Modellierung und Simulation eines bio-inspirierten Roboterbeins

Modelling and Simulation of a Bio-Inspired Robotic Leg

Bachelor-Thesis von Manuel Krönig

April 2012

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Modellierung und Simulation eines bio-inspirierten Roboterbeins
Modelling and Simulation of a Bio-Inspired Robotic Leg

Vorgelegte Bachelor-Thesis von Manuel Krönig

1. Gutachten: Prof. Dr. Oskar von Stryk
2. Gutachten: M.Sc. Stefan Kurowski

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 12. April 2012

(Manuel Krönig)

Zusammenfassung

In dieser theoretischen Arbeit werden zwei verschiedene Frameworks zur Simulation von mechanischen Systemen untersucht. Das erste Framework ist SimMechanics, eine kommerzielle Toolbox für Matlab/Simulink. Das zweite Framework ist mbslib, eine von Dr.-Ing. Martin Friedmann entwickelte C++-Bibliothek. In den beiden Simulations-Frameworks wird ein bio-inspiriertes Roboterbein modelliert. Die erstellten Modelle werden eingesetzt, um eine Optimierung der Parameter des Beins hinsichtlich maximaler Kraftausübung beim Gehen vorzunehmen.

Der zentrale Teil der Arbeit ist der Vergleich der beiden Frameworks. Zusätzlich zum Hinterbein werden mehrere physikalische Systeme modelliert und der Ablauf der Simulation und die Simulationsergebnisse untersucht. Bei den Vergleichen wird insbesondere darauf geachtet, wie aufwändig die Modelle erstellt und erweitert werden können und wie einfach es ist, eine Simulation durchzuführen. Zur besseren Vergleichbarkeit wurde ein Visualisierungsprogramm erstellt, welches es ermöglicht, die Simulationsergebnisse aus beiden Frameworks darzustellen. Es ergibt sich, dass mbslib sehr gut verwendbar ist, aber einen Nachteil bei der Simulation hat, der durch den verwendeten Integrator verursacht wird. Diese Probleme könnten durch den Austausch des Integrators gelöst werden. Dies ist möglich, da der Quelltext von mbslib verfügbar ist. Bessere Integratoren (mit Schrittweitensteuerung) befinden sich in der Implementierung.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Übersicht über den Aufbau der Arbeit . . . 1
- 1.3 Quelltext und Videos . . . 2

2 Grundlagen **3**
- 2.1 Simulation von Mehrkörpersystemen . . . 3
- 2.2 Bio-inspiriertes Roboterbein . . . 4

3 Stand der Forschung **5**

4 Durchführung und Auswertung **7**
- 4.1 Visualisierung . . . 7
- 4.2 Mathematisches Pendel . . . 7
- 4.3 Federpendel . . . 12
- 4.4 Bodenkontakt . . . 15
- 4.5 Dreifachpendel . . . 17
- 4.6 Hinterbein . . . 21
- 4.7 Seilzüge . . . 29
- 4.8 Optimierung . . . 31

5 Diskussion **35**
- 5.1 Vergleich . . . 35
- 5.2 Zusammenfassung . . . 38
- 5.3 Ausblick . . . 39

Abbildungsverzeichnis **40**

Tabellenverzeichnis **41**

Listingverzeichnis **42**

Literaturverzeichnis **43**

Anhang **44**
- Quelltext . . . 44

1 Einleitung

Das Ziel dieser Arbeit ist die Modellierung und Simulation eines bio-inspirierten Roboterbeins in zwei verschiedenen Simulations-Frameworks. Dabei sollen die Unterschiede in der Art der Modellierung und die Ergebnisse der Simulation verglichen werden. In beiden Frameworks soll ein Optimierungsvorgang durchgeführt werden. Für das Hinterbein ist die Maximierung der Schrittweite und das Erreichen einer maximalen Kraftentwicklung beim Gehen ein interessantes Optimierungsziel. Ausgehend von diesen beiden möglichen Zielen wird die Maximierung der nach unten entwickelten Kraft untersucht.

1.1 Motivation

Bei der Entwicklung von Robotern oder Roboterkomponenten ist eine Simulation der Dynamik sehr wichtig. Mit Hilfe einer Simulation lässt sich das Verhalten der Komponenten untersuchen, ohne dass diese tatsächlich gebaut werden müssen. Dadurch lassen sich beträchtliche Mengen an Zeit und Geld sparen. Es kann untersucht werden, ob geplante Aufbauten den Anforderungen genügen. Es ist außerdem möglich, die Auswirkungen von Änderungen der Parameter des Aufbaus zeitnah zu untersuchen. Ein Optimierungsvorgang wäre ohne Simulation sehr teuer und würde lange dauern. Optimierungen sind wichtig, da mit ihnen Konstruktionen hinsichtlich verschiedener Eigenschaften (z.B. Kosten, Funktionalität, etc.) verbessert werden können.

SimMechanics und mbslib sind zwei Frameworks, mit denen solche Simulationen durchgeführt werden können. SimMechanics ist eine kommerzielle Software, die häufig verwendet wird und in der viele anerkannte Algorithmen implementiert sind. mbslib ist eine quelloffene Eigenentwicklung der Arbeitsgruppe des Autors dieser Arbeit. Beide Frameworks verfolgen verschiedene Modellierungs-Strategien: In SimMechanics werden die Modelle grafisch programmiert; mbslib ist eine C++-Bibliothek, mit der die Modelle über ein objektorientiertes Interface aufgebaut werden. Der Vergleich beider Frameworks ist auf Grund dieser unterschiedlichen Hintergründe interessant. Es wäre natürlich ein Vorteil, wenn die Verwendung der kostengünstigeren Lösung uneingeschränkt zu empfehlen wäre.

Die Entwicklung von Robotern, deren Aufbau am Muskel- und Skelettapparat von Tieren und Menschen orientiert ist, gewinnt immer größere Bedeutung. Die Evolution bewirkt, dass der Aufbau der Bewegungsapparate in der Natur für den jeweiligen Lebensraum gut geeignet ist. Durch die Übertragung dieser Konzepte auf Roboter erhofft man, die durch die Evolution verursachte Optimierung zu nutzen und eine hohe Funktionalität bei geringem Energiebedarf zu erhalten. Für mobile Roboter, die die Flexibilität der gehenden Fortbewegung (z.B. größere Unabhängigkeit vom Untergrund) nutzen sollen, bietet sich aus diesem Grund die Untersuchung eines bio-inspirierten Roboterbeins an.

1.2 Übersicht über den Aufbau der Arbeit

Im Folgenden wird ein kurzer Überblick über die Struktur dieser Arbeit gegeben: In Kapitel 2 werden kurz die Grundlagen erläutert, die zum Verständnis der Arbeit notwendig sind. Dabei wird erklärt, was Mehrkörpersysteme sind und wie sie simuliert werden können. Die beiden untersuchten Simulations-Frameworks werden kurz vorgestellt. Anschließend wird das zu untersuchende Roboterbein, unter Erwähnung möglicher Vorteile des Aufbaus, kurz erläutert. In Kapitel 3 werden einige ausgewählte Arbeiten vorgestellt, die sich mit Simulation von Mehrkörpersystemen und elastischen Robotern beschäftigen. In Kapitel 4 wird die Modellierung und Simulation verschiedener Systeme detailliert behandelt. Für jedes System wird erst der Aufbau angegeben, dann die zugehörige Theorie besprochen, die Modellierung beschrieben und die Simulations-Ergebnisse sowohl untereinander als auch mit der Theorie verglichen. Zuerst werden die Unterschiede der Modellierung bei Verwendung der beiden Frameworks

am Beispiel eines mathematischen Pendels aufgezeigt. Danach wird ein Modellierung eines Federpendels betrachtet. Zur Modellierung des Roboterbeins wird eine Modellierung der Interaktion mit dem Boden benötigt: der Bodenkontakt. An diesem Beispiel wird die Erstellung eigener Komponenten in beiden Frameworks verglichen. An einem Dreifachpendel wird die Simulation eines chaotischen Systems und danach das biologisch-inspirierte Hinterbein untersucht. Da bei der Simulation des Hinterbeins Unterschiede zwischen den Frameworks auftreten, werden die verwendeten Seilzüge separat betrachtet. Zuletzt wird eine Optimierung der Befestigungsposition der Seilzüge am Hinterbein vorgenommen. In Kapitel 5 werden die beiden Frameworks verglichen und die Ergebnisse zusammengefasst. Dabei wird ein Fazit gezogen und ein Ausblick auf den weiteren Forschungs- und Entwicklungsbedarf gegeben.

1.3 Quelltext und Videos

Der Quelltext der mbslib-Modelle ist vollständig im Anhang aufgeführt. Da die Details der SimMechanics-Modelle nur in Unterdialogen erkennbar sind, werden sie nicht im Anhang aufgeführt. Alle erstellten Dateien sind in einem Subversion-Projektarchiv (*repository*) der Arbeitsgruppe abgelegt. Hier sind auch mit Hilfe des Visualisierungs-Programms erzeugte Videos der Simulationsergebnisse zu finden.

2 Grundlagen

2.1 Simulation von Mehrkörpersystemen

Ein Mehrkörpersystem (engl. *multibody system, mbs*) besteht aus starren und flexiblen Elementen, die über Verzweigungen, Drehgelenke und Schubgelenke miteinander verbunden sind. Aus diesen Komponenten können Teile von Robotern oder ganze Roboter modelliert werden. Mit einer Dynamik-Simulation lassen sich sowohl das zeitabhängige Verhalten als auch Interaktionen mit der Umgebung (externe Kräfte) untersuchen[1].

Es gibt viele Simulations-Frameworks. Häufig verwendet werden unter anderem *Adams*, *SimMechanics*, *ODE (Open Dynamics Engine)* und *DynaMechs*. Die beiden erstgenannten sind kommerzielle Produkte, die zwei letztgenannten frei verfügbare Open-Source-Software.

Diese Arbeit befasst sich nur mit SimMechanics und mbslib. SimMechanics wird betrachtet, da es ein anerkanntes Standardprogramm für die Untersuchung von mechatronischen System ist. mbslib ist eine Eigenentwicklung der Arbeitsgruppe des Autors. Im Folgenden werden beide Frameworks kurz vorgestellt:

SimMechanics:

SimMechanics ist eine Toolbox für *Matlab/Simulink*. Es ist eine kommerzielle Software der Firma MathWorks. Sie erlaubt es, Modelle von Mehrkörpersystemen grafisch aus Modulen aufzubauen und zu simulieren. Aus bestehenden Modulen und zusätzlichem Matlab-Code können eigene Module geschrieben werden. SimMechanics verfügt über eine eigene Visualisierung, verschiedene Integratoren mit automatischer Schrittweitensteuerung (als Teil von Simulink) und einen Frontend zur Optimierung. Geschlossene kinematische Ketten können modelliert und simuliert werden. Die erstellten Modelle können mittels des *Simulink Coders* in C-Code konvertiert werden. Diese konvertierten Modelle können unabhängig von Simulink ausgeführt werden. Sie sind schnell und können im *Rapid Prototyping* eingesetzt werden. SimMechanics wird in vielen Veröffentlichungen verwendet. In Tabelle 2.1 sind die Versionen der in dieser Arbeit eingesetzten Matlab/Simulink-Komponenten aufgeführt.

