wird dabei mit den Optionen übergeben, indem man zuvor `opt.Mass = M` setzt. Wie auch bei den in MATLAB integrierten Lösern ist hier die Wahl der Anfangswerte entscheidend. Die Solver sind nur erfolgreich, wenn die gewählten Startwerte in `x0` hinreichend nah an konsistenten Anfangswerten liegen.

6.1.3 Parameterschätzung der Modellparameter und Überprüfung

Um die optimalen Parameter K_p, K_i und K_d für das Modell zu finden, wird eine Minimalwertsuche mit der MATLAB-Funktion `fminsearch` durchgeführt. Diese Funktion wendet den Simplex-Algorithmus an, um ein Minimum zu finden und wird mit

```
res = fminsearch(F, awe, options)
```

aufgerufen. Dabei werden in `res` die drei berechneten Parameter K_p, K_i und K_d zurückgeliefert. `F` ist die zu minimierende Funktion, die in unserem Fall die Least-Squares-Abweichung der Simulation von (6.5) mit den aktuell verwendeten Parametern und den Messwerten berechnet:

$$\min_{K_p,K_i,K_d} F(K_p,K_i,K_d) = \min_{K_p,K_i,K_d} \sum_j (x(K_p,K_i,K_d)_j^{\text{Simulation}} - x_j^{\text{Messung}})^2 \tag{6.13}$$

`awe` enthält die Startwerte unserer Parameter sowie `options` einstellbare Optionen wie Genauigkeit oder Maximalzahl an Iterationen. Beim verwendeten Simplex-Verfahren ist es sehr wichtig, sinnvolle Startwerte vorzugeben, da der Algorithmus auch nach Auffinden eines lokalen Minimums abbricht. In `F` kommt das MATLAB-Interface von RADAU5 zum Einsatz. Da man hierbei keine weiteren Parameter an die Differentialgleichung übergeben kann, werden die Parameter global definiert. An dieser Stelle sei nochmal explizit erwähnt, dass wir hier nur die Messwerte des Schlittens mit den Simulationswerten für die Position vergleichen und anhand dieser optimieren. Die Messwerte des Pendels werden hier noch nicht betrachtet.

Als Referenzprofil wird dabei der kompliziertere Fahrversuch „Upswing“ verwendet, um möglichst viele unterschiedliche Beschleunigungen auf das System zu geben und somit möglichst viele Informationen herausziehen zu können. Denn je mehr in der Simulation passiert, d.h. je mehr unterschiedliche Situationen generiert werden, desto genauer sind die Parameter, die man erhält. Da `fminsearch` sehr langsam ist und man noch keine Erfahrung über konsistente Anfangswerte hat, starten wir die Optimierung jedoch zunächst mit dem Fahrprofil „Step1“, um akzeptable Startwerte zu finden.

Das Finden passender Startwerte für den Optimierer stellt sich als erste Schwierigkeit heraus. Beginnt man mit beliebigen Werten, so verfängt sich der Optimierer oftmals in lokalen Minima, bei denen die gefittete Kurve nicht ansatzweise in der Nähe der Versuchswerte liegt. Doch auch wenn das visuelle Überprüfen der Positionsdaten ein erfolgreiches Optimieren vermuten lässt, muss dies nicht unbedingt der Fall sein. So haben sich für Startwerte nahe Null beispielsweise folgende Lösungswerte als lokales Minimum ergeben:

$$K_p = -0.953044582, \quad K_i = 117.5989591, \quad K_d = 0.006545204. \tag{6.14}$$

Allein der negative K_p-Beiwert lässt einen stutzig werden und erscheint mehr als fraglich. Betrachtet man die Plots für diese Lösung, so sieht der Positionsvergleich wie erwähnt auf den ersten Blick nicht schlecht aus, siehe Abbildung 6.2, links. Betrachtet man jedoch auch die simulierte Beschleunigung (siehe rechte Grafik von Abbildung 6.2), so sieht man ein starkes Hin- und Herschwingen der Beschleunigung. Mag dies zwar bis zu einem gewissen Grad erwünscht sein, so klingt dieses Verhalten bei dieser Beschleunigung jedoch viel zu langsam ab. Darüber hinaus kann ein negativer Proportional-Beiwert nicht der Realität entsprechen, da diese per Definition positiv sind.

Aus diesem Grund werden weitere Startwerte getestet, um weitere lokale Minima auszumachen. Den besten Wert der Zielfunktion liefern schließlich folgende Werte:

$$K_p = 15.92552188, \quad K_i = 88.00566313, \quad K_d = 0.000708770. \tag{6.15}$$

Wie in Abbildung 6.3 zu erkennen ist, sieht diese Lösung auf den ersten Blick sehr gut aus und der Vergleich der Fehlerquadratsumme zeigt, dass es auch besser als das bisherige Modell ist. Vergleicht man aber die Positionswerte zu Beginn des Fahrprofils, insbesondere den Unterschied nach den ersten 100 ms, so erkennt man deutliche Differenzen. Dies liegt daran, dass dieser PID-Regler sehr schnell auf ca. 90% des vorgegebenen Wertes springt und sich diesem im Anschluss noch weiter annähert (vgl. rechte Grafik in Abbildung 6.4). Aber gerade durch diesen Sprung zu Beginn ist das Verhalten in diesem Bereich relativ ähnlich zum bisherigen Modell. Dieses Verhalten hat sich bei allen gefundenen lokalen Minima bestätigt. Demzufolge entsprechen insbesondere die Fehlerquadratsummen der kürzeren Fahrprofile

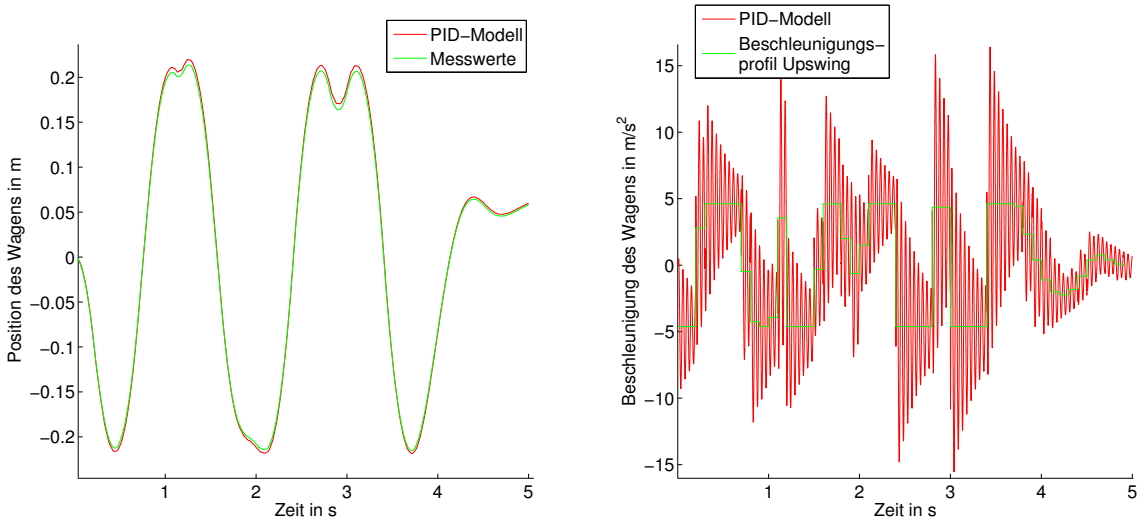


Abbildung 6.2: Simulation des Fahrprofils „Upswing“ mit dem PID-Modell und den Parametern aus (6.14)

nicht unseren Erwartungen, obwohl es auch hier Verbesserungen im Vergleich zu vorher gibt.

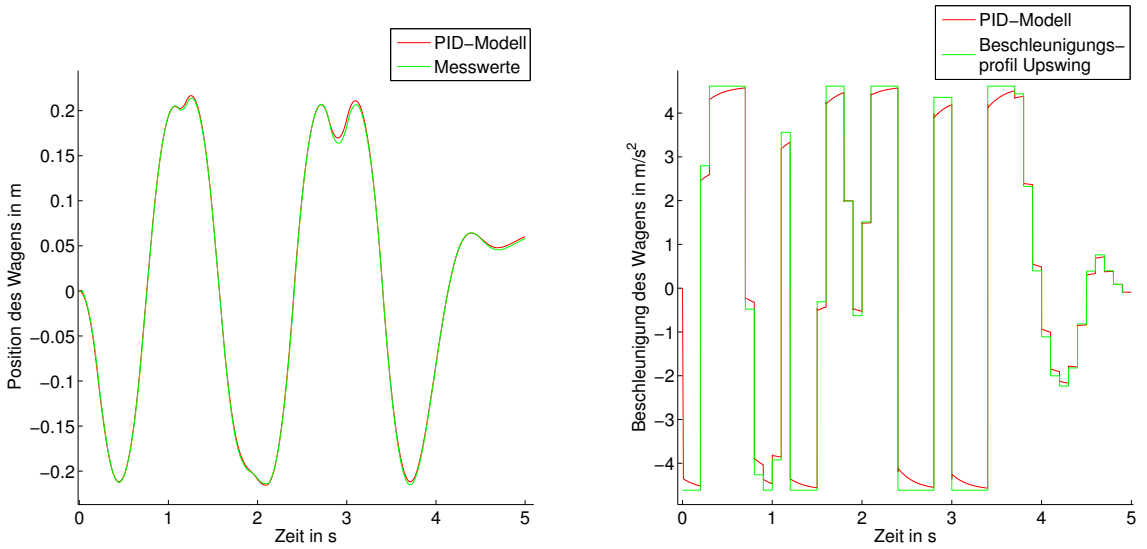


Abbildung 6.3: Simulation des Fahrprofils „Upswing“ mit dem PID-Modell und den Parametern aus (6.15)

Ursprünglich sollte zur Optimierung die MATLAB-Funktion `lsqnonlin` verwendet werden, die Verwendung mit RADAU5 als ODE-Solver hat sich jedoch als problematisch herausgestellt: Um exakte Ergebnisse erzielen zu können, muss der ODE-Löser wesentlich (d.h. 2-3 Zehnerpotenzen) genauer arbeiten als der Differenzierer im Minimierer. Im MEX-Interface

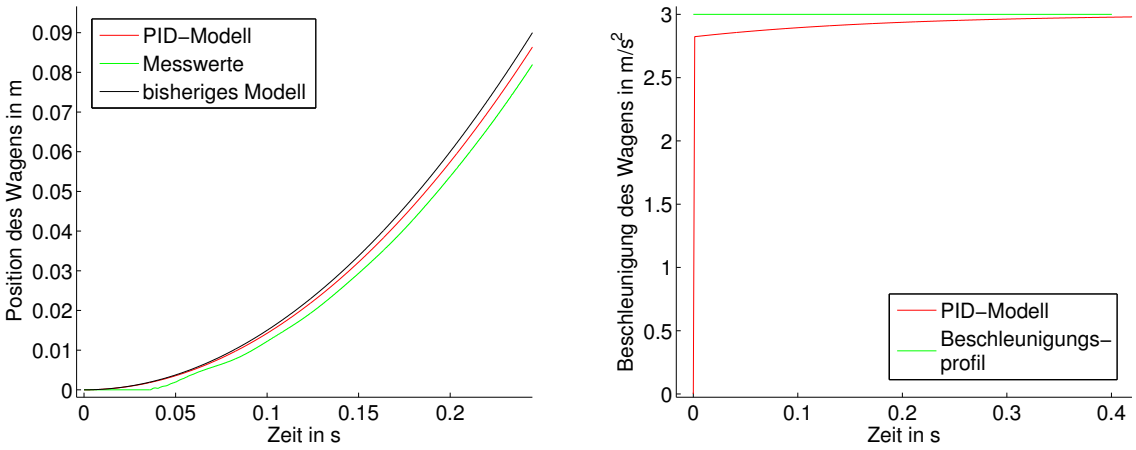


Abbildung 6.4: Simulation des Fahrprofils „Step1“ mit dem PID-Modell und den Parametern aus (6.15)

für RADAU5 gibt es jedoch keine Möglichkeit, die minimale Schrittweite von 10^{-6} zu verändern, weshalb `lsqnonlin` nur mit geringer Genauigkeit verwendbar wäre.

Auch wenn das Vorgehen hier reine Heuristik ist, legen die Ergebnisse der Parameterschätzung nahe, dass dieses Modell nicht das Richtige sein kann. Allem Anschein nach ist es nicht möglich, die Reaktionen des PID-Reglers im Servomotor exakt und in allen Einzelheiten nachbilden zu können. Dafür besteht das System womöglich aus zu vielen einzelnen Aspekten, hier seien nur kurz Haft- und Gleitreibung, evtl. auch Hysterese (durch das mechanische „Spiel“ der Umlenkzahnräder am Band bzw. im Motor), die Latenz und der übermäßig große Sprung zu Beginn genannt. Aus diesem Grund wollen wir nun eine Approximation des Ganzen betrachten und zu einem neuen Modell überleiten.

6.2 Motor als PT_2-Glied

Wie im vorherigen Abschnitt zu sehen war, ist es nicht möglich, die große Verzögerung zu Beginn und den enormen Sprung bzw. das extreme Überschwingen danach mit dem PID-Übertragungsglied zu modellieren. Aus diesem Grund wollen wir nun zu einer Approximation des Verhaltens übergehen. Es soll demnach ein Modell gefunden werden, das verzögert wirkt und überschwingt, jedoch nicht ganz so extrem wie in den Auswertungen beobachtet.