Komponente	Version
Matlab	7.9.0.529 (R2009b)
Optimization Toolbox	4.3 (R2009b)
Parallel Computing Toolbox	4.2 (R2009b)
SimMechanics	3.1.1 (R2009b)
Simscape	3.2 (R2009b)
Simulink	7.4 (R2009b)

Tabelle 2.1: Versionen der verwendeten Matlab/Simulink-Komponenten.

mbslib:

mbslib ist eine C++-Bibliothek. Sie ist nicht kommerziell und wurde von Dr.-Ing. Martin Friedmann am Fachbereich des Autors dieser Arbeit entwickelt. Mehrkörpersysteme werden als Baum modelliert, der über ein objektorientiertes Interface aufgebaut wird. Aufbauend auf dem Modell ist eine Dynamik-Simulation über den Articulated-Body-Algorithm und einen Integrator möglich. Es ist nur ein einziger Integrator (einfacher Euler-Integrator) vorhanden, der über keine Schrittweitensteuerung verfügt. mbslib unterstützt, im Gegensatz zu SimMechanics, keine geschlossenen kinematischen Ketten. Es gibt keine Visualisierung. mbslib wird nur in wenigen Veröffentlichungen erwähnt.

2.2 Bio-inspiriertes Roboterbein

Das zu modellierende Roboterbein (siehe Abbildung 2.1 ganz rechts) besteht aus drei Gliedern (schwarze Linien), die mit Drehgelenken (Kreise) verbunden sind. Das mittlere Glied ist als Pantograf ausgeführt. Über elastische Seilzüge (orange und blaue Linien) können Kräfte auf die Glieder ausgeübt werden. Der elastische Pantografen-Aufbau bietet Vorteile gegenüber einem konventionellen, mit Gleichstrom-Motoren aktuierten Aufbau: Er ist leichter und verbraucht weniger Energie bei der Bewegung, schränkt aber, trotz einer Verringerung der Bewegungsfreiheit, die Funktionsweise des Beins nicht ein. Der Aufbau des Beins (mit Achilles-Sehne) ähnelt dem Hinterbein eines Schäferhundes (siehe Abbildung 2.1, erste bis dritte Abbildung). In der ersten Abbildung ist die Skelettstruktur eines solchen Hinterbeins dargestellt. In der zweiten Abbildung ist ein Teil der Muskel- und Sehnen-Struktur hinzugefügt. Eine schematische Darstellung dieses Aufbaus wird in der dritten Abbildung gezeigt. Ersetzt man die elastischen Kopplungen durch starre Glieder, so erhält man den oben beschriebenen Aufbau. Untersuchungen haben gezeigt, dass dieser funktional ausreichend ist[2, 3].

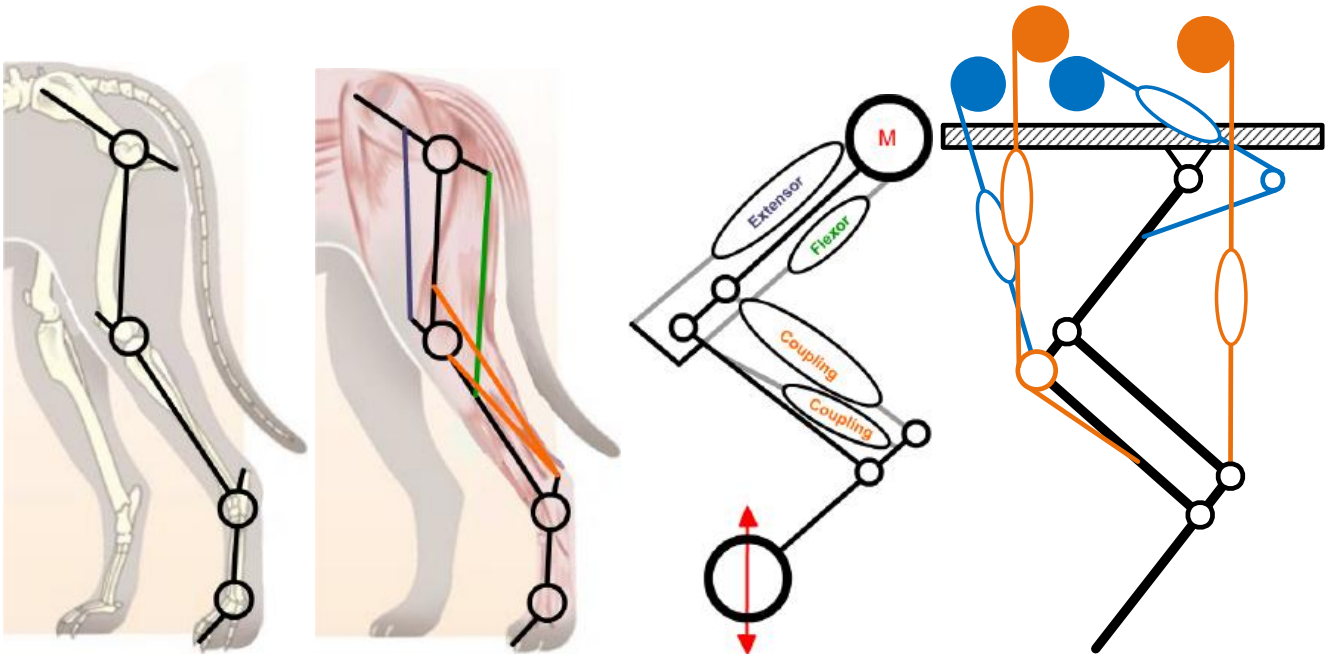

Abbildung 2.1: Erste bis dritte Abbildung: Aufbau eines Hinterbeins eines Schäferhundes (aus [3]). Vierte Abbildung: Bio-inspiriertes Hinterbein.

3 Stand der Forschung

Das Hauptaugenmerk dieser Arbeit ist der Vergleich der Modellierungs- und Simulationsframeworks SimMechanics und mbslib. Mit dieser Fragestellung hat sich bisher keine andere Arbeit auseinandergesetzt.

mbslib:

Die bisherigen Veröffentlichungen von Dr.-Ing. Martin Friedmann et. al[4, 5] zu mbslib beschränken sich hauptsächlich auf die vorhandenen Funktionen und geben ein Beispiel für die Art der Modellierung anhand des BioBipeds an. Der BioBiped ist ein zweibeiniger Laufroboter. Sein Bein ist eine Nachbildung der Skelettstruktur und der Muskeln eines menschlichen Beins aus Starrkörpern, Federn und Seilzügen[6]. Hierbei wird die Übertragung der Struktur des Beins auf eine Baumstruktur zur Modellierung in mbslib erklärt (siehe Abbildung 3.1). In der linken Abbildung wird die mechanische Struktur des Beins aufgezeigt. In der mittleren Abbildung werden Details zur Position von Schwerpunkt und Federn dargestellt. In der rechten Abbildung wird die resultierende Baumstruktur gezeigt.

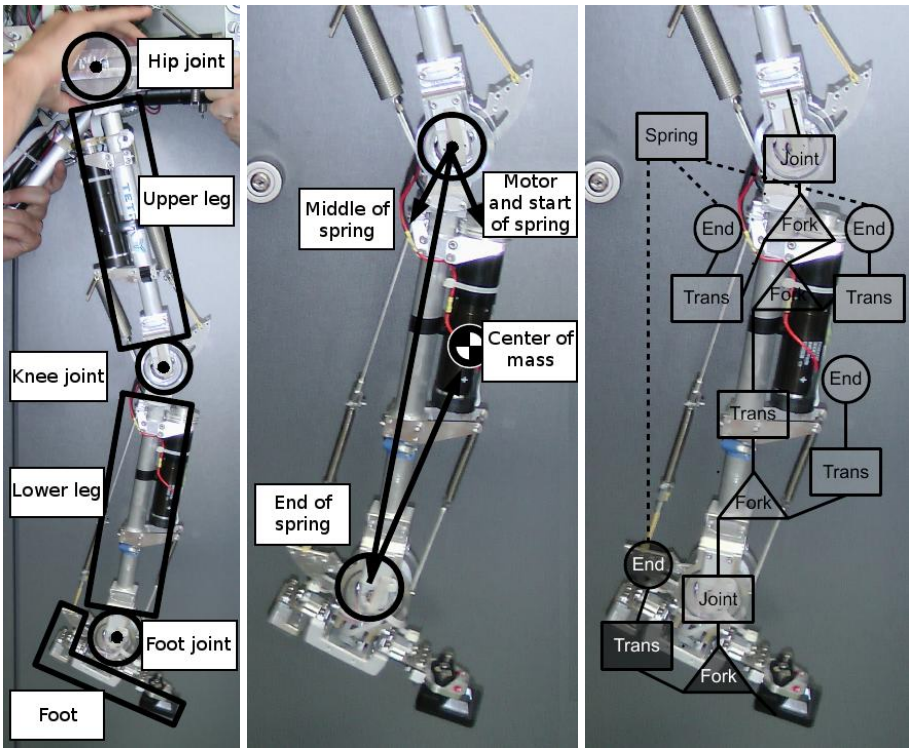

Abbildung 3.1: Beispiel für die Baumstruktur eines Roboterbeins (aus [4, 5]). Links: Aufbau. Mitte: Details. Rechts: Baumstruktur in mbslib.

SimMechanics:

Es gibt viele Veröffentlichungen, in denen die Verwendung von SimMechanics beschrieben wird: Arndt et. al[7] haben SimMechanics zur Modellierung eines ferngesteuerten Modellautos verwendet, um die Dynamik des Autos im Moment des Kippens zu untersuchen. Das Ziel ihrer Arbeit ist das automatische Balancieren eines Autos auf zwei Rädern.

Unver et. al[8] haben SimMechanics verwendet, um die Parameter eines Geckobots zu optimieren. Der Geckobot ist ein Roboter, der mittels adhäsiver Flächen an der Unterseite der Füße, ähnlich wie ein Gecko, an Wänden entlanggehen soll. Das Ziel ihrer Optimierung war die Vergrößerung der maximalen Neigung der Oberfläche, bei der der Roboter stabil an ihr haftet.

Stelzer et. al[9] untersuchten die Elastizität von Industrierobotern beim Hochgeschwindigkeitsschneiden. Elastizitäten in den Gelenken schränken die Schneidegenauigkeit ein. Diese Positionsungenauigkeiten sollen durch gezieltes Gegensteuern ausgeglichen werden. Dazu wird ein Modell eines fünfarmigen Roboters in SimMechanics erstellt. Die im Experiment gemessenen Abweichungen entsprechen qualitativ den simulierten Abweichungen.

Lens et. al[1] entwickelten ein auf SimMechanics aufbauendes Toolkit zur Modellierung und Simulation von Manipulatoren und Robotern mit hoher Gelenkelastizität unter Verwendung realistischer Kontaktkräfte. Sie führen folgenden Vorteile bei der Verwendung von SimMechanics auf: Die leichte Integrierbarkeit in andere Programme, die einfache Modellierung und die Möglichkeit zur Generierung von stand-alone ausführbaren Programmen aus den Modellen. Zudem seien in SimMechanics viele anerkannte Algorithmen zur Lösung von Bewegungsgleichungen implementiert.