Aus diesem Grund wird im Folgenden das sogenannte PT_2-Übertragungsglied betrachtet. Dieses besitzt ein proportionales Übertragungsverhalten mit Verzögerung 2. Ordnung. Gebräuchliche Beispiele sind in der Elektrotechnik der RLC-Schwingkreis und im Maschinenbau die Feder-Masse-Dämpfer-Anordnung. [8]

Die zugehörige Funktionalbeziehung im Zeitbereich ist die Differentialgleichung

$$T^2\,\ddot{x}_a(t) + 2\,d\,T\dot{x}_a(t) + x_a(t) = K_s\,x_e(t), \tag{6.16}$$

wobei $x_e(t)$ die Eingangs- und $x_a(t)$ die Ausgangsgröße beschreibt. $d > 0$ bezeichnet den sogenannten Dämpfungsgrad, $K_s > 0$ die Übertragungskonstante bzw. den Verstärkungsfaktor und $T > 0$ die Zeitkonstante.

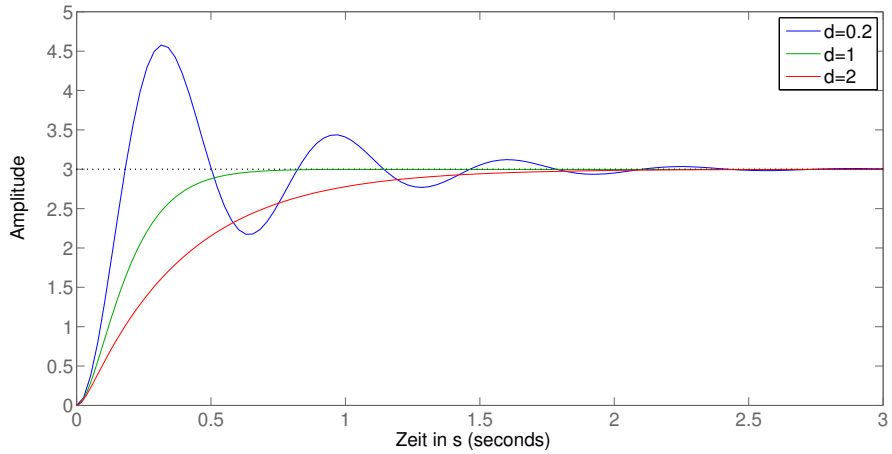


Abbildung 6.5: Sprungantwort eines PT_2-Glieds ($K_s = 3, T = 0.1, d = 0.2/1.0/2.0$)

Wie bei der in Abbildung 6.5 gezeigten Sprungantwort des PT_2-Glieds zu erkennen ist, ist für das Überschwing-Verhalten der Dämpfungsgrad d entscheidend. Im sogenannten periodischen Fall, d.h. $0 < d < 1$, erhält man das gewünschte Überschwingen mit anschließendem Einpendeln. Ein weiterer Aspekt, der für die Verwendung des PT_2-Glieds spricht, ist die Tatsache, dass die erste Ableitung anfangs identisch Null ist. Durch dieses Verhalten lässt sich die Verzögerung des Schlittens zu Beginn modellieren.

Die Auswertungen in Abschnitt 5.4.3 haben gezeigt, dass der Wagen die vorgegebene Beschleunigung (und damit auch die vorgegebene Geschwindigkeit und Position) nicht vollständig erreicht. Dieser Aspekt kann hier durch den Verstärkungsfaktor K_s geregelt werden und bekräftigt die Rechtfertigung ein PT_2-Glied zur Modellierung des Motors zu verwenden.

6.2.1 Herleitung des Simulationsmodell

Die Eingangsgröße des PT_2-Glieds $x_e(t)$ ist in unserem Zusammenhang die vorgegebene Steuerung $u(t)$. Den Ausgang $x_a(t)$ bezeichnen wir mit $x_5(t)$, d.h. $x_5(t)$ stellt nun die „verzögerte“ Beschleunigung dar und dient somit als rechte Seite von $\dot{x}_4(t)$ als Eingang in unser bestehendes Differentialgleichungssystem. Mit der Substitution $x_6(t) := \dot{x}_5(t) := \dot{x}_a(t)$ erhält man also

$$T^2\,\dot{x}_6(t) + 2\,d\,T\,x_6(t) + x_5(t) = K\,u(t). \tag{6.17}$$

Stellt man dies nun nach $\dot{x}_6(t)$ um und koppelt es anschließend an das bisherige Modell an, so ergibt sich insgesamt

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\,(g \sin x_1(t) + x_5(t) \cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= x_5(t) \\
\dot{x}_5(t) &= x_6(t) \\
\dot{x}_6(t) &= \frac{1}{T^2}\,(K\, u(t) - 2\, d\, T x_6(t) - x_5(t)).
\end{aligned} \tag{6.18}$$

In der zweiten Zeile wird wiederum $u(t)$ durch $x_5(t)$ ersetzt. Im Vergleich zum vorherigen PID-Modell ist das aufgrund der nicht vorhandenen Massenmatrix[1] ein ungleich einfacheres und leichter zu lösendes Differentialgleichungssystem.

6.2.2 Parameterschätzung der Modellparameter

Zunächst werden die Parameter des PT_2-Glieds, d.h. des Wagenmodells, geschätzt. Ist dieser Schritt durchgeführt, betrachten wir die Pendelgleichungen unseres Modells, d.h. die ersten beiden Zeilen aus (6.18). Da sich das Wagenmodell verändert hat, ist davon auszugehen, dass auch die Parameter der Pendelgleichungen anzupassen sind, da letzere direkt von der Beschleunigung des Wagens ($= x_5(t)$) beeinflusst werden. Nach der Schätzung der Wagenparameter werden wir also ebenso die Pendelparameter optimieren.

Schätzung der Wagenparameter

Zur Schätzung der im Modell auftretenden unbekannten Parameter T, d und K werden erneut Optimierungsprobleme betrachtet, die auf ein Least-Squares-Funktional zur Minimierung des Abstands zwischen Positionsdaten und Modellantwort führen. Da wir nicht mehr auf den impliziten Löser RADAU5 angewiesen sind, können wir die Differentialgleichungen nun mit `ode23` lösen und die Minimierung mit dem Kleinste-Quadrate-Optimierer `lsqnonlin` durchführen. Dieser arbeitet mit Hilfe des sogenannten Trust-Region-Verfahrens. Die Fehlerquadratsumme wird dabei vom Algorithmus selbst gebildet. Es reicht also in der zu minimierenden Zielfunktion F lediglich den Differenzvektor aus Simulationswerten und Messwerten des Fahrprofils „Upswing“ zu berechnen und zurückzugeben. Die Differentialgleichung wird hierbei mit einer relativen und absoluten Genauigkeit von 10^{-10} gelöst. Der Optimierer wiederum verwendet anschließend eine Genauigkeit von 10^{-8} für die gesuchten Parameter als auch für die Zielfunktion. Auch in diesem Fall müssen zunächst passende Startwerte gefunden werden, die erneut durch visuelle Überprüfung ausgewählt werden. Im vorliegenden Fall liefert der Algorithmus für die Startwerte $K_s = 0.98$, $T = 0.010$ und $d = 0.5$ die folgende Lösung:

$$K_s = 0.975587973, \quad T = 0.010837977, \quad d = 0.330599232. \tag{6.19}$$

[1] bzw. genau genommen einer Einheitsmatrix als Massenmatrix

Die visuelle Überprüfung der geschätzten Parameter erfolgt im Anschluss, der numerische Vergleich zum alten Simulationsmodells zum Abschluss des Kapitels in Abschnitt 6.4.

Verifikation des Wagenmodells

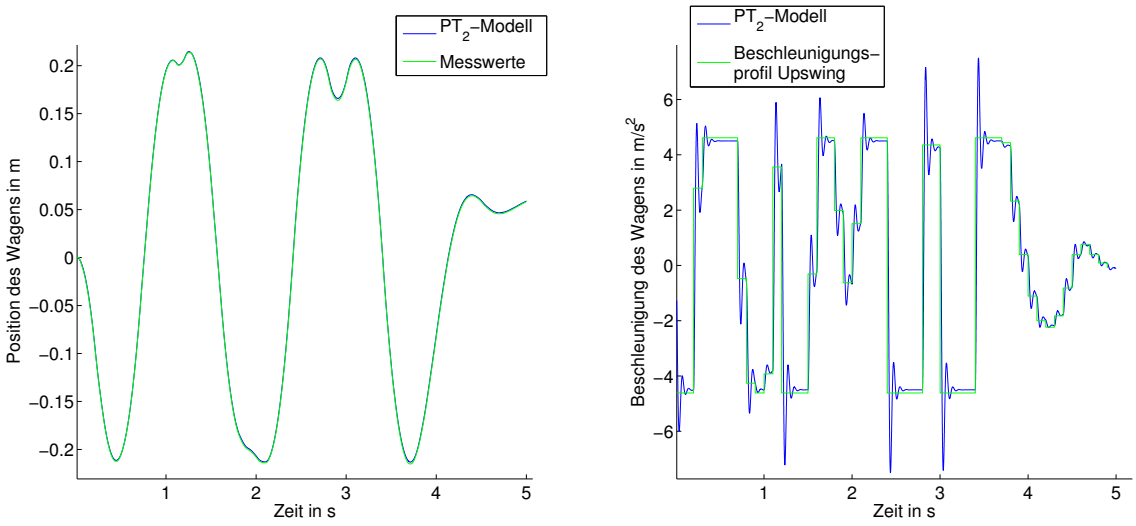


Abbildung 6.6: Simulation des Wagens mit dem PT_2-Modell und den Parametern aus (6.19) beim Fahrprofil „Upswing“

Die Simulation des Fahrprofils „Upswing“ mit dem PT_2-Modell und den Parametern aus (6.19) ist in Abbildung 6.6 dargestellt. Während im linken Teil aufgrund der sehr guten Simulationswerte und der groben Skalierung keine merklichen Unterschiede zwischen Simulation und Messung erkennbar sind, sieht man in rechten Teil deutlich das (erwünschte) Überschwingen der Beschleunigung bei jedem Beschleunigungswechsel und das anschließende schnelle Einpendeln. Zur Unterstreichung der Güte der Simulation ist in Abbildung 6.7 nochmals ein vergrößerter Ausschnitt derselben Simulation zu sehen, nun zusätzlich auch mit dem direkten Vergleich zum alten Simulationsmodell. Man kann erkennen, dass auch nach einer Steuerungszeit von drei Sekunden die Simulation der Realität sehr gut entspricht.

Auch der Vergleich mit dem Fahrprofil „Step1“ ist mehr als zufriedenstellend, siehe Abbildung 6.8. Die Verzögerung zu Beginn kann durch das PT_2-Modell gut aufgefangen werden. Auch im weiteren Verlauf ist optisch lediglich ein marginaler Unterschied zwischen den Messwerten und dem Simulationsmodell auszumachen.

Da nicht nur die korrekte Position des Schlittens von hoher Bedeutung für die richtige Angabe des Pendelwinkels ist, sondern auch dessen Geschwindigkeit, soll diese nun auch untersucht

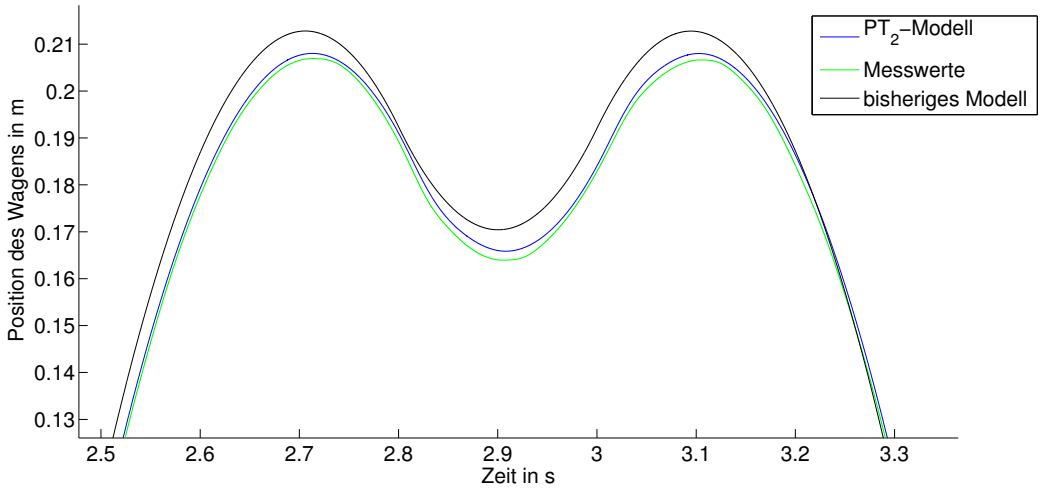


Abbildung 6.7: Simulation des Fahrprofils „Upswing“ mit dem PT_2-Modell und den Parametern aus (6.19)

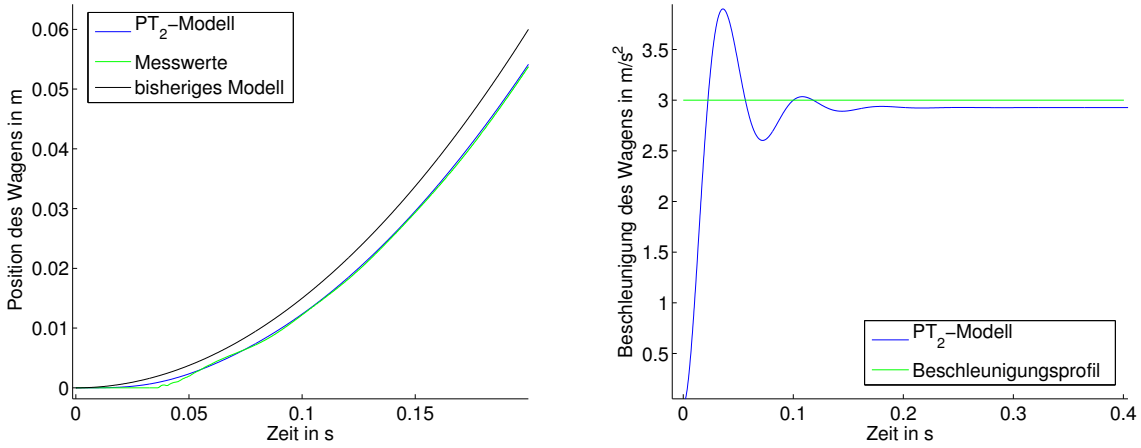


Abbildung 6.8: Simulation des Fahrprofils „Step1“ mit dem PT_2-Modell und den Parametern aus (6.19)

werden. Der Vergleich der Geschwindigkeit des Schlittens mit der in Abschnitt 5.3 via Polynomfitting ermittelten Geschwindigkeit zeigt große Übereinstimmung, wie Abbildung 6.9 belegt. Der Sprung zu Beginn kann durch das PT_2-Modell natürlich nicht exakt nachgebildet werden, aber man erkennt, dass sich die Geschwindigkeit des Wagens im Simulationsmodell auf der fast identischen Gerade wie die approximierte Gerade einfindet. Ebenso ist auch die minimal geringere Steigung im Vergleich zum alten Modell übereinstimmend.