Auf Grund der thematischen Ähnlichkeit ist der folgende Artikel besonders interessant: Abele et. al[10] vergleichen die Modellierung und Simulation in Adams und SimMechanics am Beispiel eines Industrieroboters mit elastischen Gelenken. Adams wird dabei als weitläufig anerkannte Software bezeichet. SimMechanics hingegen wird wegen der Eignung für schnelle Modellerstellung und Debugging in der Matlab-Umgebung hervorgehoben. Abele et. al vergleichen die Simulation anhand eines ISO 9283 Pfades. Die Modelle in Adams und SimMechanics ohne Spiel liefern gleiche Ergebnisse. Bei den Modellen mit Spiel gibt es kleine Abweichungen, die auf Modellierungsunterschiede zurückgeführt werden. Das Gelenkspiel wurde in Adams als Spline und in SimMechanics als abschnittsweise definierte Funktion modelliert. Die Ausführungszeit beider Simulationen wird als vergleichbar bezeichnet. Zum Zeitpunkt der Veröffentlichung des Artikels wurde noch kein Vergleich der simulierten Ergebnisse mit einem Experiment durchgeführt.

Elastische Roboter:

Hardt et. al[2] untersuchten den Bewegungsapparat von Tier und Mensch. Bei beiden werden Bewegungen durch einen Apparat aus elastischen Sehnen und Muskeln angetrieben. Diese Elastizitäten speichern einen Teil der Energie und federn Stöße ab. Die gespeicherte Energie kann wieder abgegeben werden. Mit einem elastischen Aufbau kann somit Energie eingespart werden, die bei einem starren Aufbau z.B. zur Erzeugung von Bremskräften zur Stabilisierung verbraucht wird. Als weitere Möglichkeit zur Energie-Einsparung wird die Entkopplung von Antriebs- und Tragemechanismus, z.B. durch einen Pantografen-Aufbau, genannt.

Radkhah et. al[3] entwickelten die Struktur eines bio-inspirierten Beins für einen vierbeinigen Roboter. Die Struktur wird aus dem Aufbau eines Hinterbeins eines Schäferhundes abgeleitet. Der vorgestellte Aufbau entspricht der Glieder-Struktur des in dieser Arbeit zu modellierenden Beins.

4 Durchführung und Auswertung

4.1 Visualisierung

Eine Fehlersuche ist bei der Entwicklung eines komplexen Modells in mbslib ohne eine Visualisierung sehr umständlich. Aus diesem Grund wurde von mir eine externe Visualisierung (*mbsplot*) über die Python-Bibliothek matplotlib erstellt. matplotlib wird auch in *ROS* (*Robot Operating System*) verwendet und bietet viele Möglichkeiten zur Erstellung von zweidimensionalen Diagrammen, die in Python leicht zu verwenden sind. Seit Version 1.1 besitzt matplotlib ein *backend*-unabhängiges Animations-Framework, sodass Animationen erstellt werden können, ohne dabei auf einen *Timer* des *GUI*-Toolkits zugreifen zu müssen.

mbsplot kann Daten über *stdin* oder aus einer Datei erhalten. Es können sowohl Daten aus mbslib als auch von Matlab erzeugte mat-Dateien (z.B. aus SimMechanics) unter Verwendung der Bibliothek *scipy* eingelesen werden. Somit ist es auch möglich SimMechanics-Modelle darzustellen. Die Visualisierung ist nicht modell-spezifisch, da die Animation aus beliebig vielen Linienzüge mit separat konfigurierbaren Grafik-Optionen aufgebaut wird. Die interne Visualisierung von SimMechanics erfolgt automatisch, allerdings erlaubt sie nur die Anzeige von Starrkörpern und Gelenken. Eigene Module können nicht automatisch visualisiert werden. mbsplot hat gegenüber der SimMechanics-Visualisierung den Vorteil, dass der genaue Verlauf der Seilzüge beobachtet werden kann. Die erzeugten Animationen können als Video-Datei gespeichert werden. Dies soll laut Beschreibung auch mit der internen Visualisierung von SimMechanics möglich sein. Bei einem während dieser Arbeit durchgeführten Test wurde allerdings eine stark verzerrte Video-Datei erstellt. Die SimMechanics-Visualisierung verfügt, im Gegensatz zu mbsplot, das nur die x-y-Ebene darstellen kann, über eine frei drehbare dreidimensionale Ansicht.

4.2 Mathematisches Pendel

Zum Vergleich der Modellierung beider Frameworks bietet sich ein mathematisches Pendel an. Es hat einen einfachen Aufbau. Zudem ist das physikalisch korrekte Verhalten eines Pendels bekannt.

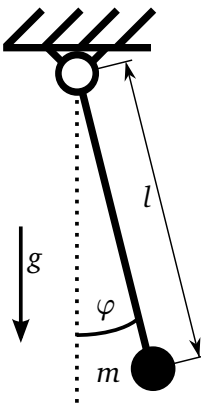

Abbildung 4.1: Aufbau des mathematischen Pendels.

4.2.1 Aufbau

Ein mathematisches Pendel (siehe Abbildung 4.1) ist eine ausdehnungslose Punktmasse (mit Masse m), die an einer massenlosen, infinitesimal dünnen, starren Stange (mit Länge l) befestigt ist. Die Stange ist

ohne Reibung an einem Gelenk um eine Achse drehbar aufgehängt. Der Winkel, um den die Stange aus der Ruhelage (verdeutlicht durch die gepunktete Linie) verdreht ist, heiße im Folgenden φ. Die Stange selbst hat keinen Schwerpunkt und keine Trägheit.

4.2.2 Theorie

Um die Dynamik eines Systems zu untersuchen, kann ein Phasenraumdiagramm hilfreich sein. Dabei wird, im eindimensionalen Fall, die Winkelgeschwindigkeit über der Auslenkung aufgetragen. In einem ungedämpften System (also ohne Energieverlust) erwartet man eine geschlossene Kurve. Kollabiert die Kurve zu einem Fixpunkt (stabile Spirale), verliert das System Energie. In Abbildung 4.2 sind Phasenraumtrajektorien für ein ungedämpftes Pendel mit verschiedenen Anfangsbedingungen dargestellt.

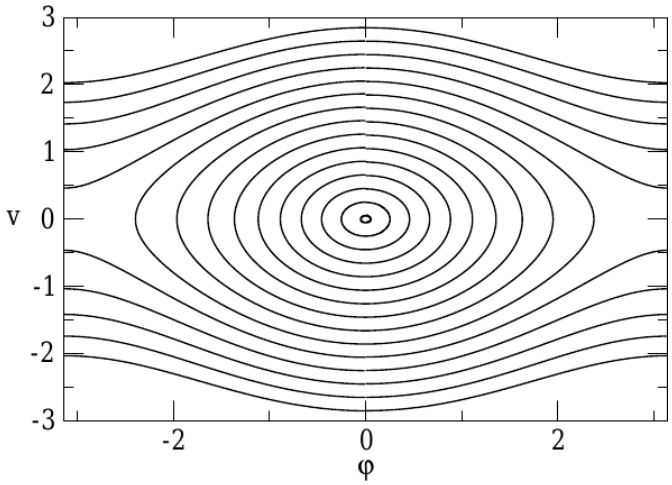

Abbildung 4.2: Trajektorien eines reibungsfreien Pendels im Phasenraum (aus [11]).

4.2.3 Modellierung (ohne Reibung)

Im Folgenden werden die Unterschiede der Modellierung in SimMechanics (grafische Programmierung) und in mbslib (objektorientierte Programmierung) gezeigt. In beiden Fällen hat die Stange des modellierten Pendels eine Länge von $l = 1$ m und die Punktmasse eine Masse von $m = 1$ kg. Die Erdbeschleunigung wirkt in die negative y-Richtung und hat den üblichen Wert $g = 9{,}81$ m/s^2. Die Rotationsachse ist die z-Achse (in der Darstellung senkrecht zum Papier). Die folgenden Anfangsbedingungen werden verwendet:

- Auslenkung $\varphi = 9°$,
- Geschwindigkeit $\dot{\varphi} = 0°/\text{s}$ und
- Beschleunigung $\ddot{\varphi} = 0°/\text{s}^2$.

SimMechanics:

Modelle in SimMechanics (siehe Abbildung 4.3) bestehen aus Blöcken mit Anschlüssen, die mit Linien verbunden sind. Blöcke werden durch verschiedene Symbole dargestellt, die ihre Funktion zeigen sollen. Die Anschlüsse werden in der Abbildung durch kleine Kreise, Rauten oder Quadrate dargestellt. Sie haben verschiedene Funktionen.

Der mechanische Aufbau des Pendels besteht aus drei Blöcken: dem Befestigungspunkt (in der Abbildung *Ground* genannt), einem Drehgelenk (*Revolute*) und einem Starrkörper (*Body*). Mit dem Befestigungspunkt ist ein Modul verbunden, mit dem sich die Einstellungen der Umgebung (z.B. Gravitation)

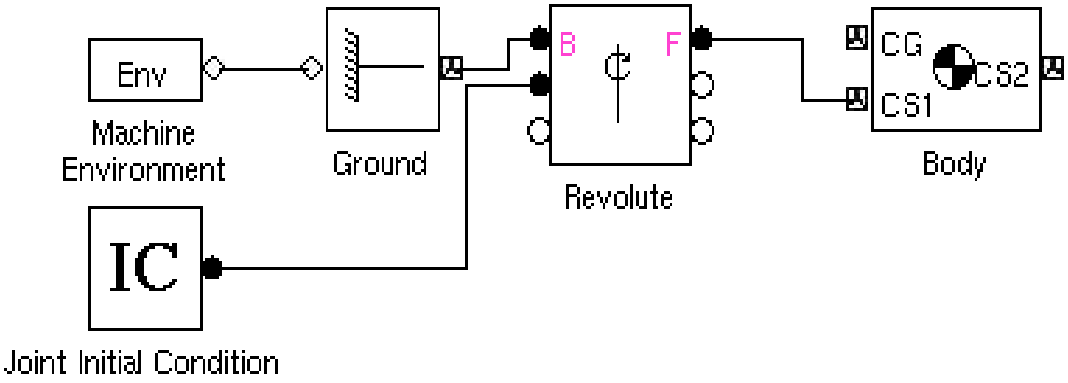

Abbildung 4.3: Modellierung des reibungsfreien mathematischen Pendels in SimMechanics.

ändern lassen. Das Drehgelenk ist mit einem Modul verbunden, das es ermöglicht, die Anfangsbedingungen zu setzen. Die Eigenschaften der Blöcke lassen sich durch Doppelklick in einem Dialog ändern. In Abbildung 4.4 sind die Einstellungen des Starrkörpers dargestellt. Hier können der Trägheitstensor, die Masse und die Position des Schwerpunktes eingestellt werden. Es lassen sich beliebig viele Anschlüsse zur Verbindung mit anderen Blöcken erstellen.