Ebenso zeigt der Vergleich der Geschwindigkeiten des Fahrprofils „Upswing“ erhebliche Verbesserungen im Vergleich zum bisherigen Modell. So wird die approximierte Geschwindigkeit

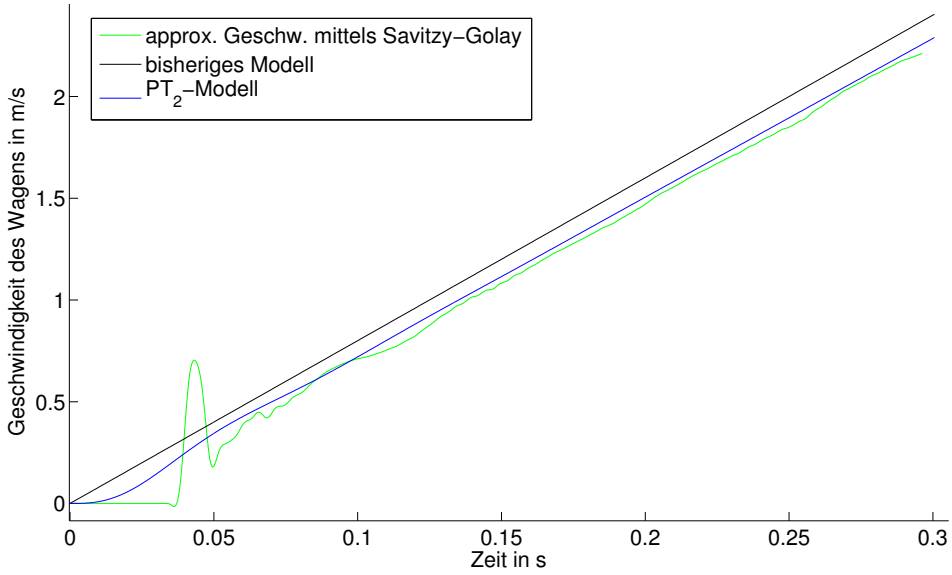


Abbildung 6.9: Simulation des Fahrprofils „Step3“ mit dem PT_2-Modell und den Parametern aus (6.19)

sowohl bei konstanten Beschleunigungsstücken als auch bei Beschleunigungswechseln sehr genau getroffen. Hierzu sei auf den Ausschnitt, der in Abbildung 6.10 zu sehen ist, verwiesen. Man beachte hierbei, dass es sich nach wie vor um eine Vorwärtssimulation handelt und wiederum der Bereich nach drei Sekunden Fahrzeit betrachtet wird.

Schätzung der Pendelparameter

Betrachtet man das neu eingeführte Modell (6.18), so sieht man, dass auch die Pendelgleichung in Zeile 2 in Form von $x_5(t)$ direkt vom neuen Motormodell beeinflusst wird. Dementsprechend werden in diesem Abschnitt die Parameter angepasst und neu geschätzt.

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\,(g \sin x_1(t) + x_5(t) \cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= x_5(t) \\
\dot{x}_5(t) &= x_6(t) \\
\dot{x}_6(t) &= \frac{1}{T^2}\,(K_s\, u(t) - 2\, d\, T\, x_6(t) - x_5(t)).
\end{aligned} \tag{6.20}$$

Zur Schätzung der auftretenden Parameter K_1 und K_2 werden erneut Optimierungsprobleme betrachtet, die auf ein Least-Squares-Funktional führen, diesmal jedoch zur Minimierung des Abstands zwischen gemessenen Winkelwerten des Pendels und der entsprechenden Modellantwort. Auch hier kann erneut der Kleinste-Quadrate-Optimierer `lsqnonlin` zur

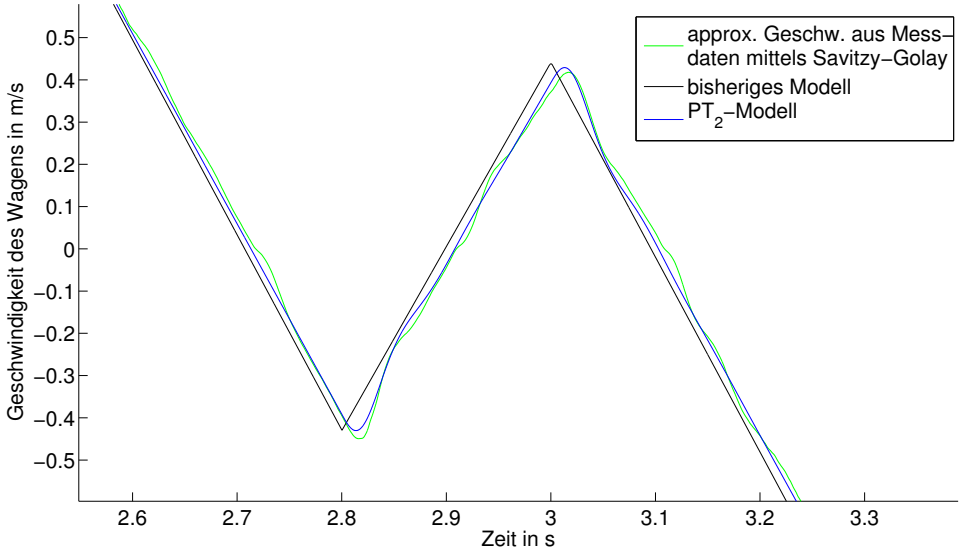


Abbildung 6.10: Simulation des Fahrprofils „Upswing“ mit dem PT_2-Modell und den Parametern aus (6.19)

Optimierung eingesetzt werden. Die zu minimierende Zielfunktion F berechnet nun den Differenzvektor aus Simulationswerten und Winkelmessungen des Fahrprofils „Upswing“ und gibt diesen zurück. Auch hier ist „Upswing“ wie zuvor letztendlich das beste Referenzprofil, da es am meisten Informationen enthält. Die Differentialgleichung wird hierbei erneut mit einer relativen und absoluten Genauigkeit von 10^{-10} gelöst. Der Optimierer wiederum verwendet anschließend eine Genauigkeit von 10^{-8} für die gesuchten Parameter als auch für die Zielfunktion. Als Startwerte wurden die Parameter aus dem alten Modell gewählt: $K_1 = 0.029787$ und $K_2 = 1.466434$. Im vorliegenden Fall liefert der Algorithmus für diese Anfangswerte die folgende Lösung:

$$K_1 = 0.01616628, \quad K_2 = 1.46086441. \tag{6.21}$$

Während sich die zweite Konstante prozentual gesehen wenig ändert, so ist in der ersten Konstanten für die Reibung schon eine beträchtliche Änderung zu erkennen.

Die hier geschätzten Parameter werden nachfolgend visuell überprüft, der numerische Vergleich zum alten Simulationsmodells erfolgt am Ende des Kapitels in Abschnitt 6.4.

Verifikation des Pendelmodells

Abbildung 6.11 und Abbildung 6.12 zeigen die Simulation des Pendelwinkels bei den Fahrprofilen „Upswing“ bzw. „Step1“ im Vergleich zu den aufgenommenen Messwerten und dem alten Simulationsmodell. Das neue Modell ist hierbei so gut, dass die Messwerte in grün davon so gut wie nicht zu unterscheiden sind. Beim „Upswing“-Profil lässt sich eine maximale Differenz von 0.0268 rad feststellen. Dies entspricht einem Winkel von etwa 1.5°. Bezogen auf die lange Fahrzeit von fünf Sekunden ist dies ein absolut zufriedenstellender Wert. Das

alte Simulationsmodell hatte bei diesem Fahrversuch sogar einen Überschlag des Pendels berechnet.

Beim Fahrversuch „Step1“ beträgt der maximale Unterschied zwischen neuem Modell und Messwerten sogar nur 0.0031 rad $\widehat{=}$ 0.18° und liegt damit nur knapp über der möglichen Auflösungsgrenze des Pendelsensors[2] von 0.0261 rad.

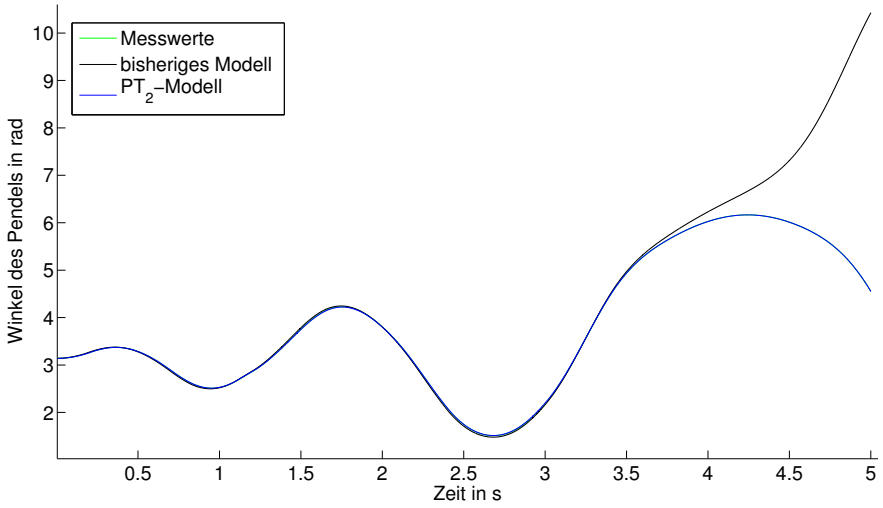


Abbildung 6.11: Vergleich der Messwerte mit den Simulationswerten des Winkels beim Fahrprofil „Upswing“

6.2.3 Das PT_2-Modell

Zum Abschluss dieses Abschnitts wird das nun gefundene Simulationsmodell mit einem PT_2-Glied als Motormodell nochmals vollständig niedergeschrieben:

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\, (g \sin x_1(t) + x_5(t) \cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= x_5(t) \\
\dot{x}_5(t) &= x_6(t) \\
\dot{x}_6(t) &= \frac{1}{T^2}\, (K_s\, u(t) - 2\, d\, T\, x_6(t) - x_5(t)).
\end{aligned} \tag{6.22}$$

[2] In Abbildung 6.16 sind die Abstufungen der Winkelmesswerte deutlich zu erkennen.

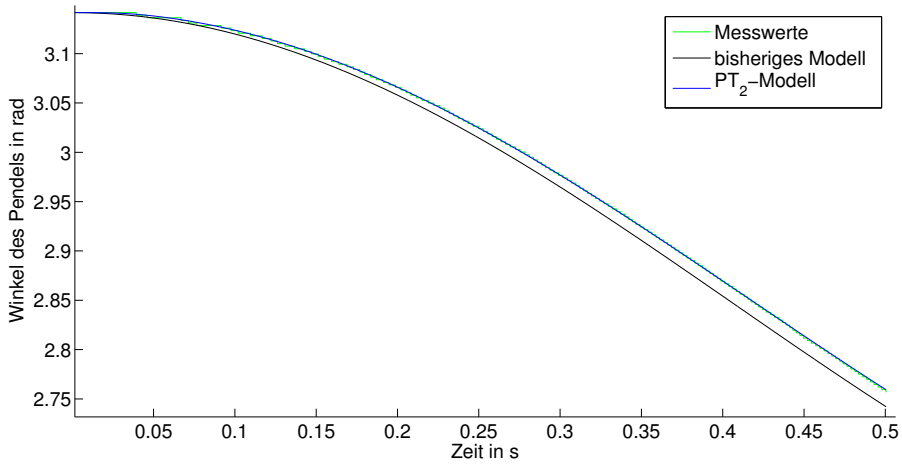


Abbildung 6.12: Vergleich der Messwerte mit den Simulationswerten des Winkels beim Fahrprofil „Step1“

mit den Konstanten

$$\begin{aligned} K_1 &= 0.01616628 \\ K_2 &= 1.46086441 \\ g &= 9.81 \\ K_s &= 0.975587973 \\ d &= 0.330599232 \\ T &= 0.010837977. \end{aligned} \tag{6.23}$$

6.3 Motor als PT_1-Glied

Wie im letzten Abschnitt zu sehen war, entspricht das eingeführte PT_2-Modell sehr gut der Realität. Betrachtet man aber die extremen Spitzen, d.h. das extreme Überschwingen der Beschleunigung bei Beschleunigungswechseln in der rechten Grafik von Abbildung 6.6, so kann dies in der Praxis numerisch problematisch werden. Dieses steife Differentialgleichungssystem setzt einen sehr genauen Löser voraus, was wiederum mit einer langen Rechenzeit einhergeht. Gerade bei Echtzeit-Regelalgorithmen hat man hierfür unter Umständen nicht die nötigen Ressourcen.

Aus diesem Grund soll nun noch ein weiteres Modell eingeführt werden, das annähernd genau, aber wesentlich leichter zu lösen sein wird. Es handelt sich hierbei um das sogenannte Verzögerungsglied 1. Ordnung oder kurz PT_1-Glied. Dies ist ein Übertragungsglied, bei dem die Ausgangsgröße nach einer sprunghaften Änderung der Eingangsgröße exponentiell mit einer bestimmten Anfangssteigung asymptotisch gegen einen Endwert strebt. Es besitzt also insbesondere kein Überschwingen, das uns später beim Lösen der Differentialgleichung Probleme bereiten könnte. Allgemein sieht die Differentialgleichung eines PT_1-Gliedes fol-

gendermaßen aus:

$$x_a(t) + T\,\dot{x}_a(t) = K\,x_e(t), \tag{6.24}$$

wobei $x_e(t)$ die Eingangs- und $x_a(t)$ die Ausgangsgröße beschreibt. $T > 0$ ist wiederum die Zeitkonstante und $K > 0$ der Verstärkungsfaktor.

6.3.1 Herleitung des Simulationsmodell

Die Eingangsgröße des PT_1-Glieds $x_e(t)$ ist analog zum PT_2-Modell die vorgegebene Steuerung $u(t)$. Den Ausgang dieses Übertragungsglieds $x_a(t)$ bezeichnen wir erneut mit $x_5(t)$, dieser ist somit wiederum die „verzögerte“ Beschleunigung. Nach Einsetzen in (6.24) und anschließendem Umstellen erhält man

$$\dot{x}_5(t) = \frac{1}{T}\,(K\,u(t) - x_5(t)). \tag{6.25}$$

In Analogie zu obigem Vorgehen können wir dies nun erneut an das bisherige Simulationsmodell anhängen.

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\,x_2(t) + K_2\,(g\sin x_1(t) + x_5(t)\cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= x_5(t) \\
\dot{x}_5(t) &= \frac{1}{T}\,(K\,u(t) - x_5(t))
\end{aligned} \tag{6.26}$$

6.3.2 Parameterschätzung der Modellparameter

Analog zum Vorgehen beim PT_2-Modell werden zunächst die Parameter des PT_1-Glieds, d.h. des Wagenmodells, geschätzt, bevor anschließend die Parameter der Pendelgleichungen optimiert werden.

Schätzung der Wagenparameter

Die Schätzung der im Modell auftretenden unbekannten Parameter K und T wird erneut mittels Kleinste-Quadrate-Optimierung in `lsqnonlin` durchgeführt. Es werden dabei die gleichen Fehlertoleranzen wie vorher verwendet. Die Positionswerte der Simulation werden wiederum mit den Messungen des Fahrprofils „Upswing“ verglichen. Als Startwerte für die Optimierung werden $T = 0.010$ und $K = 0.99$ gewählt. Der Algorithmus liefert anschließend die folgenden Lösungswerte:

$$T = 0.00719139, \quad K = 0.97745639. \tag{6.27}$$

Die visuelle Überprüfung der geschätzten Parameter erfolgt wiederum direkt im Anschluss, der numerische Vergleich zum alten Simulationsmodells danach zum Abschluss des Kapitels in Abschnitt 6.4.

Verifikation des Wagenmodells

Die Simulation des Fahrprofils „Upswing“ mit dem PT_1-Modell und den Parametern aus (6.27) ist in Abbildung 6.13 zu sehen. In der linken Grafik sind auch nach einer Steuerungszeit von fünf Sekunden keine merklichen Unterschiede zwischen Simulation und Messung erkennbar. Im rechten Teil der Abbildung sieht man deutlich, dass nun kein Überschwingen der Beschleunigungen mehr vorhanden ist.

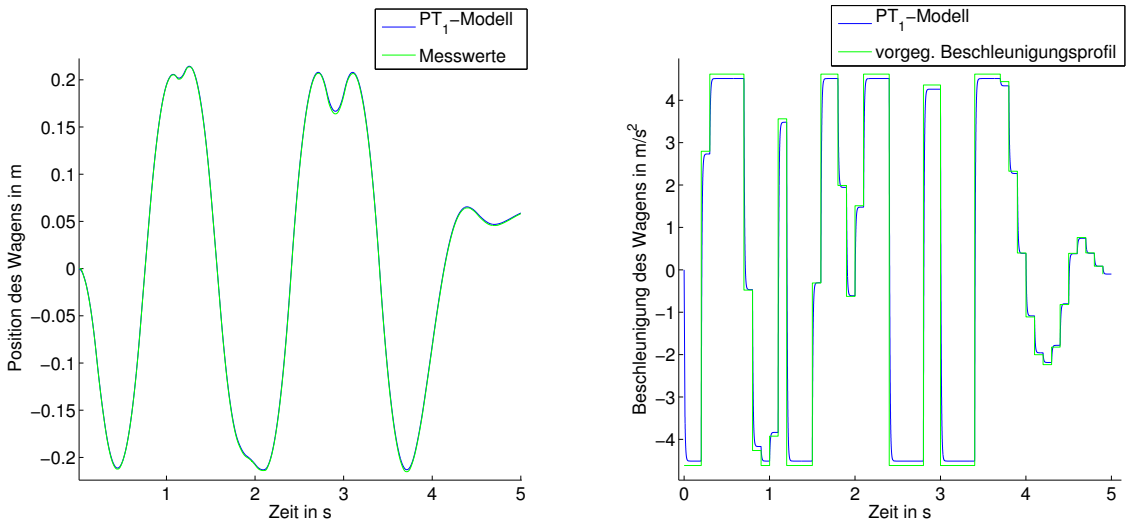


Abbildung 6.13: Simulation des Wagens mit dem PT_1-Modell und den Parametern aus (6.27) beim Fahrprofil „Upswing“

Auch der Vergleich mit dem Fahrprofil „Step2“ ist sehr zufriedenstellend, siehe Abbildung 6.14. Auf den ersten Blick sieht das PT_1-Modell ähnlich gut wie das PT_2-Modell aus. Es ist jedoch insbesondere ein Unterschied im Anfangsbereich festzustellen. Auf diese Differenzen wird in Abschnitt 6.4 nochmals explizit eingegangen. Die Verzögerung zu Beginn kann durch das PT_1-Modell nicht ganz so gut aufgefangen werden wie beim PT_2-Modell, dennoch ist im weiteren Verlauf optisch lediglich ein marginaler Unterschied zwischen den Messwerten und dem Simulationsmodell auszumachen. Im Gegensatz hierzu ist die Abweichung des alten Modells jedoch deutlich sichtbar.

Auch der in Abbildung 6.15 gezeigte Ausschnitt des Geschwindigkeitsvergleichs des Wagens zeigt deutlich, wie gut das PT_1-Modell ist. Auch hier sind nur im Anfangsbereich Unterschiede zum PT_2-Modell festzustellen. Die exakte Fehlerbetrachtung und der Vergleich dieser folgt ebenso in Abschnitt 6.4.

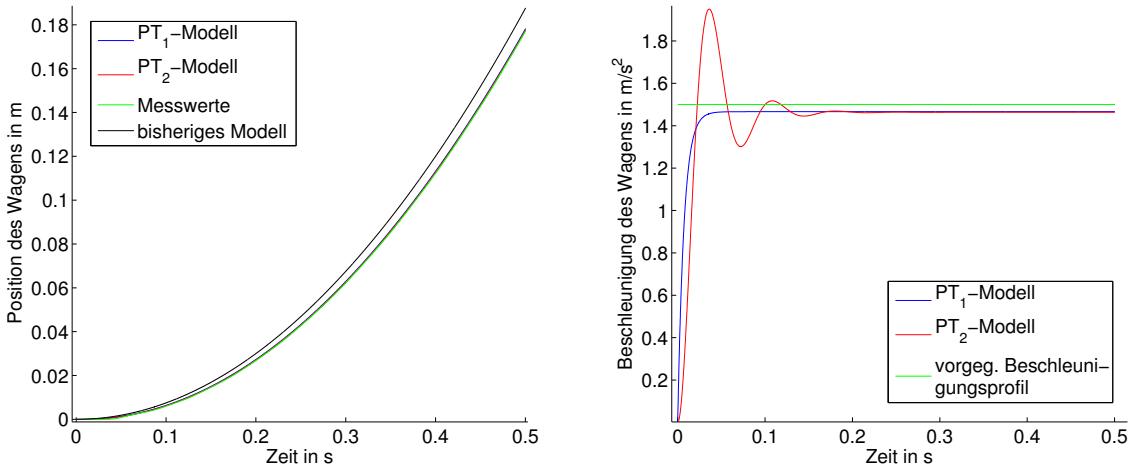


Abbildung 6.14: Simulation des Wagens mit dem PT_1-Modell und den Parametern aus (6.27) beim Fahrprofil „Step2“ im Vergleich zu den Messwerten und den anderen Modellen

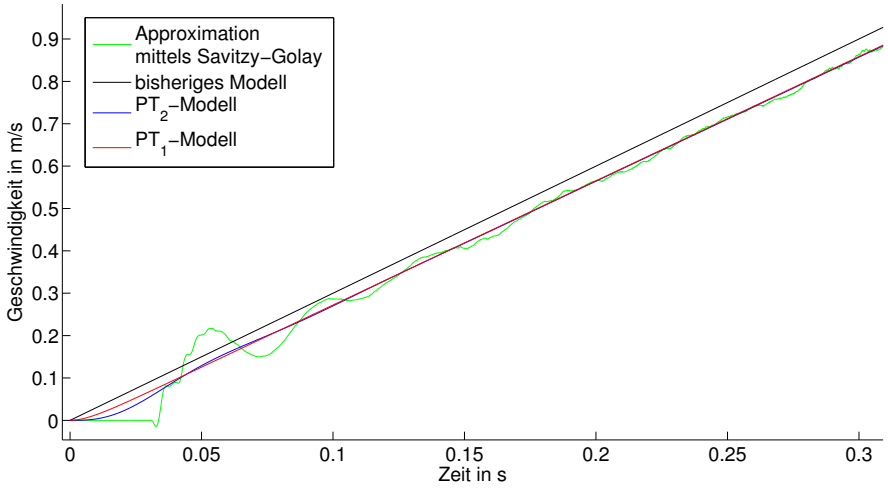


Abbildung 6.15: Vergleich der Geschwindigkeiten des Wagens beim Fahrprofil „Step1“

Schätzung der Pendelparameter

Auch hier ist es aufgrund der veränderten Motormodellierung nötig, die Pendelparameter neu zu schätzen. Analog zum Vorgehen beim PT_2-Modell verwenden wir `lsqnonlin` zur Minimierung des Abstands zwischen gemessenen Winkelwerten des Pendels und der entsprechenden Modellantwort. Die zu minimierende Zielfunktion F berechnet somit wieder den Differenzvektor aus Simulationswerten und Winkelmessungen des Fahrprofils „Upswing“ und gibt diesen zurück. Es werden erneut dieselben Fehlertoleranzen wie oben verwendet. Als Startwerte werden wiederum die Parameter aus dem bisherigen Modell gewählt: $K_1 = 0.029787$ und $K_2 = 1.466434$. Im vorliegenden Fall liefert der Algorithmus für diese Anfangswerte die

folgende Lösung:

$$K_1 = 0.01402773, \quad K_2 = 1.45998265. \tag{6.28}$$

Analog dem PT_2-Modell ändert sich insbesondere die Reibungskonstante K_1 prozentual gesehen beträchtlich im Vergleich zum bisherigen Modell.

Die visuelle Überprüfung der geschätzten Parameter erfolgt wie oben im Anschluss, der numerische Vergleich zum alten Simulationsmodells wird am Ende des Kapitels in Abschnitt 6.4 gezogen.

Verifikation des Pendelmodells

Der Vergleich der Pendelsimulation mit den Messwerten liefert ähnlich gute Ergebnisse wie bereits mit dem PT_2-Modell. Exemplarisch sei hierfür das Fahrprofil „Const1“ in Abbildung 6.16 abgebildet.