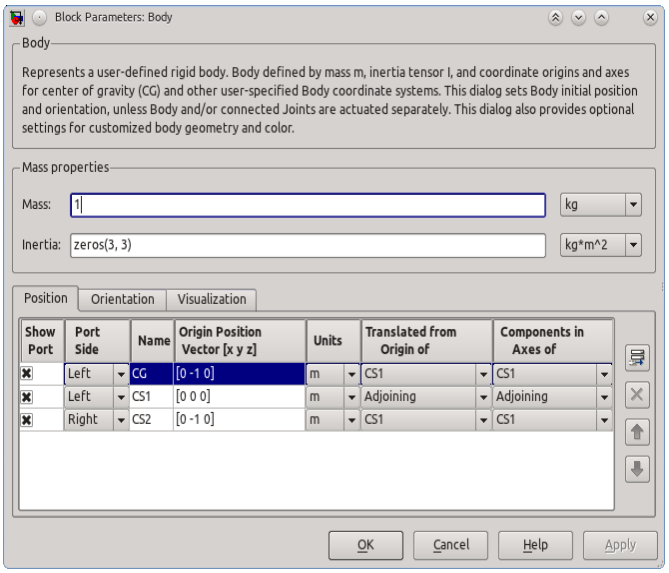

Abbildung 4.4: Eigenschaften des Starrkörpers in SimMechanics.

mbslib:

Modelle in mbslib werden wie normale C++-Programme geschrieben. Der Quelltext des Pendels ist im Anhang 5.1 komplett dargestellt. Im Folgenden werden die einzelnen Befehle erläutert:

```
1 #include <mbs/MbsCompoundWithBuilder.h>
2
3 using namespace mbslib;
4 int main(void)
5 {
```

Hier wird die mbslib wie eine übliche C++-Bibliothek eingebunden. Die Klassen der mbslib befinden sich im namespace mbslib. Damit die Befehle im Folgenden besser lesbar sind, wird hier der namespace des Programms gesetzt. Zudem werden einige leere Zeilen weggelassen, weshalb die Zeilennummern nicht fortlaufend sind. Das Modell wird in der main-Methode aufgebaut:

```
6   MbsCompoundWithBuilder mbs("Mathematisches␣Pendel");
```

Hier wird eine Instanz des MbsCompoundWithBuilder erzeugt. MbsCompoundWithBuilder ist eine Klasse, die ein Mehrkörpersystem repräsentiert und über Methoden verfügt, mit denen die einzelnen Komponenten hinzugefügt werden.

```
8   mbs.addFixedBase("Anker");
```

Eine Basis mit einer festen Position wird hinzugefügt.

```
9   Joint1DOF * j1 = mbs.addRevoluteJoint(TVector3(0, 0, 1), "Drehgelenk");
```

Ein Drehgelenk um die z-Achse wird hinzugefügt. Dabei wird ein Zeiger auf das Gelenk gespeichert, um die Anfangsbedingungen einstellen zu können.

```
10  mbs.addRigidLink(TVector3(0, -1, 0), TVector3(0, 0, 0),
11                   0, TMatrix3x3::Zero());
```

Ein Starrkörper mit Endpunkt in (0, -1, 0), dessen Schwerpunkt im Anfangspunkt liegt, der eine Masse von 0 kg und keinen Trägheitstensor hat, wird hinzugefügt.

```
12  mbs.addEndpoint(1, TMatrix3x3::Zero(), "Masse");
```

Eine Punktmasse mit Masse $m = 1$ kg und einem Null-Trägheitstensor wird hinzugefügt.

```
14  mbs.setGravitation(TVector3(0, -9.81, 0));
15  j1->setJointPosition(9 * M_PI / 180);
```

Die Gravitation und die Anfangsbedingungen werden gesetzt.

```
17  for (int i = 0; i <= 20 / 0.0001; i++)
18  {
19    mbs.doABA();
20    mbs.integrate(0.0001);
21  }
```

Die Dynamik wird mittels des *Articulated-Body Algorithm* und Integration simuliert. Die Integration verwendet eine feste Schrittrate. Jeder Schritt muss einzeln aufgerufen werden. Um das Beispiel kurz zu halten, wird hier auf Ausgabe der Ergebnisse verzichtet.

```
22  return 0;
23 }
```

4.2.4 Modellierung (mit Reibung)

Im Folgenden wird gezeigt, welche Änderungen nötig sind, um eine Reibung von 1 Nms/rad im Drehgelenk hinzuzufügen:

SimMechanics:

In SimMechanics ist keine vorgefertigte Gelenkreibung vorhanden. Sie kann durch Auslesen der Gelenkgeschwindigkeit, Multiplikation des Wertes mit dem Negativen des Reibungskoeffizienten und Generierung eines Drehmoments, das auf das Gelenk wirkt, erzeugt werden. Die benötigten Änderungen sind in Abbildung 4.5 markiert.

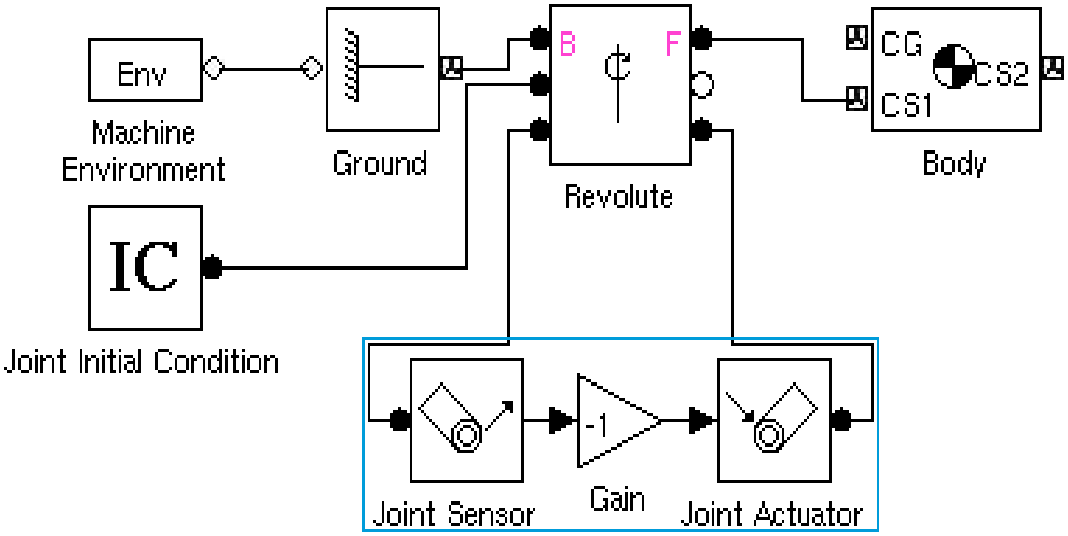

Abbildung 4.5: Modellierung des gedämpften mathematischen Pendels in SimMechanics. Änderungen gegenüber der reibungsfreien Version sind blau markiert.

mbslib:

Gelenkreibung wird von mbslib ohne Änderungen unterstützt. Zur Verwendung der Funktionalität muss nur der Reibungskoeffizient gesetzt werden. Dazu wird

```
16  j1->setParameter(0, 1);
```

in Zeile 16 eingefügt.

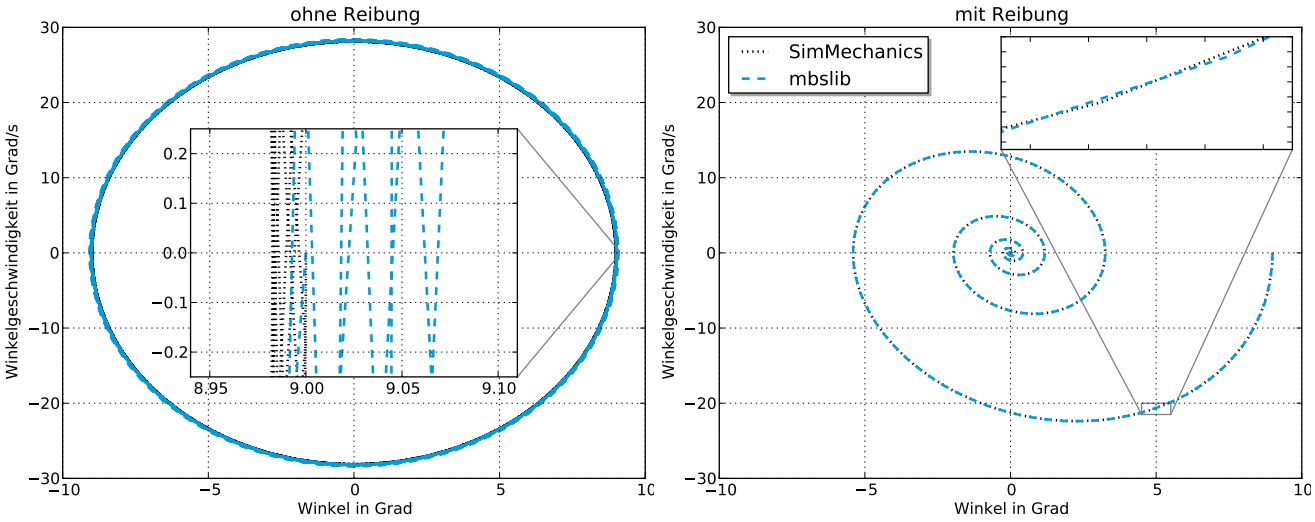

Abbildung 4.6: Trajektorie eines Pendels im Phasenraum. Links: reibungsfrei. Rechts: mit Reibung.

4.2.5 Ergebnisse der Simulation

Das reibungsfreie Pendel schwingt mit konstanter Periode und Amplitude. Die Trajektorien im Phasenraum sind in Abbildung 4.6 links dargestellt. Ausgehend von der Anfangsposition ($\varphi = 9°$ und $\dot{\varphi} = 0°/s$) ist die Trajektorie sowohl in SimMechanics (gepunktete schwarze Linie) als auch in mbslib (gestrichelte blaue Linie) scheinbar geschlossen. Dies bedeutet, dass, wie erwartet, keine Energie verloren geht. Betrachtet man aber die extreme Vergrößerung, so kann man sehen, dass in SimMechanics die Energie

minimal (Änderung der Auslenkung $< 0{,}01°$) abnimmt, während sie in mbslib leicht zunimmt (Änderung der Auslenkung $\approx 0{,}07°$). Die Energieänderung in SimMechanics ist vernachlässigbar und bei numerischen Verfahren zu erwarten, während die Änderung durch den Euler-Integrator in mbslib signifikante Verfälschungen verursachen kann (siehe Abschnitt 5.1).

Die Amplitude des gedämpften Pendels nimmt schnell ab. Die Trajektorien im Phasenraum sind in Abbildung 4.6 rechts dargestellt. In beiden Framworks kann eine identische stabile Spirale, ausgehend von der gleichen Ausgangsposition wie zuvor, beobachtet werden. Der Abstand der einzelnen Punkte ist in den beiden Abbildungen unterschiedlich, da SimMechanics eine Schrittweitensteuerung verwendet. Diese bewirkt, dass der zeitliche Abstand der ersten Punkte kleiner als der der restlichen Punkte ist. Die Vergrößerung bestätigt diese Eindrücke.

4.3 Federpendel

Die Modellierung eines Federpendels als einfaches Beispiel für ein elastisches System bietet sich als Vorunteruschung zur Modellierung des bio-inspirierten Roboterbeins an, da beim Roboterbein viele elastische Komponenten auftreten.

4.3.1 Aufbau

Ein ideales Federpendel (siehe Abbildung 4.7) besteht aus einer ausdehnungslosen Punktmasse (mit Masse m), der an einer massenlosen Feder (mit Federkonstante k und Länge l_F) angebracht ist. Die aktuelle Länge der Feder und damit die Höhe der Punktmasse unter der Befestigung wird im Folgenden mit l bezeichnet.

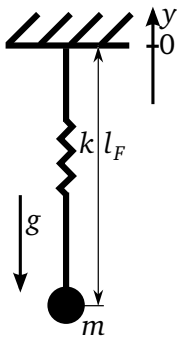

Abbildung 4.7: Aufbau des Federpendels.