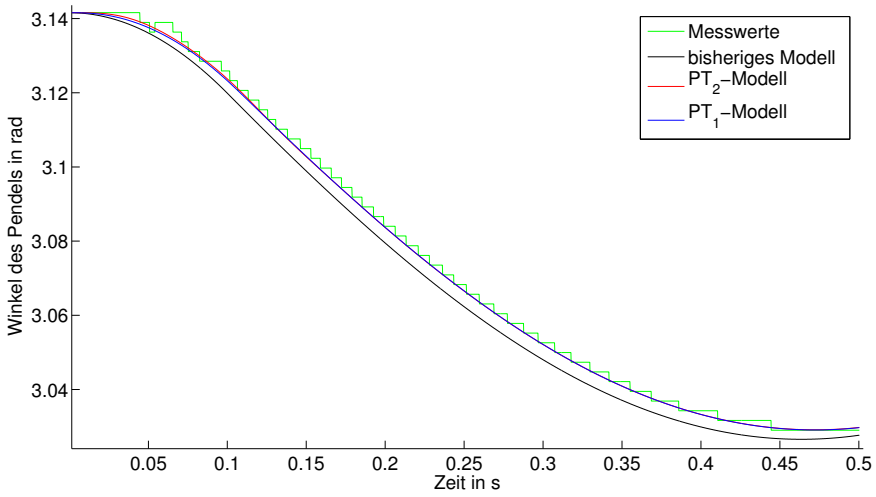


Abbildung 6.16: Simulation des Pendels mit dem PT_1-Modell und den Parametern aus (6.28) beim Fahrprofil „Const1“ im Vergleich zu den Messwerten und den anderen Modellen

6.3.3 Das PT_1-Modell

Das gefundene Simulationsmodell mit einem PT_1-Glied als Motormodell lautet somit vollständig:

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\, (g \sin x_1(t) + x_5(t) \cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= x_5(t) \\
\dot{x}_5(t) &= \frac{1}{T}\, (K\, u(t) - x_5(t))
\end{aligned} \tag{6.29}$$

mit den Konstanten

$$\begin{aligned} K_1 &= 0.01402773 \\ K_2 &= 1.45998265 \\ g &= 9.81 \\ K &= 0.97745639 \\ T &= 0.00719139. \end{aligned} \tag{6.30}$$

6.4 Vergleich von PT_2- und PT_1-Modell mit dem alten Simulationsmodell

In den Abbildungen 6.14, 6.15 und 6.16 war bereits zu erkennen, dass sich das PT_2-Modell und das PT_1-Modell ab einer Zeit von etwa 100 ms kaum mehr unterscheiden. Die Differenzen treten demnach insbesondere gleich zu Beginn auf. Aufgrund der zunächst sehr flachen Steigung des PT_2-Glieds bei einer sprunghaften Änderung der Eingangsgröße (vgl. Abbildung 6.5) kann die Verzögerung des Systems zu Beginn dadurch besser modelliert werden. Die Kurve des PT_2-Modells ist somit wesentlich schneller näher an der Kurve der Messwerte als das PT_1-Modell. Dieses Verhalten ist insbesondere im Ausschnitt der ersten 100 ms des Fahrprofils „Step3“ in Abbildung 6.17 gut zu sehen.

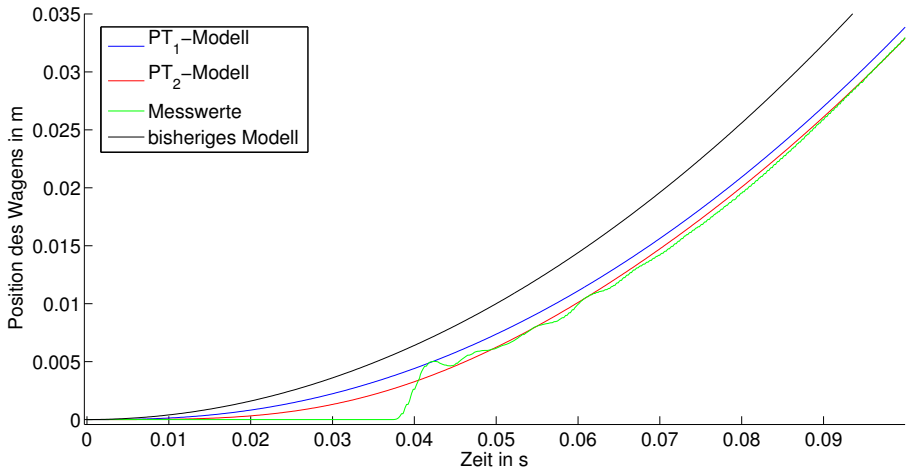


Abbildung 6.17: Simulation des Wagens in den verschiedenen Modellen im Vergleich zu den Messwerten beim Fahrprofil „Step3“

Je länger eine konstante Beschleunigung angewendet wird, desto mehr nähert sich die Kurve des PT_1-Modells der des PT_2-Modells an. Um die Verbesserungen im Vergleich zum bisherigen Simulationsmodell nicht nur grafisch darzulegen, wird dieses in Tabelle 6.1 mit den neu eingeführten PT_2- und PT_1-Modellen verglichen. Hierzu werden die Fehlerquadratsumme

	Fehlerquadratsumme in 10^{-3} m^2			Maximaler Abstand in 10^{-3} m		
Fahrprofil	altes Modell	PT_2-Modell	PT_1-Modell	altes Modell	PT_2-Modell	PT_1-Modell
Step1	405.878	1.100	4.482	20.882	1.032	2.009
Step2	115.826	0.557	1.722	10.550	0.687	1.135
Step3	492.488	1.434	7.515	27.663	2.709	3.827
Const1	62.718	0.508	1.110	5.745	1.029	1.448
Const2	162.738	0.979	2.357	9.429	1.703	2.401
Distinctive1	681.101	61.099	65.167	14.653	2.915	3.090
Distinctive2	190.692	60.014	55.231	8.343	3.499	3.132
Upswing	1298.415	68.454	71.353	12.291	2.339	3.195

Tabelle 6.1: Vergleich des PT_2- und PT_1-Modells mit dem bisherigen Modell bzgl. der Positionsmesswerte aller verwendeten Fahrprofile

und der maximale Abstand aus Simulations- und Positionsmesswerten bei allen Fahrprofilen betrachtet.

Man sieht bei beiden neuen Modellen sowohl bei der Fehlerquadratsumme als auch beim maximalen Abstand deutliche Verbesserungen im Vergleich zu vorher. Aufgrund der angesprochenen Thematik im Vergleich zwischen PT_2- und PT_1-Modell ist das letztgenannte geringfügig schlechter.

Auch bei der Geschwindigkeit soll sich die Aussage nicht nur auf grafische Ergebnisse stützen, weshalb hier, wie bereits oben, der Vergleich zwischen dem alten und den beiden neuen Modellen in Tabelle 6.2 gezogen wird - diesmal natürlich bezüglich der approximierten Geschwindigkeit. Bei der Fehlerquadratsumme ist die Verbesserung deutlich ersichtlich. Die maximale Abweichung ist hier leider nur bedingt aussagekräftig. So könnte man meinen, dass das Fahrprofil „Step3“ teilweise ein schlechteres Ergebnis als vorher abliefert. Dies liegt jedoch – wie Abbildung 6.9 gezeigt hat – nur am extremen Überschwingen zu Beginn.

	Fehlerquadratsumme in $(m/s)^2$			Maximaler Abstand in 10^{-2} m/s		
Fahrprofil	altes Modell	PT_2-Modell	PT_1-Modell	altes Modell	PT_2-Modell	PT_1-Modell
Step1	7.731	1.099	1.370	11.262	7.907	8.954
Step2	2.083	0.268	0.342	5.494	3.823	4.340
Step3	26.099	7.714	8.880	41.241	49.087	47.507
Const1	2.297	1.281	1.556	11.328	8.274	9.014
Const2	5.664	2.953	3.623	20.940	23.750	23.063
Distinctive1	9.300	0.918	0.987	5.596	2.688	2.681
Distinctive2	10.152	1.422	1.444	6.777	3.384	3.706
Upswing	59.921	21.619	23.378	54.587	52.366	51.815

Tabelle 6.2: Vergleich des PT_2- und PT_1-Modells mit dem bisherigen Modell bzgl. der Geschwindigkeitsapproximation aller verwendeten Fahrprofile

Auch der Vergleich der Pendelsimulationen zeigt eine deutliche Verbesserung im Hinblick auf Fehlerquadratsumme und maximalem Abstand zu den Messwerten bei den beiden neuen Simulationsmodellen, wie Tabelle 6.3 zeigt. Aufgrund der sehr groben Auflösung des Pendelsensors (vgl. Abbildung 6.16) gestaltet sich die numerische Differentiation nochmals schwieriger als bei den Positionsdaten. Ein aussagekräftiger Vergleich mit den Simulationswerten war aus diesem Grund nicht möglich.

	Fehlerquadratsumme in 10^{-2} rad^2			Maximaler Abstand in 10^{-3} rad		
Fahrprofil	altes Modell	PT_2-Modell	PT_1-Modell	altes Modell	PT_2-Modell	PT_1-Modell
Step1	48.200	2.942	2.764	22.719	7.297	7.778
Step2	15.567	0.812	0.100	11.591	3.639	3.884
Step3	39.285	17.157	13.327	34.720	16.095	14.770
Const1	6.015	0.395	0.409	6.300	2.775	3.318
Const2	9.661	1.447	1.218	8.965	4.445	4.024
Distinctive1	582.913	84.755	80.789	31.986	17.161	16.647
Distinctive2	265.000	33.812	35.893	32.806	16.761	16.947
Upswing	1262750.787	28578.485	28568.560	2422.694	221.062	221.357

Tabelle 6.3: Vergleich des PT_2- und PT_1-Modells mit dem bisherigen Modell bzgl. der Winkelmesswerte aller verwendeten Fahrprofile

Kapitel 7

Weitere Ideen und Ansätze zur Modellierung

Nachdem im letzten Kapitel die neuen Simulationsmodelle eingeführt wurden, werden im Folgenden weitere Ideen und Ansätze zur Modellierung kurz beschrieben. Diese stellten sich zunächst als inpraktikabel heraus, lassen sich aber unter Umständen bei intensiver Betrachtung noch mit berücksichtigen.

7.1 Modellierung mit zusätzlichem Geschwindigkeits- bzw. Positionssollwert

Bei dem in Abschnitt 6.1.1 vorgestellten PID-Modell wird als Regeldifferenz der Unterschied zwischen vorgegebener und aktueller Beschleunigung betrachtet. Eine Idee zur Erweiterung dieses Modells wäre, nicht die Differenz der Beschleunigungen zu betrachten, sondern die der Positionen bzw. Geschwindigkeiten. Man hätte somit eine sogenannte unterlagerte Positions- bzw. Geschwindigkeitsregelung. Die Tatsache, dass der PID-Regler anfangs einen großen Sprung macht und versucht, auf die Solltrajektorie der Positionskurve zu springen, erfordert eigentlich solch eine unterlagerte Regelung. Gäbe es keine solche Referenzkurve der Position bzw. Geschwindigkeit, wäre dieser extreme Sprung zu Beginn nicht zu erklären.

Um solch eine unterlagerte Regelung in unserem System zu realisieren, muss die gewünschte Sollbeschleunigung zunächst zweimal integriert werden, um die gewünschte Sollposition zu erhalten. Diese kann anschließend mit der aktuellen Ist-Situation verglichen werden. Durch zweimaliges Ableiten dieser Differenz erhält man schließlich die „Korrekturbeschleunigung“, die nötig ist, um die gewünschte Position zu erreichen. Diese Vorgehensweise zeigt Abbildung 7.1 als Blockschaltbild. Die Versuche zur Implementierung scheiterten hier jedoch, was allerdings auch an nicht konsistenten Anfangswerten für das vorliegende differential-algebraische Gleichungssystem liegen könnte. Dieses Problem wäre eventuell durch den Einsatz eines Beobachters lösbar.