4.3.2 Theorie

Die Bewegungsgleichung eines idealen Federpendels lautet im Fall ohne Reibung

$$\ddot{y}m = -mg - k(y + l_F). \tag{4.1}$$

Für den Fall mit Reibung muss der Term $-\mu\dot{y}$ zur rechten Seite der Gleichung addiert werden. Setzt man $\ddot{y} = \dot{y} = 0$, erhält man die Ruhelage. Für beide Fälle erhält man

$$y_0 = -l_F - \frac{mg}{k}. \tag{4.2}$$

Das reibungsfreie Federpendel oszilliert beliebig lange um die Ruhelage. Mit Reibung nimmt die Amplitude der Schwingung ab, bis die Ruhelage eingenommen wird.

4.3.3 Modellierung

Für die Feder wird die Länge $l_F = 1$ m und die Federkonstante $k = 10$ N/m angenommen. Die Punktmasse hat, wie beim mathematischen Pendel, eine Masse von $m = 1$ kg. Die Erdbeschleunigung wirkt in die negative y-Richtung und hat den Wert $g = 9{,}81$ m/s^2. Im gedämpften System wird ein Reibungskoeffizient von $\mu = 1$ Ns/m verwendet. Die folgenden Anfangsbedingungen werden verwendet:

- Abstand der Punktmasse von der Befestigung der Feder $y = -l_F = -1$ m,
- Geschwindigkeit $\dot{y} = 0$ m/s und
- Beschleunigung $\ddot{y} = 0$ m/s^2.

Die vorgenommene Modellierung ist, wie bei den bisher erstellten Modellen, auf den zweidimensionalen Fall beschränkt.

SimMechanics:

Zur Modellierung eines Federpendels in SimMechanics kann z.B. ein Modul namens *Custom Joint* verwenden werden, mit dem sich beliebige Kombinationen aus rotatorischen und translatorischen Freiheitsgraden erzeugen lassen. Mit ihm lässt sich der Starrkörper translatorisch frei an einem Anker befestigen (siehe Abbildung 4.8). An einem zweiten Anker ist eine Feder/Dämpfer-Kombination (*Body Spring & Damper*) angebracht, die mit dem gleichen Starrkörper verbunden ist.

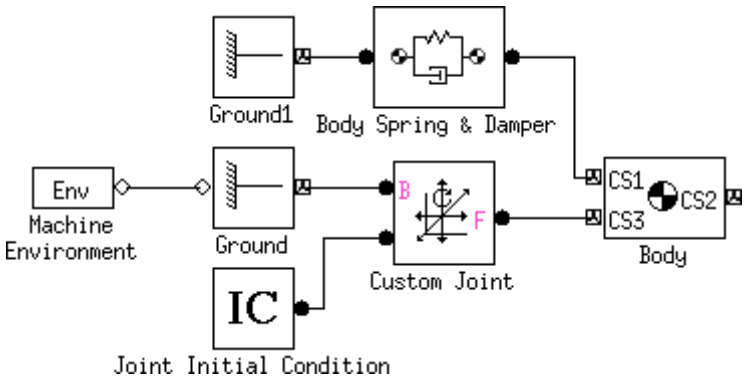

Abbildung 4.8: Modell des Federpendels in SimMechanics.

mbslib:

Die Modellierung eines Federpendels in mbslib ist kompliziert. Dies liegt zum einen an der Art der Modellierung von Federn und zum anderen an der von Mehrkörpersystemen (als Baum). Federn sind in mbslib Linienzüge entlang von Endpunkten, die sich innerhalb der Baumstruktur befinden. Die Feder selbst ist nicht Teil der Baumstruktur. Zudem sind in mbslib zwei Typen von Federn vorhanden: `LinearSpringModel` und `LinearSpringWithRopeModel`. `LinearSpringModel` ist eine ideale ausdehnungslose Feder. `LinearSpringWithRopeModel` ist eine Feder, die an einem massenlosen Seil befestigt ist. Sie übt nur eine Kraft aus, wenn sie über die Länge des Seiles hinaus ausgedehnt wird; sie kann sich also nur zusammenziehen. Es wäre einfach, eine eigene Feder, die in Ruhe eine Länge ungleich Null hat, zu erstellen, aber dieses Beispiel soll auch als Testfall für die vorgefertigte Feder dienen. Wie man über Vererbung eine eigene Komponente eines Mehrkörpersystems erzeugt, wird in Abschnitt 4.4 am Beispiel des Bodenkontaktes erläutert. Im Folgenden wird gezeigt, wie man diese Beschränkungen mit geschickter Modellierung umgehen kann:

Damit die Feder eine Länge von l_F bekommt, wird eine Feder ohne Länge an einem festen Verschiebungs-Element (`FixedTranslation`) der Länge l_F befestigt. Das Hauptproblem ist somit, die

Punktmasse in die Baumstruktur einzubinden. Sie kann nicht frei existieren. Um sie, im Kontext des Modells, frei zu modellieren, wird sie an einem reibungsfreien Schubgelenk (PrismaticJoint) befestigt. Um die benötigten zwei Freiheitsgrade für eine realistische Simulation zu erhalten, wird das Schubgelenk an einem reibungsfreien Drehgelenk (RevoluteJoint) angebracht. Da das Verschiebungs-Element, das die Länge der Feder erzeugt, nicht biegbar ist, muss es ebenfalls am Drehgelenk befestigt werden. Dies geschieht mit einer Verzweigung (Fork). Die vollständige Baumstruktur ist in Abbildung 4.9 dargestellt. Der resultierende Quelltext kann in Anhang 5.2 eingesehen werden.

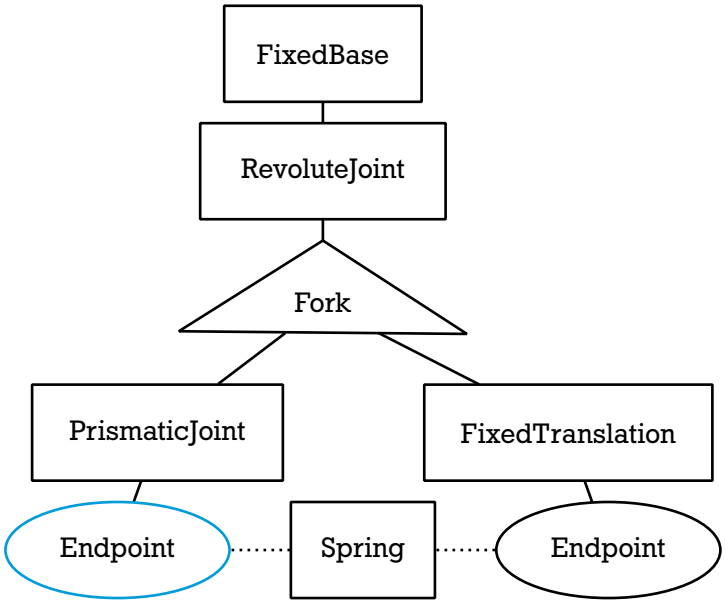

Abbildung 4.9: Baumstruktur des Federpendels (mbslib).

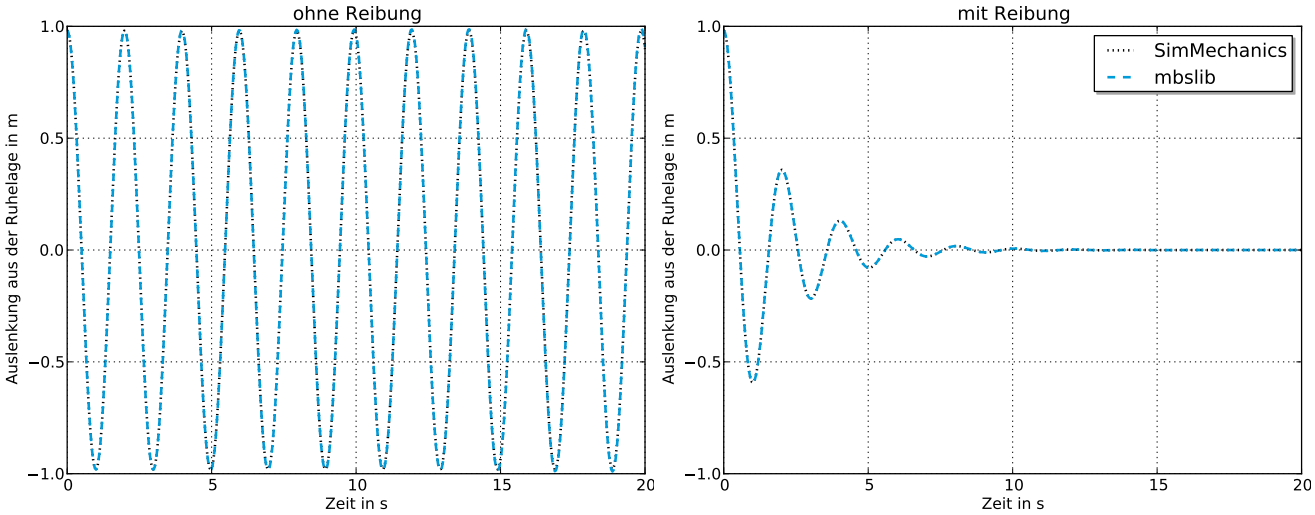

Abbildung 4.10: Auslenkung des Federpendels aus der Ruhelage. Links: ungedämpft. Rechts: gedämpft.

4.3.4 Ergebnisse der Simulation

Setzt man die verwendeten Werte in Gleichung 4.2 ein, erhält man die Ruhelage $y_0 = -1{,}981$ m. In Abbildung 4.10 sind links die Simulationsergebnisse aus mbslib und SimMechanics für das ungedämpfte Federpendel dargestellt. Es ist die Auslenkung aus der Ruhelage (also $y - y_0$) über der Zeit aufgetragen.

Die Ergebnisse aus SimMechanics sind als schwarz gepunktete Linie und die aus mbslib als blau gestrichelte Linie eingezeichnet. Man kann erkennen, dass die Punktmassen mit konstanter Amplitude um die berechnete Ruhelage oszillieren. Analog dazu sind in Abbildung 4.10 rechts die Simulationsergebnisse mit Dämpfung abgebildet. Die Punktmassen oszillieren wieder um die Ruhelage. Die Amplitude ihrer Schwingungen nimmt schnell ab. Sie kommen, wie erwartet, in der berechneten Ruhelage zur Ruhe (die Auslenkung geht gegen 0).

4.4 Bodenkontakt

Zur Simulation der Dynamik des Hinterbeins wird eine Modellierung der Wechselwirkung zwischen Bein und Boden benötigt. Da diese Wechselwirkung in beiden Frameworks nicht vorhanden ist, bietet es sich an, die Erstellung von eigenen Komponenten in beiden Frameworks an diesem Beispiel zu vergleichen.

4.4.1 Aufbau

Die Wechselwirkung zwischen Bein und Boden wird über einen punktförmigen Bodenkontakt vorgenommen. Der Boden selbst wird stark vereinfacht als gedämpfte Feder in y-Richtung modelliert. Der Boden übt nur eine Kraft aus, wenn der Kontaktpunkt in ihn eindringt. Diese Modellierung ist in Anbetracht der Problemstellung ausreichend.

Testaufbau 1: Freie Punktmasse

Zur Kontrolle der korrekten Funktion wird eine Punktmasse mit einer Masse von 1 kg an einem reibungsfreien Schubgelenk befestigt. An der Punktmasse wird der Bodenkontakt angebracht. Der Boden befindet sich 1,0 m unter der Anfangsposition der Masse.