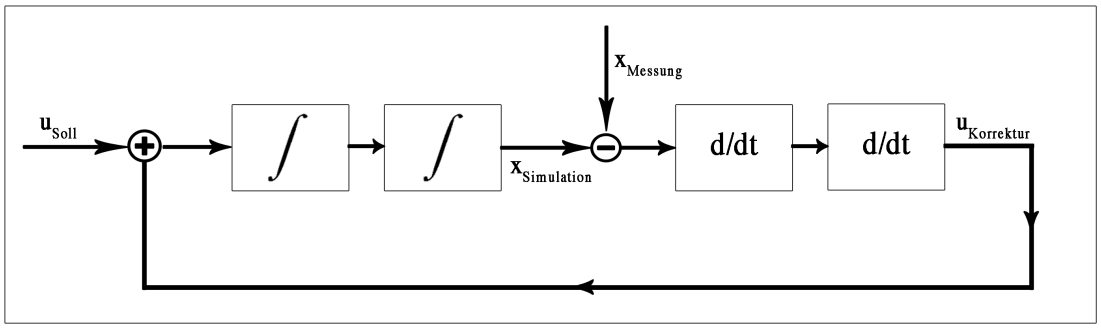


Abbildung 7.1: Berücksichtigung der aktuellen Position bei Beschleunigungsgebung

Im Folgenden wird die mathematische Herleitung dieser Idee betrachtet. Grundlage ist erneut die Gleichung des PID-Reglers

$$y(t) = K_p\, e(t) + K_i \int_{t_0}^{t} e(\tau) d\tau + K_d\, \dot{e}(t), \tag{7.1}$$

nun ist aber die Regelabweichung $e(t)$ nicht mehr die Differenz der Beschleunigungen, sondern die der Positionen:

$$e(t) = x_{\text{Soll}}(t) - x_{\text{Ist}}(t). \tag{7.2}$$

Hier ist $x_{\text{Soll}}(t) = x_3(t)$ und $x_{\text{Ist}}(t) = x_{\text{Messung}}(t)$. Die korrigierende Beschleunigung $u_{\text{Korrektur}}(t)$ erhält man anschließend durch zweimaliges Ableiten von (7.1). Pflegt man dies nun analog der Vorgehensweise in Abschnitt 6.1.1 in das bisherige Simulationsmodell ein, so ergibt sich

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\, (g \sin x_1(t) + x_5(t) \cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= u_{\text{Soll}}(t) + u_{\text{Korrektur}}(t) \\
u_{\text{Korrektur}}(t) &= \ddot{y}(t) \\
y(t) &= K_p\, e(t) + K_i \int_{t_0}^{t} e(\tau) d\tau + K_d\, \dot{e}(t) \\
e(t) &= x_3(t) - x_{\text{Messung}}(t).
\end{aligned} \tag{7.3}$$

Setzt man nun $x_5(t) := \dot{y}(t)$, so gilt $\dot{x}_5(t) = u_{\text{Korrektur}}(t)$ und es folgt:

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\, (g \sin x_1(t) + x_5(t) \cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= u_{\text{Soll}}(t) + \dot{x}_5(t) \\
x_5(t) &= \dot{y}(t) = K_p\, \dot{e}(t) + K_i\, e(t) + K_d\, \ddot{e}(t) \\
e(t) &= x_3(t) - x_{\text{Messung}}(t)
\end{aligned} \tag{7.4}$$

Mit $x_6(t) := e(t)$ und $x_7(t) := \dot{x}_6(t)$ und anschließendem Umstellen erreicht man

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\,(g \sin x_1(t) + x_5(t) \cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= u_{\text{Soll}}(t) + \dot{x}_5(t) \\
x_5(t) &= K_p\, x_7(t) + K_i\, x_6(t) + K_d\, \dot{x}_7(t) \\
\dot{x}_6(t) &= x_7(t) \\
x_6(t) &= x_3(t) - x_{\text{Messung}}(t)
\end{aligned} \tag{7.5}$$

Sortieren nach den auftretenden Ableitungen liefert schließlich:

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\,(g \sin x_1(t) + x_5(t) \cos x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) - \dot{x}_5(t) &= u_{\text{Soll}}(t) \\
0 &= x_3(t) - x_6(t) - x_{\text{Messung}}(t) \\
\dot{x}_6(t) &= x_7(t) \\
K_d\, \dot{x}_7(t) &= x_5(t) - K_i\, x_6(t) - K_p\, x_7(t)
\end{aligned} \tag{7.6}$$

Hier tritt somit wiederum ein Differentialgleichungssystem mit nicht-invertierbarer Massenmatrix auf, was erneut den Einsatz von RADAU5 bzw. anderen Lösern für differential-algebraische Probleme erfordert.

7.2 Motormodell durch lineare Approximation

Eine weitere Idee ist es, den Einfluss des Motors auf die vorgegebene Beschleunigung als lineare Abbildung zu sehen und damit zu approximieren. Man hätte dementsprechend eine Systemmatrix $A_{\text{Motor}} \in \mathbb{R}^{k \times k}$, eine Eingangsmatrix $B_{\text{Motor}} \in \mathbb{R}^{k \times 1}$ und eine Ausgangsmatrix $C_{\text{Motor}} \in \mathbb{R}^{1 \times k}$. Eingangsgröße ist hierbei die vorgegebene Beschleunigung $u(t)$, Ausgangsgröße ist die „weitergegebene“ Beschleunigung $u_{\text{Motor}}(t) =: y(t)$, die wie in Abschnitt 5.4.2 gesehen, deutlich vom Sollwert abweichen kann. Das Gesamtsystem lässt sich demnach folgendermaßen beschreiben:

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) \\
\dot{x}_2(t) &= -K_1\, x_2(t) + K_2\,(g \sin\, x_1(t) + C_{\text{Motor}}\, y(t) \cos\, x_1(t)) \\
\dot{x}_3(t) &= x_4(t) \\
\dot{x}_4(t) &= C_{\text{Motor}}\, y(t) \\
\dot{y}(t) &= A_{\text{Motor}}\, y(t) + B_{\text{Motor}}\, u(t).
\end{aligned} \tag{7.7}$$

Die Matrixeinträge werden hierbei ganz allgemein und mathematisch angegangen, ohne sich intensivere Gedanken über den physikalischen Sinn der einzelnen Einträge zu machen. Zum Ermitteln der Matrixeinträge kann hierbei die *System Identification Toolbox* in MATLAB

dienen. Dort kann aus der Angabe des Systemeingangs (d.h. bei „Step1“ konstant $u = 3$ m/s^2) und Systemausgangs (entsprechend die Beschleunigung aus Abbildung 5.16) eine lineare Approximation als Abbildung berechnet werden. Anschließend können die Matrixeinträge durch Parameterschätzen anhand einer Kleinste-Quadrate-Optimierung in `lsqnonlin` bzgl. des schwierigeren Fahrprofils „Upswing“ weiter verbessert werden.

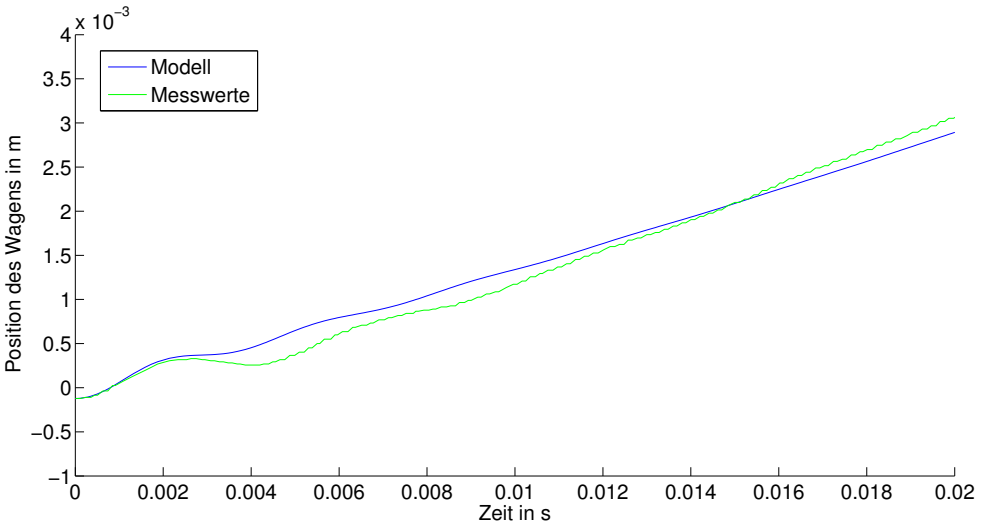


Abbildung 7.2: Simulation des Motormodells als lineare Approximation im Vergleich zu den Messwerten des Fahrversuchs „Step1“

Bei der doch eher kurzen Betrachtung in dieser Arbeit konnte zumindest festgestellt werden, dass die ersten „Rucker“ des Wagens zu Beginn zwar abgebildet werden, das System aber nach kurzer Zeit instabil wird, sodass das Motormodell nach kürzester Zeit stark von der Realität abweicht.

7.3 Reibung

Ein Aspekt, der in dieser Arbeit bisher unberücksichtigt blieb, ist die auftretende Reibung des Wagens, d.h. der Bewegungswiderstand, der sich als Widerstandskraft sich berührender Körper gegen die Einleitung einer Relativbewegung (= Haftreibung) bzw. deren Aufrechterhaltung (= Gleitreibung) äußert. [7] Dieser Abschnitt orientiert sich an [3] und [13].

Existiert keine tangentiale Relativbewegung zwischen den Oberflächen, so liegt Haftreibung vor. Dann kann in horizontaler Richtung eine gewisse Kraft F angreifen, ohne dass der betrachtete Gegenstand in Bewegung gerät. Solange F unterhalb eines Grenzwertes F_0 bleibt, ist die Haftreibung H gleich F. Bei der Grenzlast F_0 nimmt die Haftreibung seinen maximalen Wert H_0 an. Charles Augustin de Coulomb[1] hat gezeigt, dass dieser Grenzwert

[1] französischer Physiker, 1736-1806

in erster Näherung proportional zur Normalkraft N ist, d.h. zur durch die Unterlage auf den Gegenstand ausgeübten resultierenden Kraft der Normaldruckverteilung:

$$H_0 = \mu_0 N. \tag{7.8}$$

μ_0 ist der sogenannte Haftreibungskoeffizient, der lediglich von der vorliegenden Oberflächenpaarung, aber nicht von deren Größe abhängt. Der Richtungssinn der Haftreibungskraft H ist stets so, dass die Bewegung verhindert wird, d.h. gegen die gegebene Beschleunigungkraft. Der Körper haftet also, solange für die vorgegebene Kraft F gilt:

$$|F| \leq H_0 = \mu_0 N. \tag{7.9}$$

Wird diese Kraft überschritten, wird sich der betrachtete Gegenstand bewegen. Für die dann auftretende Reibungserscheinung, die sogenannte Gleitreibung R, gilt in guter Näherung, dass sie zum einen proportional zur Normalkraft N (mit Proportionalitätsfaktor μ) und zum anderen unabhängig von der Geschwindigkeit v und dieser entgegengesetzt gerichtet ist. Es gilt somit

$$R = -\mu N \frac{v}{|v|}. \tag{7.10}$$

Der Reibungskoeffizient μ ist dabei stets kleiner als der Haftreibungskoeffizient μ_0. Die beiden Formeln (7.9) und (7.10) werden auch Coulomb'sche Reibungsgesetze genannt.

In Abbildung 7.3 ist die Reibungskraft in Abhängigkeit der Geschwindigkeit aufgetragen. Hierbei wird die Unstetigkeit im Nullpunkt sowie der singuläre Charakter der Haftreibung deutlich.

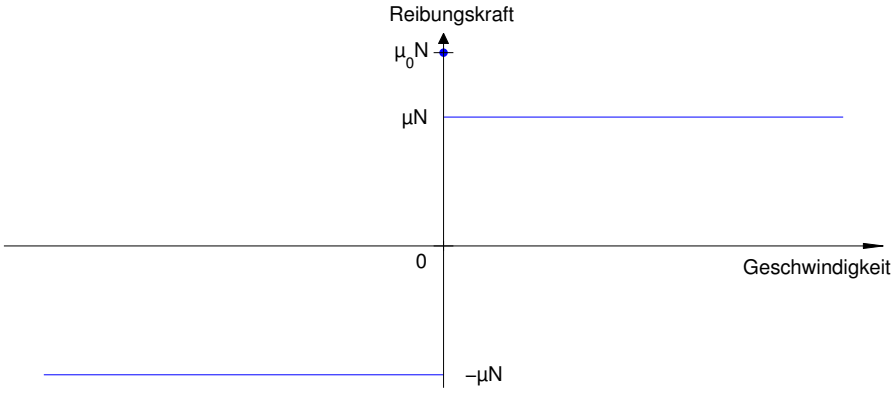


Abbildung 7.3: Abhängigkeit der Reibungskraft von der Geschwindigkeit

Möchte man die Reibungseffekte in ein Simulationsmodell einbauen, so kann die Abbildung 7.3 gezeigte Unstetigkeit zu großen Problemen bei den Differentialgleichungslösern führen.

Aus diesem Grund ist es naheliegend, den unstetigen Verlauf der Reibkraft-Geschwindigkeits-Kennlinie durch eine stetige Kennlinie zu ersetzen. Hierfür bietet sich die Verwendung der Arkustangens-Funktion

$$g(t) = \arctan\, t \tag{7.11}$$

in Kombination mit deren zweiten Ableitung

$$g''(t) = \frac{-2\,t}{(1+t^2)^2} \tag{7.12}$$

an, siehe Abbildung 7.4. Mit einer höheren Gewichtung des Arguments in der Arkustangens-Funktion ähnelt diese bereits sehr der Reibkraft-Geschwindigkeits-Kennlinie. Mit Hilfe der zweiten Ableitung lässt sich zusätzlich noch die größere Haftreibung modellieren, die es anfangs zu überwinden gilt (vgl. Abbildung 7.5). Die abgebildete Funktion lautet

$$f(t) = \arctan\,(10\,t) + \frac{2\,t}{(1+t^2)^2}. \tag{7.13}$$

Durch Anpassen der vorhandenen bzw. Einfügen weiterer Konstanten kann das regularisierte Verhalten beeinflusst und geändert werden.

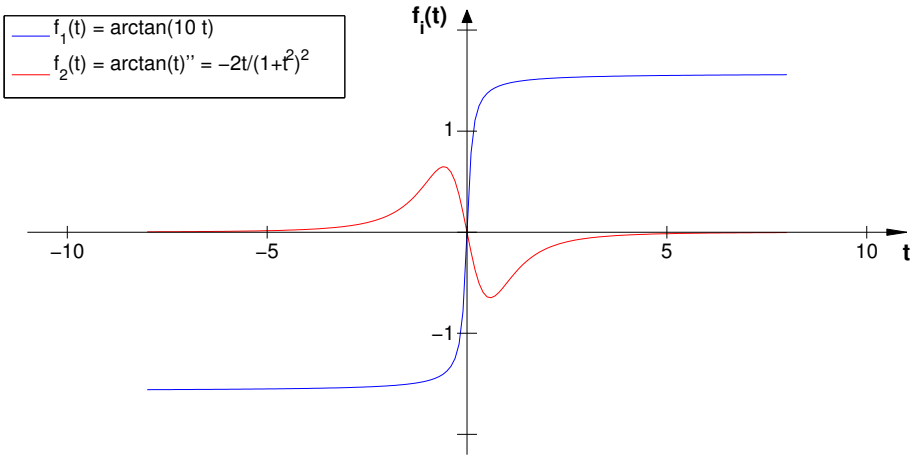


Abbildung 7.4: Arkustangens und dessen zweite Ableitung

Im Bereich geringer Geschwindigkeiten hat man nun keinen Sprung mehr, sondern lediglich eine Kurve mit hoher Steigung. Im Haftbereich entspricht die Reibung somit einer hohen viskosen Dämpfung, d.h. es sind trotz Haftens Relativgeschwindigkeiten möglich. Dieser

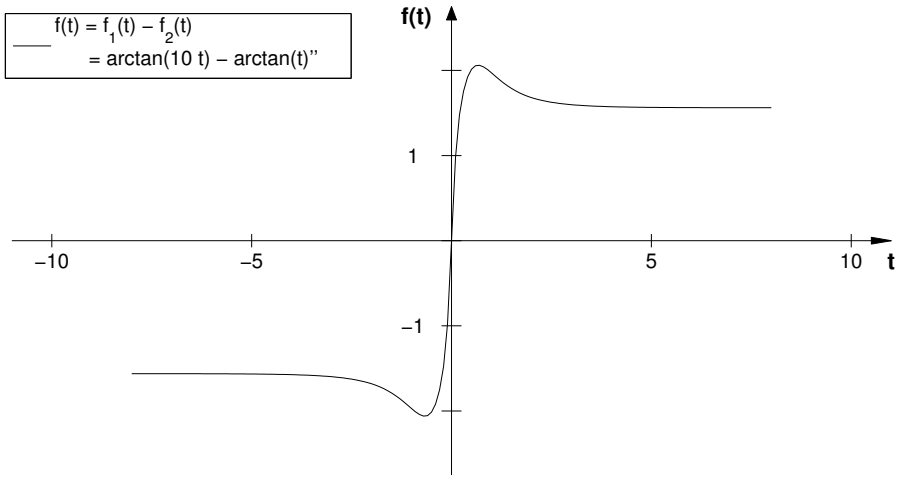


Abbildung 7.5: Regularisierte Reibungskraft durch Arkustangens-Funktion

Nachteil wird jedoch aufgewogen durch den Vorteil, dass die Dynamik des Systems nun als gewöhnliche Differentialgleichung mit stetiger rechter Seite formuliert werden kann. Diese Bewegungsgleichung hat durch die hohe Steigung der Kennlinie zwar den Makel, dass sie steif ist, aber es existieren für diese Art Differentialgleichungen genügend etablierte Lösungsverfahren. Dies setzt bei Echtzeitreglern natürlich wiederum ausreichend viel Zeit für die Berechnung der Lösung voraus. Geschieht – wie hier – eine solche Umformulierung mit nur qualitativ begründbaren Modifikationen lediglich zum Zwecke der Berechenbarkeit, so spricht man von Regularisierung. Ebenso wäre eine ereignisgesteuerte Integration des Systems mit Reibung denkbar. [31]

Die Modellierung der Gleitreibung gestaltet sich sehr schwierig, da bei realen technischen Gleitvorgängen stets eine Überlagerung der verschiedenen Reibungskomponenten auftreten kann. Dazu zählen die Adhäsionskomponente, die Deformationskomponente und die Furchungskomponente der Gleitreibung[2]. Aus diesem Grund ist eine theoretische Abschätzung von Reibungszahlen im Allgemeinen nicht möglich.

Auch wenn die Modellierungsversuche mit dem Ansatz aus (7.13) – bei Verwendung unterschiedlicher Konstanten und somit diverser Reibkraft-Geschwindigkeits-Kennlinien – erfolglos blieb, soll hier kurz auf die Beobachtungen bezüglich der Wagenreibung eingegangen werden. Der mechanische Aufbau des Wagens besitzt mehrere Komponenten, die eine Reibung verursachen. Zum einen ist es die Befestigung des Schlittens auf den beiden Schienen. Das Seilzugsystem besitzt zudem zwei Umlenkzahnräder und auch das Getriebe und der Motor haben einen wesentlichen Anteil an der Reibung.

Das stark verzögerte Losfahren des Wagens zu Beginn der Fahrversuche legt den Verdacht nahe, dass die Haftreibung vergleichsweise hoch ist. In Versuchen konnte festgestellt werden, dass sich die Zielposition des Wagens bei unterschiedlichen Startpositionen nicht unterschei-

[2]Für eine Erläuterung dieser Begriffe sei auf die Literatur des Fachbereichs Tribologie verwiesen, z.B. in [7]

det. Das bedeutet aber nicht zwangsweise, dass die Haftreibung an allen Startpositionen gleich ist. Es ist auch möglich, dass der integrierte Regler im Servomotor dies ausgleicht und diese Differenz deshalb in der Endposition nicht mehr erkennbar ist. Wie bereits zu sehen war (vgl. Abbildung 4.3) konnte der Regler auch unterschiedliche Latenzen in derselben Startposition ausgleichen.

Darüber hinaus wurden mehrere Versuchsläufe sowohl mit sehr trockener Bahn, als auch mit sehr stark geölter Bahn durchgeführt, um eventuell auftretende Unterschiede bei der Gleitreibung herausstellen zu können. Es kann festgehalten werden, dass diesbezüglich nur marginale Differenzen von 0.1 mm in der Endposition des Wagens festgestellt wurden, was wiederum innerhalb der Messungenauigkeit liegt (vgl. Abschnitt 4.2.3). Dies spricht dafür, dass die vorhandene Gleitreibung grundsätzlich sehr gering ist und nicht ins Gewicht fällt.

Kapitel 8

Zusammenfassung

Ziel dieser Arbeit war eine Verbesserung des Simulationsmodells für das vorliegende inverse Pendel von Googol Technology. Nachdem das grundlegende Modell des inversen Pendels eingeführt wurde, war zunächst eine intensive Beschäftigung mit der Programmierung und Steuerung des Motors nötig, um eine Testumgebung in C++ entwickeln zu können, mit der verschiedenartige Fahrversuche inklusive Messung der Positions- und Winkeldaten durchgeführt werden können. Die Messdaten von vier verschiedenen Fahrversuchen bildeten die Grundlage für die darauffolgende Analyse der Geschwindigkeit und der Beschleunigung des Schlittens. In dieser war anschließend ersichtlich, dass sich der Wagen in der Zeit des Losfahrens nicht – wie zu erwarten wäre – polynomial fortbewegt, sondern dass nach großer Verzögerungszeit ein enormer Sprung der Beschleunigung stattfindet. Es wurde zudem gezeigt, dass mit diesem Sprung die „verlorene“ Zeit dennoch nicht komplett wettgemacht werden kann, weshalb der Wagen stets hinter der bisher erwarteten Position und Geschwindigkeit zurückblieb.

Da im Servocontroller des Motors ein PID-Regler arbeitet, wurde dieser zunächst ins Modell integriert. Nachdem sich diese Vorgehensweise als erfolglos herausstellte, wurde der Motor mit einem PT_2-Glied approximiert. Es zeigte sich, dass dies eine sehr gute Näherung für das Modell des Wagens liefert. Die Ergebnisse der Simulationen waren überaus zufriedenstellend und stimmten sehr gut mit den durchgeführten Messungen überein. Schließlich wurden die Parameter der Pendelgleichung geschätzt, um ein vollständiges Simulationsmodell zu erhalten. Im Anschluss daran wurde eine weitere Vereinfachung in Form eines PT_1-Modells eingeführt. Dieses ist zwar etwas ungenauer als das zuvor eingeführte PT_2-Modell, im Vergleich zu diesem aber einfacher zu lösen und immer noch wesentlich genauer als das ursprüngliche Modell.

Abschließend wurden noch kurz weitere Ideen und Ansätze zur Modellierung betrachtet.

Zusammenfassend lässt sich sagen, dass somit zwei exaktere Simulationsmodelle für den Wagen eingeführt wurden und dies nun als Grundlage für das weitere Arbeiten am Pendel gesehen werden kann. So könnte das nun vorliegende Testprogramm zukünftigen Studenten dazu dienen, weitere Analysen durchzuführen um eventuell die Parameter der Pendelgleichung nicht wie hier nur zu schätzen, sondern diese auch grundlegend intensiver zu spezifizieren. Darüber hinaus wäre z.B. die Implementierung eines Beobachters als weiterer Schritt sicherlich sinnvoll. Das gefundene PT_2-Modell ist zwar sehr exakt, die zugrundelie-

genden Differentialgleichungen aufgrund der enormen Spitzen bei Beschleunigungswechseln (vgl. Abbildung 6.6) aber extrem steif. Diese Stellen können numerisch problematisch werden und setzen einen sehr genauen Löser voraus. Man hat also eine genaue Simulation, diese geht jedoch mit einer langen Rechenzeit einher. Möchte man nun eine Echtzeitregelung mit kurzen Steuerungssequenzen realisieren, so könnten die einzelnen Takte nicht lang genug sein, um darin das steife Differentialgleichungssystem exakt lösen zu können. In diesem Fall ist es eventuell besser, das ein wenig ungenauere aber dafür schneller zu lösende PT_1-Modell für die Simulation zu verwenden, da das PT_2-Modell bei ungenauer Lösung anfälliger für Fehler wird.

Es ist somit immer von der konkreten Anwendungssituation abhängig, welches Modell im Endeffekt die besseren Ergebnisse in Bezug auf die Regelung des Pendels liefert. Beide Modelle haben ihre Vor- und Nachteile, weshalb stets ein Trade-Off zwischen Kosten und Qualität – sprich Berechnungszeit und Genauigkeit – nötig ist.

Anhang A

Inhalt der beiliegenden CD

Auf der beiliegenden CD befinden neben der PDF-Version dieser Arbeit alle Programme, die in diese Arbeit eingegangen sind. Dies ist zum einen das C/C++-Programm zur Durchführung der Fahrversuche am Pendel. Zum anderen sind es die Implementierungen in MATLAB der in dieser Arbeit verwendeten Verfahren und die Routinen, mit denen die dargestellten Grafiken erzeugt wurden. Darüber hinaus befinden sich diverse Benutzerhandbücher sowie die Internetquellen als PDF-Dokumente auf der CD.

Im Folgenden ist ein Dateiverzeichnis der CD aufgeführt.

/CPP

`/AccControl.cpp`	Abgeleitete Klasse von `PendulumBase.cpp` zur Kapselung der Beschleunigungsgebung
`/acccontrol.h`	Headerfile der Aufrufmethoden in `AccControl.cpp`
`/axisoff-close.exe`	Im Falle einer nicht erfolgreichen Beendigung des Fahrversuch-Programms kann mit dieser ausführbaren Datei die aktuell aktive Achse geschlossen und der Motor ausgeschaltet werden.
`/GT400.DLL`	Programmbibliothek zur Steuerung des Pendels
`/IP_Testumgebung.cbp`	Code::Blocks-Projektdatei
`/main.cpp`	Hauptprogramm zur Durchführung der Fahrversuche
`/PendulumBase.cpp`	Klasse zur Kapselung der Aufrufmethoden des Motion Controllers
`/pendulumbase.h`	Headerfile der Aufrufmethoden in `PendulumBase.cpp`
`/StopWatch.cpp`	Klasse zur Zeitmessung während der Fahrversuche
`/stopwatch.h`	Headerfile der Klassendefinition in `StopWatch.cpp`
`/Userlib.h`	Headerfile der Aufrufmethoden in `GT400.DLL`

/MATLAB

Die MATLAB-Routinen sind im Allgemeinen als Hauptprogramme implementiert, d.h. sie enthalten alle notwendigen Parameter und können direkt ausgeführt werden. Eine Änderung der Parameter kann direkt im Programmkopf vorgenommen werden. Ausnahmen sind Hilfsroutinen, die von diesen Programmen aufgerufen werden. Diese sind als Funktionen implementiert und benötigen eine Parameterübergabe. Die Beschreibung der notwendigen Eingabeparameter und der Rückgabewerte ist in jeder Funktion als Kommentar vor dem Code enthalten.

/MATLAB/DATA

Das Verzeichnis enthält alle aufgenommenen Mess- und Beschleunigungsdaten.