Testaufbau 2: Pendel

Als weiterer Testfall wird eine ein Meter lange Stange mit einem Radius von 5 cm und einer Masse von 1 kg an einem Drehgelenk 0,5 m über dem Boden befestigt. Zu Beginn der Simulation befindet sich die Stange parallel zum Boden.

4.4.2 Theorie

Durch den beschriebenen Aufbau wirkt die folgende Kraft auf den Kontaktpunkt:

$$F(y) = \ddot{y}m = \begin{cases} 0, & y > y_B \\ -k_B * (y + y_F) - \dot{y}\mu_B, & y \leq y_B \end{cases} \tag{4.3}$$

mit der Position des Bodens y_B, der Federkonstante k_B und der Dämpfung μ_B.

Analog zu Gleichung 4.2 ist zu erwarten, dass die Ruhelage des Kontaktpunkts unterhalb der Höhe des Bodens liegt. Der Punkt dringt also in den Boden ein. Die Eindringtiefe ist abhängig von den Kräften, die auf den Kontaktpunkt wirken. Für die Testaufbauten mit den Werten $\mu_B = 100$ Ns/m und $k_B = 5000$ N/m erhält man eine Abweichung von ca. 2 mm.

4.4.3 Modellierung

SimMechanics:

Zur Modellierung in SimMechanics wird ein Untermodul erstellt (siehe Abbildung 4.11). Das Modul hat eine Verbindung nach außen, die mit dem Kontaktpunkt verbunden wird. Über einen Einstellungsdialog können die verwendeten Konstanten (Position, Federkonstante und Dämpfung) modifiziert werden.

Über einen Starrkörper-Sensor wird die Position und Geschwindigkeit des Kontaktpunktes ausgelesen. Aus der Position wird mit Hilfe der Positions-Konstante die Eindringtiefe berechnet. Die Eindringtiefe wird mit der Federkonstanten multipliziert. Davon wird das Produkt aus Geschwindigkeit und Dämpfung subtrahiert. Falls die Eindringtiefe positiv ist, wird das Produkt des Ergebnisses mit dem Einheitsvektor in y-Richtung als Kraft auf den Kontaktpunkt ausgeübt. Dazu wird ein Starrkörper-Aktuator verwendet.

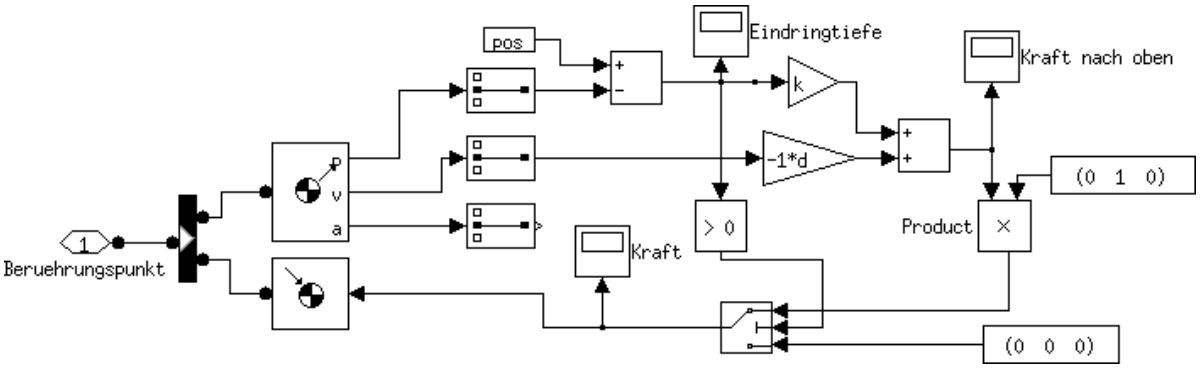

Abbildung 4.11: Modellierung des Bodenkontakts in SimMechanics.

mbslib:

Zur Modellierung des Bodenkontakts in mbslib wird die Klasse GroundContact erstellt, die von der Klasse IForceGenerator erbt. Dazu müssen die Methoden applyForce(), resetForce(), getName() und getIntegrator() implementiert werden. Von diesen Methoden ist nur die erste für unseren Vergleich interessant, deshalb wird nur sie im Folgenden (siehe Listing 4.1) erläutert. Der vollständige Quelltext ist in Anhang 5.3 und 5.4 einzusehen.

Listing 4.1: Ausschnitt aus GroundContact.cpp (siehe Listing 5.4 im Anhang).

```
47 void GroundContact::applyForce()
48 {
49     TScalar depth, speed, springforce;
50
51     depth = direction * (point.getCoordinateFrame().r.y() - position);
52     speed = point.getCoordinateFrame().v.y();
53     springforce = depth * springConstant - (speed * dampingConstant);
54
55     force = 0;
56     // condassign fuer Kompatibilitaet mit ADOC_C
57     // codassign (a, b, c, d) entspricht a = (b > 0) ? c : d;
58     condassign(force, depth, force + springforce, force);
59
60     point.setExternalForceTorque(TVector3(0, force, 0),
61                                  TVector3::Zero());
62 }
```

In den Zeilen 51 bis 53 wird zunächst die Eindringtiefe berechnet, danach wird die Geschwindigkeit des Kontaktpunktes ausgelesen. Mit diesen Werten wird die von der Feder ausgeübte Kraft berechnet. Die Variable direction legt die Richtung fest, in welche die Abstoßung wirkt. -1 entspricht einer Abstoßung nach oben und $+1$ einer Abstoßung nach unten. Die unkonventionelle Konstruktion der Zeilen 55 bis 58 wird benötigt, damit die Kraft mittels ADOL-C differenziert werden kann. Beim Kompilieren ohne ADOL-C wird condassign automatisch durch den im Kommentar angegebenen Ausdruck ersetzt. Zuletzt wird in den Zeilen 60 und 61 die berechnete Kraft auf den verbundenen Endpunkt ausgeübt.

Zur Anwendung des Bodenkontakts muss eine Instanz erzeugt werden, die mit dem Befehl addForceGenerator zum Mehrkörpersystem hinzugefügt wird. Der folgende Quelltext (siehe Listing 4.2) stammt aus dem Modell von Testaufbau 1 (vollständiger Quelltext in Listing 5.5 im Anhang). Der Boden ist bei $y_B = -1{,}0$ m und die Abstoßung wirkt nach oben. Die Federkonstante hat einen Wert von $k_B = 5000$ N/m und die Dämpfung beträgt $\mu_B = 100$ Ns/m.

Listing 4.2: Ausschnitt aus bodenkontakt_mp.cpp (siehe Listing 5.5 im Anhang).

```
56      mbs.addForceGenerator(*gc);
```

4.4.4 Ergebnisse der Simulation

In Abbildung 4.12 sind die Simulationsergebnisse der Testfälle dargestellt. Links ist der erste Testaufbau abgebildet. Man kann erkennen, dass die Ergebnisse aus SimMechanics (gepunktete schwarze Linie) und mbslib (gestrichelte blaue Linie) gleich aussehen, es ist keine signifikante Abweichung feststellbar. Die Punktmasse fällt aus der Höhe von 1 m, dringt dann leicht in den Boden ein, springt eine ganz kleine Strecke über den Boden und kommt dann, nach ca. 0,5 s, knapp unter der Bodenkante zur Ruhe. Beim zweiten Testaufbau (rechts) bewegt sich der Kontaktpunkt aus der initialen Höhe von 0,5 m auf den Boden zu. Man kann erkennen, dass die Ergebnisse aus SimMechanics und mbslib in Bodennähe voneinander abweichen (siehe Vergrößerung). Diese Abweichung wird durch Unterschiede der Integratoren verursacht.

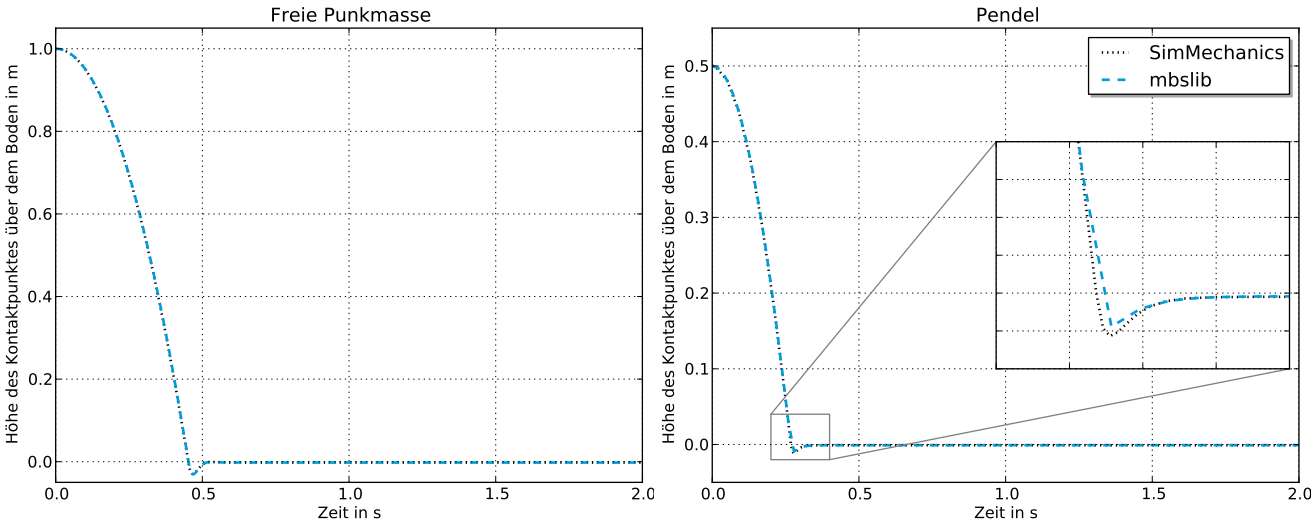

Abbildung 4.12: Höhe des Kontaktpunktes über dem Boden.

4.5 Dreifachpendel

Ein Dreifachpendel ist ein einfaches und trotzdem eindrucksvolles Beispiel für ein chaotisches System. Ein Simulationsframework, dessen Simulation realistisch sein soll, muss chaotisches Verhalten korrekt simulieren können, da sonst die korrekte Simulation komplexer Systeme nicht möglich ist. In der Realität zeigen komplexe mechanische Systeme mit einer nichtlinearen Komponente in der Bewegungsgleichung chaotisches Verhalten[12].

4.5.1 Aufbau

Das Dreifachpendel besteht aus drei starren, massiven Stangen, die mit reibungsfreien Drehgelenken verbunden sind (siehe Abbildung 4.13). Die erste Stange ist über ein Drehgelenk (blauer Kreis) fest verankert. Jeder der Stangen hat eine Masse von $m = 1$ kg, eine Länge von $l = 1$ m und einen Radius von $r = 0{,}05$ m. Der Trägheitstensor entspricht einem Zylinder. Der Schwerpunkt der Stangen ist jeweils in ihrer Mitte. Zu Beginn des Experiments werden die Stangen so angehoben, dass sie (fast) senkrecht nach oben stehen. Die Abweichungen der Winkel der Drehgelenke aus der senkrechten Position werden mit φ_1, φ_2 und φ_3 bezeichnet.

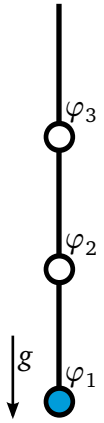

Abbildung 4.13: Aufbau des Dreifachpendels. Das blau markierte Drehgelenk ist verankert.

4.5.2 Theorie

Ein chaotisches System hat die Eigenschaft, dass minimale Änderungen der Anfangsbedingungen sehr starke Auswirkungen auf das Verhalten des Systems haben. Für das Dreifachpendel ist zu erwarten, dass pseudozufällige Variationen der Initialen φ_i mit $i \in \{1, 2, 3\}$ mit Werten zwischen 0 rad und 10^{-8} rad zu stark abweichenden Positionen nach Ende der Simulation der Dynamik führen.

4.5.3 Modellierung

Die Modellierung wird nicht im Detail erläutert, weil sie, außer dem Setzen der Trägheitstensoren kaum Unterschiede zum mathematischen Pendel aufweist.

Da die Bewegung eines Dreifachpendels sich nicht befriedigend in ein statisches Diagramm auftragen lässt, wird die folgende Alternative verwendet: Mit einem Python-Skript (siehe Listing 5.6) wurden 100 Sätze von jeweils drei Gelenkwinkeln generiert. Die Simulation wird für jeden dieser Sätze aufgerufen. Danach werden die Positionen des Pendels nach 20 Sekunden Simulation verglichen.

Im anfangs erwähnten Subversion-Projektarchiv (siehe Abschnitt 1.3) sind Beispielvideos der Dynamik zu finden.

Listing 4.3: makeInertiaTensorCylinderX.m

```
1 function I = makeInertiaTensorCylinderX(radius, length, mass)
2   I1 = mass * radius * radius / 4.0 + mass * length * length / 12.0;
3   I2 = 0.5 * mass * radius * radius;
4
5   I = diag([I2, I1, I1]);
6 end
```

SimMechanics

Zur Berechnung des Trägheitstensors wird eine Funktion der mbslib verwendet, die nach Matlab portiert wurde (siehe Listing 4.3). Die Funktion hätte mit Hilfe einer Formelsammlung neu erstellt werden können, allerdings ist eine Portierung schneller und bietet die Bequemlichkeit der gleichen Signatur.

Die Variation von Parametern eines SimMechanics-Modells aus Matlab ist umständlicher als erwartet: Um Konstanten des Modells aus Matlab ändern zu können, wird das komplette Modell in ein Untermodul (in diesem Beispiel *sub* genannt) verschoben. Die Konstanten, die gesetzt werden sollen, werden zur Eingabemaske hinzugefügt. Dies erlaubt es, die Werte an allen Stellen, an denen sie verwendet werden, gleichzeitig zu ändern. In Listing 4.4 ist die zur Variation verwendete Funktion angegeben. Sie bekommt die drei Winkel als Parameter übergeben und gibt die Position der Gelenke und des Endpunktes des letzten Starrkörpers zur Visualisierung zurück. Zuerst wird das Modell des Dreifachpendels geladen. Danach werden die drei Winkel in der Maske des Untermoduls gesetzt. Alle Werte der Maske müssen Strings sein. Deshalb werden die Winkel jeweils mit num2str in Strings umgewandelt. In Zeile 8 wird die Simulation gestartet. Da es nicht möglich ist, auf Ergebnisse der Simulation direkt zuzugreifen[1], schreibt das Modell die Simulations-Ergebnisse in eine Datei. Diese Datei wird in Zeile 10 wieder eingelesen. In Zeile 11 wird die Endposition zurückgegeben.

Listing 4.4: dreifachpendel_20s.m

```
1  function p=dreifachpendel_20s(q1, q2, q3)
2    load_system('dreifachpendel')
3
4    set_param('dreifachpendel/sub', 'q1', num2str(q1))
5    set_param('dreifachpendel/sub', 'q2', num2str(q2))
6    set_param('dreifachpendel/sub', 'q3', num2str(q3))
7
8    sim('dreifachpendel')
9
10   p = load('dreifachpendel_plot.mat', '-mat', 'ans')
11   p = p.ans(:,end)
12 end
```

Die vom Python-Skript generierten Parametersätze werden eingelesen (siehe Listing 4.5). Für jeden Satz wird die Funktion einmal aufgerufen und das Ergebnis gespeichert.

Listing 4.5: dreifachpendel_run.m

```
1  [a,b,c,d] = textread('dreifachpendel.rnd', '%f%f%f%f');
2
3  runs = 100;
4  res = zeros(runs, 8);
5
6  for i = 1:runs
7    res(i, 2:8) = dreifachpendel_20s(b(i), c(i), d(i))';
8    res(i, 2) = 0;
9  end
10
11 save -ascii dreifachpendel_run2.dat res
```

Damit die wiederholte Simulation mit verschiedenen Parametern möglichst schnell geschieht, wird der *Accelerator*-Modus verwendet. Dadurch wird das Modell nicht normal interpretiert, sondern vor der Ausführung kompiliert, was bei wiederholter Ausführung zu einer großen Zeitersparnis führt.

[1] Dies ist ein Ergebnis eigener Untersuchungen und wurde von der MathWorks-Hotline am 20.03.2012 bestätigt: Laut Aussage der Hotline kann aus Matlab nur über Dateien oder globale Variablen des Workspaces auf Simulink-Komponenten zugegriffen werden.

mbslib

Das mbslib-Modell ist in Listing 5.7 im Anhang vollständig abgedruckt. Die initialen Positionen der drei Gelenke können per Kommandozeilen-Argument geändert werden. Dies geschieht mit einem weiteren Python-Skript (siehe Listing 5.8): Es liest die mit dem oben beschriebenen Python-Skript generierten Parametersätze ein und ruft das Modell mit den jeweiligen Parametern auf. Aus Geschwindigkeitsgründen werden vier Threads zur Ausführung genutzt. In jedem Thread wird die Simulation mit einem Parametersatz durchgeführt. Die Ergebnisse werden gesammelt und abgespeichert.

4.5.4 Ergebnisse der Simulation

In Abbildung 4.15 ist ein Beispiel für den Verlauf der Gelenkwinkel des Dreifachpendels abgebildet. Die Simulationsergenisse aus SimMechanics sind als schwarze gepunktete Linie und die aus mbslib als schwarz gestrichelte Linie aufgetragen. Die initialen Winkel sind $\varphi_1 = 7{,}71196 \cdot 10^{-09}$ deg, $\varphi_2 = 4{,}65666 \cdot 10^{-09}$ deg und $\varphi_3 = 1{,}59058 \cdot 10^{-09}$ deg. Die Werte aus SimMechanics wurden mit dem Modul *Continous Angle* aus dem intern verwendeten Winkelbereich (zwischen $-180°$ und $180°$) in die kontinuierliche Darstellung integriert. Man kann sehen, dass das äußerste Glied (φ_3, untere Abbildung) sich mehrere Male komplett dreht und dabei mehrfach die Richtung ändert. Zu Beginn der Simulation stimmen die Winkel in SimMechanics und mbslib überein, aber schon nach ca. vier Sekunden fangen sie an sich zu unterscheiden. Bei φ_3 beträgt der Unterschied zeitweise mehr als 20 komplette Umdrehungen.

In Abbildung 4.14 sind die vollständigen Simulationsergebnisse abgebildet. Jeder schwarze Linienzug entspricht den Gelenkstellungen nach 20 Sekunden einer Simulation. Obwohl sich die Positionen in SimMechanics (links) und mbslib (rechts) unterscheiden, können die gleichen Phänomene beobachtet werden: Bei beiden Framworks verursachen kleine Änderungen der Anfangsposition, wie erwartet, starke Änderungen der Endposition. Chaotisches Verhalten kann also in beiden Frameworks simuliert werden.

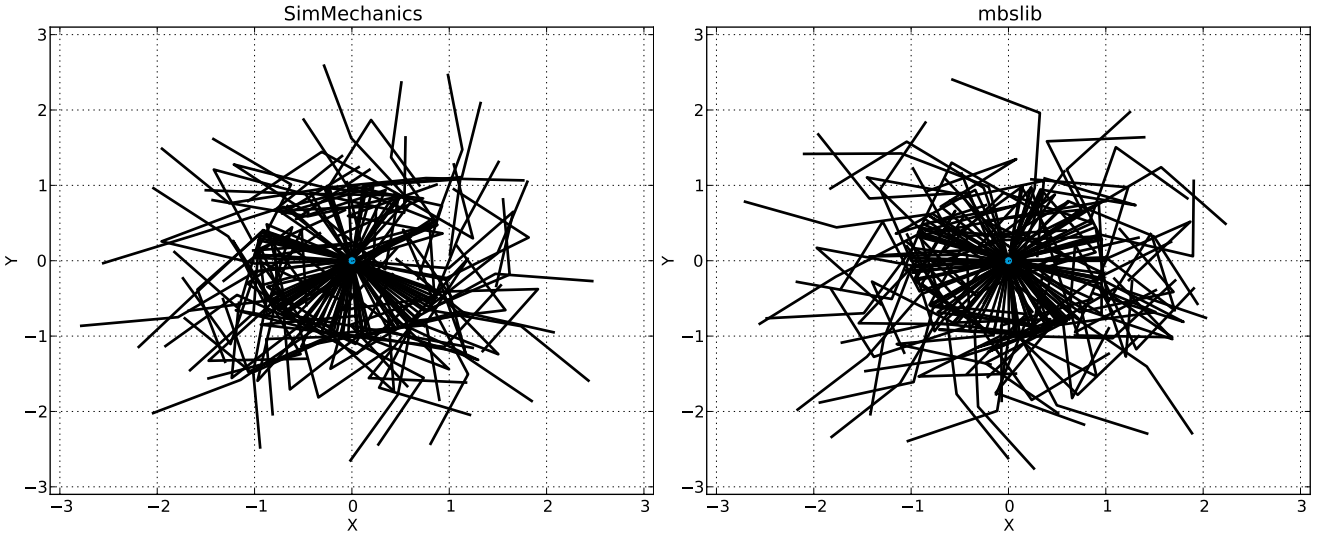

Abbildung 4.14: Positionen des Dreifachpendels nach 20 Sekunden Simulation. Die initialen Winkel sind per Pseudozufallszahlen zwischen 0 rad und 10^{-8} rad 100 Mal variiert.

Da die Endpositionen so sensibel auf Änderungen reagieren, ist klar, dass Unterschiede in Integrator und der Schrittweite der Integration abweichende Ergebnisse verursachen. Der Vergleich der Ergebnisse zwischen den Frameworks kann somit nicht funktionieren.