/MATLAB/KAPITEL3

`/vcurve.m`	Programm zur Erstellung des Geschwindigkeitsprofils in Abb. 3.3

/MATLAB/KAPITEL4

`/preparing_data.m`	Funktion zum Einlesen der Mess- und Beschleunigungswerte
`/vgl_messungen.m`	Routine zum Vergleich der Messungen eines Fahrprofils (Abb. 4.2, 4.3, 4.4)

/MATLAB/KAPITEL5

`/A_anfangsbeschl.m`	Berechnung der Beschleunigung gleich zu Beginn der Messaufzeichungen (Abb. 5.13)
`/A_DQ_MA_SG.m`	Berechnung der Beschleunigung aus den Positionsdaten mittels Differenzenquotienten nach Anwendung von Moving-Average-Filter und Savitzky-Golay-Filter, vorheriges Downsampling möglich (Abb. 5.11, 5.12)
`/A_glm_beschl.m`	Für jede aufgenommene Position des Wagens wird diejenige gleichmäßige Beschleunigung berechnet, die nötig wäre, um diesen Ort in der angegebenen Zeit zu erreichen (Abb. 5.18)
`/A_PF_intervall.m`	Stückweises Polynomfitting zur Ermittlung der zweiten Ableitung, d.h. der Beschleunigung (Abb. 5.16, 5.17)
`/A_PF.m`	Berechnung der Beschleunigung aus den Positionsdaten mittels Polynomfitting mit Polynomen 2. und 3. Grades (Abb. 5.14, 5.15)

/delay_mean_std.m	Betrachtung der Verzögerung zu Beginn: Berechnung von Mittelwert, Standardabweichung, Varianz, minimale und maximale Verzögerung zu Beginn der Messungen
/ode_sys_alt.m	Differentialgleichungssystem des bisherigen Modells
/ode_sys_poly.m	Differentialgleichungssystem für A_PF_intervall.m
/preparing_data.m	Funktion zum Einlesen der Mess- und Beschleunigungswerte
/SG_beispiele.m	Diese Routine beinhaltet zwei Beispiele für die Anwendung des Savitzky-Golay-Filters (Abb. 5.4)
/V_DQ_MA_SG.m	Berechnung der Geschwindigkeit aus den Positionsdaten mittels Differenzenquotienten nach Anwendung von Moving-Average-Filter und Savitzky-Golay-Filter (mit unterschiedlichen Filterlängen) und mit Hilfe der Savitzy-Golay-Koeffizienten (aus implizitem Polynomfitting) nach Anwendung des Moving-Average-Filters (Abb. 5.8, 5.9, 5.10)
/V_DQ_MA.m	Berechnung der Geschwindigkeit aus den Positionsdaten mittels Differenzenquotienten nach Anwendung des Moving-Average-Filters mit unterschiedlichen Filterlängen (Abb. 5.6, 5.7)

/MATLAB/KAPITEL6_PID

Der Ordner enthält alle Dateien, die zum MEX-Interface des impliziten Lösers RADAU5 gehören. Des Weiteren sind folgende Dateien enthalten:

/main_opti.m	Routine zur Optimierung der Parameter für das PID-Modell des Wagens anhand der gemessenen Positionsdaten und anschliessendes Plotten des Ergebnisses (Abb. 6.2, 6.3, 6.4)
/min_PID_para.m	Funktion zur Berechnung der Zielfunktion (Fehlerquadratsumme aus Simulation und Messwerten)
/ode_sys_alt.m	Differentialgleichungssystem des bisherigen Modells
/ode_sys_PID.m	Differentialgleichungssystem des PID-Modells
/preparing_data.m	Funktion zum Einlesen der Mess- und Beschleunigungswerte

/MATLAB/KAPITEL6_PT2_PT1

Der Suffix „_P“ im Dateinamen sagt aus, dass in diesem Programm/dieser Funktion die Pendelparameter betrachtet werden. „_W“ steht entsprechend für die Betrachtung der Wagenparameter. Die Zeilen zur Anwendung der Optimierung wurden jeweils auskommentiert, um die Plots direkt ausgeben zu können.

/main_PT1_vgl_v.m	Vergleich des PT_1-Modells mit der numerisch berechneten Geschwindigkeit mittels Savitzky-Golay-Filter (Polynomfitting) und dem alten Modell (Abb. 6.15)

/main_PT1_opti_P.m	Routine zur Optimierung der Pendelparameter für das PT_1-Modell anhand der gemessenen Winkeldaten und anschließendes Plotten des Ergebnisses (Abb. 6.16)
/main_PT1_opti_W.m	Routine zur Optimierung der Parameter für das PT_1-Modell des Wagens anhand der gemessenen Positionsdaten und anschließendes Plotten des Ergebnisses (Abb. 6.13, 6.14, 6.17)
/main_PT2_vgl_v.m	Vergleich des PT_2-Modells mit der numerisch berechneten Geschwindigkeit mittels Savitzky-Golay-Filter (Polynomfitting) und dem alten Modell (Abb. 6.9, 6.10)
/main_PT2_opti_P.m	Routine zur Optimierung der Pendelparameter für das PT_2-Modell anhand der gemessenen Winkeldaten und anschließendes Plotten des Ergebnisses (Abb. 6.11, 6.12)
/main_PT2_opti_W.m	Routine zur Optimierung der Parameter für das PT_2-Modell des Wagens anhand der gemessenen Positionsdaten und anschließendes Plotten des Ergebnisses (Abb. 6.6, 6.7, 6.8)
/main_PT2_step.m	Beispielhafte Sprungantwort eines PT_2-Glieds (K=3, T=0.1, d=0.2/1.0/2.0), vgl. Abb. 6.5
/min_PT1_P.m	Funktion zur Berechnung des Differenzenvektors aus Simulation und Positionsmesswerten
/min_PT1_W.m	Funktion zur Berechnung des Differenzenvektors aus Simulation und Winkelmesswerten
/min_PT2_P.m	Funktion zur Berechnung des Differenzenvektors aus Simulation und Positionsmesswerten
/min_PT1_W.m	Funktion zur Berechnung des Differenzenvektors aus Simulation und Winkelmesswerten
/ode_sys_alt.m	Differentialgleichungssystem des bisherigen Modells
/ode_sys_PT1_P.m	Differentialgleichungssystem des PT_1-Modells mit variablen Pendelparametern
/ode_sys_PT1_W.m	Differentialgleichungssystem des PT_1-Modells mit variablen Wagenparametern
/ode_sys_PT1.m	Differentialgleichungssystem des PT_1-Modells
/ode_sys_PT2_P.m	Differentialgleichungssystem des PT_2-Modells mit variablen Pendelparametern
/ode_sys_PT2_W.m	Differentialgleichungssystem des PT_2-Modells mit variablen Wagenparametern
/ode_sys_PT2.m	Differentialgleichungssystem des PT_2-Modells
/preparing_data.m	Funktion zum Einlesen der Mess- und Beschleunigungswerte

/MATLAB/KAPITEL7

Neben den unten aufgeführten Dateien beinhaltet dieser Ordner noch Messdaten, bei denen die Verzögerung zu Beginn abgeschnitten wurde und eine MATLAB-Routine, die die Koordinatenachsen in einem Plot ins Zentrum der Grafik verschiebt.

`/lin_motor.sid`	Gespeicherte Session der System Identification Toolbox zur Approximation des Motors als lineares Modell
`/main_lin_motor.m`	Routine zur Optimierung der Matrixeinträge fuer die lineare Approximation des Motors. Die Startwerte für die Matrizen wurden durch die System Identification Toolbox gefunden, siehe `lin_motor.sid` (Abb. 7.2)
`/main_reibung.m`	Routine zur Erstellung der Abb. 7.3, 7.4, 7.5
`/min_lin_motor.m`	Funktion zur Berechnung des Differenzenvektors aus Simulation und Positionsmesswerten
`/ode_sys_alt.m`	Differentialgleichungssystem des bisherigen Modells
`/ode_sys_lin_motor.m`	Differentialgleichungssystem des neuen Modells
`/preparing_data.m`	Funktion zum Einlesen der Mess- und Beschleunigungswerte

/DOCS

`/GT_Programming_Manual.pdf`	Programmierhandbuch des Motion Controllers
`/GT_User_Manual.pdf`	Benutzerhandbuch des Motion Controllers
`/IP_Experimental_Manual.pdf`	Allgemeine Einführung in die Stabilisierung inverser Pendel
`/IP_Instruction_Manual.pdf`	Erläuterungen und Aufbauhinweise für das inverse Pendel von Googol Technology
`/PA_Minas_A4.pdf`	Handbuch für den Servomotor von Panasonic
`/RADAU5.pdf`	Beschreibung des MEX-Interface zu RADAU5.F von Hairer und Wanner

/QUELLEN

In diesem Verzeichnis befinden sich die Internetquellen [4], [20], [23], [24], [32] und [34] als PDF-Dokumente.

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] Ascher, U.M. ; Petzold, L.R.: *Computer Methods for Ordinary Differential Equations and Differental-Algebraic Equations*. Society for Industrial and Applied Mathematics, Philadelphia, 1998

[2] Aulbach, B.: *Gewöhnliche Differenzialgleichungen*. Elsevier GmbH, München, 2004

[3] Balke, H.: *Einführung in die Technische Mechanik*. Springer-Verlag, Berlin Heidelberg New York, 2007

[4] Becker, M.: *Synchronisation von Threads/Mutexe*. `http://www.ijon.de/comp/tutorials/threads/synchro.html`. Version: 22.10.2012

[5] Berger, B.: *Realisierung einer prototypischen Hardwarelösung für ein inverses Pendel: FPGA-only based control for a very compact inverted pendulum*. GRIN Verlag, München, 2004

[6] Czado, C. ; Schmidt, T.: *Mathematische Statistik*. Springer-Verlag, Heidelberg u.a., 2011

[7] Czichos, H. ; Habig, K.-H.: *Tribologie-Handbuch*. Vieweg+Teubner Verlag, Wiesbaden, 2010

[8] Fischerauer, G.: *Messtechnik*. Vorlesungsskript, Universität Bayreuth, 2011

[9] Gander, W. ; Hrebicek, J.: *Solving Problems in Scientific Computing Using Maple and MATLAB*. Springer-Verlag, Berlin Heidelberg New York, 2002

[10] Googol Technology (HK) Ltd.: *Programming Manual For GT Series Motion Controller*. 2003

[11] Googol Technology (HK) Ltd.: *Inverted Pendulum - Experimental Manual*. 2006

[12] Googol Technology (HK) Ltd.: *Inverted Pendulum - Instruction Manual*. 2006

[13] Gross, D. ; Hauger, W. ; Schröder, J. ; Hall, W.: *Technische Mechanik*. Springer-Verlag, Berlin Heidelberg New York, 2006

[14] Grüne, L.: *Mathematische Kontrolltheorie I - Lineare Systeme*. Vorlesungsskript, Universität Bayreuth, WS 2006-07

[15] Grüne, L.: *Modellierung mit Differentialgleichungen.* Vorlesungsskript, Universität Bayreuth, WS 2006-07

[16] Grüne, L. ; Pannek, J.: *Nonlinear Model Predictive Control.* Springer-Verlag, London u.a., 2011

[17] Hairer, E. ; Wanner, G.: *Solving Ordinary Differential Equations II.* Springer-Verlag, Berlin Heidelberg New York, 2002

[18] Kreiss, J.-P. ; Neuhaus, G.: *Einführung in die Zeitreihenanalyse.* Springer-Verlag, Heidelberg u.a., 2006

[19] Kunkel, P. ; Mehrmann, V.: *Differential-Algebraic Equations.* European Mathematical Society Publishing House, Zürich, 2006

[20] Ludwig, C.: *M3/Software - ODE MEXfiles Homepage.* `http://www-m3.ma.tum.de/Software/ODEWebHome`. Version: 22.10.2012

[21] Lunze, J.: *Regelungstechnik 1.* Springer-Verlag, Berlin Heidelberg, 2001

[22] Mann, H. ; Schiffelgen, H. ; Froriep, R.: *Einführung in die Regelungstechnik.* Carl Hanser Verlag, München, 2001

[23] Panasonic Electric Works Deutschland GmbH: *Plug and Play Positioniersystem mit Panasonic Kleinsteuerungen und Servoantrieben.* `http://www.panasonic-electric-works.com/peweu/de/downloads/D-021.pdf`. Version: 22.10.2012

[24] Panasonic Electric Works Europe AG: *Motion Control Solutions: Minas A4/A4N/E Servo Drives/FP-Series PLCs.* `http://www.panasonic-electric-works.com/peweu/en/downloads/ds_637_en_minas_a4_a4n_e.pdf`. Version: 22.10.2012

[25] Paulssen, J.: *Das Problem der Kurvenanpassung.* 2009

[26] Press, W. H. ; Vetterline, W. T. ; Teukolsky, S. A. ; Flannery, B. P.: *Numerical Recipes in Fortran.* Cambridge University Press, New York, 1996. – 644–649 S.

[27] Rauber, T. ; Rünger, G.: *Multicore: Parallele Programmierung.* Springer-Verlag, Berlin Heidelberg, 2008

[28] Riordon, J. ; Zubritsky, E. ; Newman, A.: Top 10 Articles. In: *Analytical Chemistry* 72 (2000), Nr. 9, S. 324–329

[29] Rosenheinrich, W.: *Numerische Mathematik.* Lehrmaterial, Fachhochschule Jena, 2002

[30] Schröder, J. ; Gockel, T. ; Dillmann, Rüdiger: *Embedded Linux.* Springer-Verlag, Berlin Heidelberg, 2009

[31] Stamm, W.: *Modellierung und Simulation von Mehrkörpersystemen mit flächigen Reibkontakten*. KIT Scientific Publishing, Karlsruhe, 2009

[32] The MathWorks Inc.: *Create or alter options structure for ordinary differential equation solvers*. `http://www.mathworks.de/de/help/matlab/ref/odeset.html#f92-1023650`. Version: 20.10.2012

[33] Unbehauen, H.: *Regelungstechnik I*. Vieweg+Teubner Verlag, Wiesbaden, 2008

[34] Wiedl, W.: *Lösung von Anfangswertproblemen mit den Matlab Integratoren*. `http://www1.uni-hamburg.de/W.Wiedl/Skripte/Matlab/ODE/ODE-File.html`. Version: 24.10.2012

[35] Zahn, M.: *Unix-Netzwerkprogrammierung mit Threads, Sockets und SSL*. Springer-Verlag, Berlin Heidelberg, 2006

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Bayreuth, den 30. Oktober 2012

(Christoph Baumann)