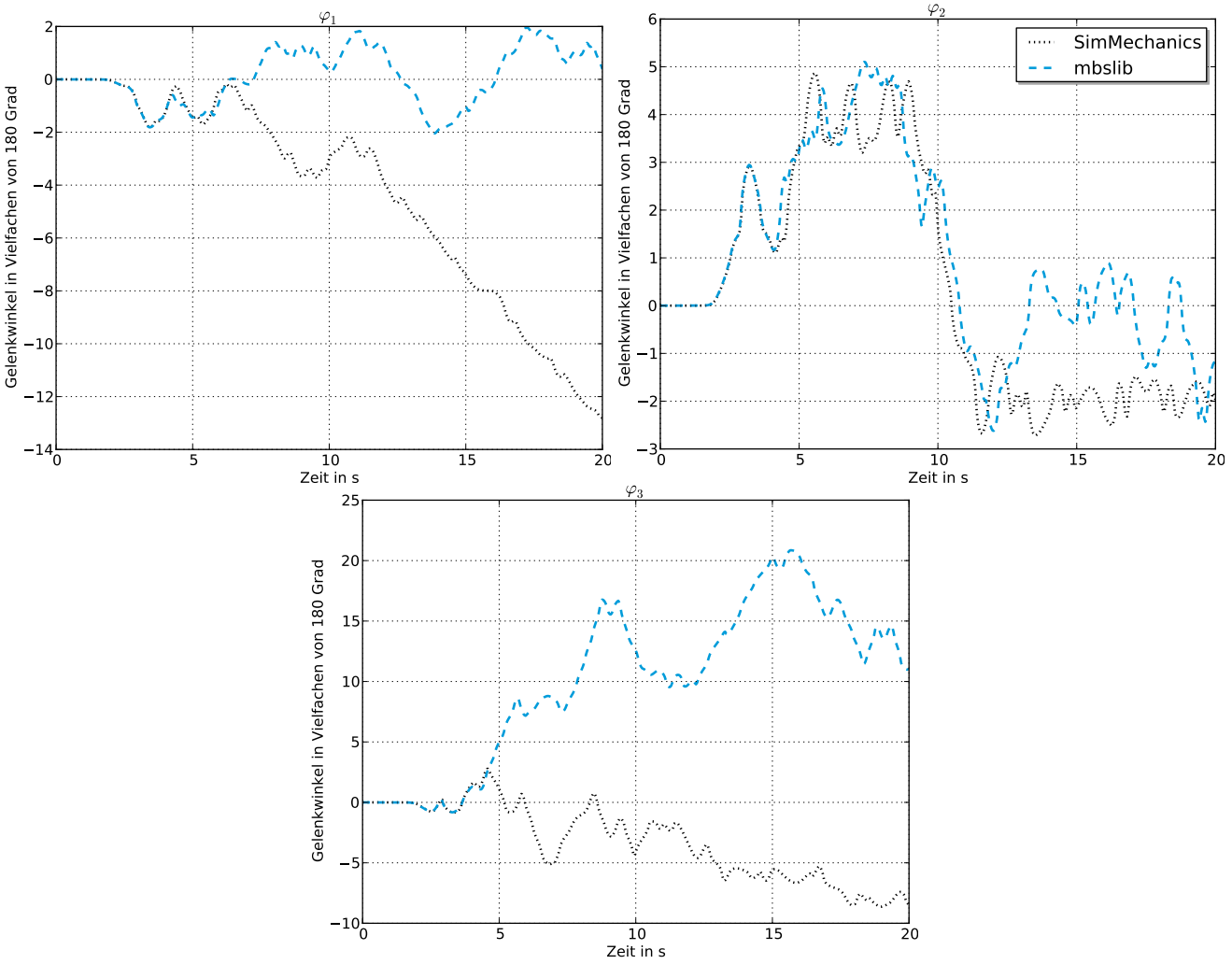

Abbildung 4.15: Verlauf der Winkel des Dreifachpendels innerhalb von 20 Sekunden.

4.6 Hinterbein

Die Modellierung und Untersuchung des bio-inspirierten Roboterbeins ist eines der Ziele dieser Arbeit. In den folgenden Abschnitten werden, wie bei den anderen betrachteten Modellen, erst Aufbau und Implementierung beschrieben und danach die Ergebnisse besprochen.

4.6.1 Aufbau

Die Skelettstruktur des zu modellierenden Roboterbeins (siehe Abbildung 4.16) besteht aus vier Starrkörpern (schwarze Linien). Die Drehgelenke werden durch schwarze oder orangefarbene Kreise dargestellt. Der erste Starrkörper (Os femoris) ist an einem Drehgelenk befestigt. An ihm sind zwei weitere Drehgelenke angebracht. An diesen sind zwei parallele, gleich lange Starrkörper (Tendo achillis oben und Fibula/Tibia unten) befestigt. An ihren Enden befinden sich weitere Drehgelenke, die sie mit dem letzten Starrkörper (Ossa metatarsalia) verbinden. Am Os femoris ist der Endpunkt des Gluteus (rechte blaue Linie) befestigt. Der Gluteus läuft von diesem Endpunkt aus über eine Umlenkrolle (blauer Kreis) zu seinem Anfangspunkt. An dem Drehgelenk zwischen Os femoris und Sartorius ist der Endpunkt des Sartorius (linke blaue Linie) angebracht. Dort befindet sich auch die Umlenkrolle (orangefarbener Kreis) des Versus lateralis (linke orangefarbene Linie), der an Fibula/Tibia endet. Am Gelenk zwischen Tendo achillis und Ossa metatarsalia befindet sich der Endpunkt des Biceps femoris (rechte orangefarbene Linie).

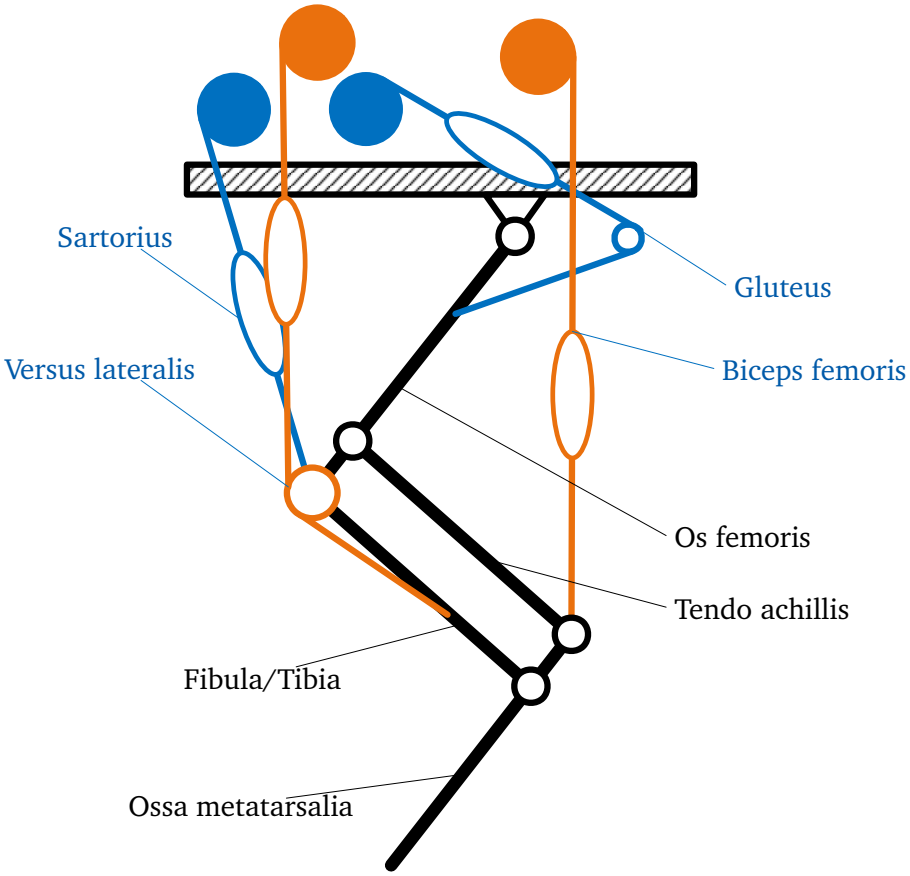

Abbildung 4.16: Aufbau des Hinterbeins (Seitenansicht: links ist vorne).

Diese vier Muskeln (Gluteus, Sartorius, Versus lateralis und Biceps femoris) werden durch eine Kombination aus einer Feder und einem Seil realisiert (im Folgenden Seilzug genannt). Ihre Anfangspunkte (große, ausgefüllte Kreise) liegen über dem ersten Drehgelenk. Über die Feder-/Seil-Kombination werden Zugkräfte im Bein erzeugt. Eine Kraft wirkt nur, wenn das entsprechende Seil gespannt ist, also wenn die aktuelle Länge der Seil-/Feder-Kombination größer als die Länge des Seils ist.

Nicht nur der Aufbau, sondern auch die Aktuierung ist dem biologischen Vorbild nachempfunden: Die Starrkörper werden, wie zuvor angedeutet, nicht über Motoren in den Drehgelenken, sondern über die Muskeln bzw. Seilzüge bewegt. Um das Steuerungsprinzip zu verstehen, ist es hilfreich, die Seilzüge in Paare von Gegenspielern zu unterteilen. Diese Unterteilung wird in der Skizze des Aufbaus durch die farbige Markierung angedeutet. Durch Anziehen von Sartorius oder Gluteus (blau markiert) kann der Os femoris nach links bzw. rechts (also vor bzw. zurück) bewegt werden. Analog dazu kann durch Anziehen von Versus lateralis oder Biceps femoris das Bein gestreckt bzw. gebeugt werden. Beim Anziehen eines Seilzuges muss sein Gegenspieler gelockert werden.

4.6.2 Parametrisierung

Zur einfachen Untersuchung der Eigenschaften des Beins müssen alle wichtigen Parameter leicht veränderbar sein. Dazu werden ihre Werte in eine Vielzahl von Variablen ausgelagert (siehe Listing 5.10). In Tabelle 4.1 werden alle Parameter beschrieben. Variablen, die Längen und Positionen definieren, sind zudem in Abbildung 4.17 in die Skizze des Hinterbeins eingetragen. Die Längen, die in Blau eingetragen sind, werden in Vielfachen der Länge des Starrkörpers, an dem sie aufgetragen sind, ausgedrückt.

Der grüne Pfeil in der Abbildung markiert die Stelle, an der zur Modellierung als offene kinematische Kette eine Feder verwendet wird. Dies wird im folgenden Abschnitt weiter erläutert.

Variable	Beschreibung
friction	Reibungskoeffizient der Drehgelenke.
muscle_constant	Federkonstante der Muskeln.
muscle_damping	Dämpfung der Muskeln.
radius	Radius der Starrkörper zur Berechnung der Trägheitstensoren.
ground_constant	Federkonstante des Bodens.
ground_damping	Dämpfung des Bodens.
ground_pos	Position des Bodens.
os_femoris_length	Länge des Os femoris.
os_femoris_mass	Masse des Os femoris.
os_femoris_com	Schwerpunkt des Os femoris. Angegeben in Vielfachen der Länge des Starrkörpers (vom obersten Drehgelenk betrachtet).
os_femoris_gluteus	Position des Endes des Gluteus an Os femoris in Vielfachen der Länge des Os femoris.
fibula_tendo_achillis	Abstand von Fibula/Tibia und Tendo achillis in Vielfachen der Länge des Os femoris.
fibula_length	Länge der Fibula/Tibia und Tendo achillis.
fibula_mass	Masse von Fibula/Tibia.
fibula_com	Schwerpunkt der Fibula/Tibia. Angegeben in Vielfachen der Länge des Starrkörpers (von Os femoris aus betrachtet).
fibula_versus_lateralis	Abstand des Endpunktes des Versus lateralis an Fibula/Tibia in Vielfachen der Länge von Fibula/Tibia in Prozent (auf Seite des Ossa metatarsalia).
tendo_achillis_constant	Federkonstante der Feder zur Verbindung von Tendo achillis und Ossa metatarsalia.
tendo_achillis_damping	Dämpfung der Feder zur Verbindung von Tendo achillis und Ossa metatarsalia.
tendo_achillis_mass	Masse des Tendo achillis.
tendo_achillis_com	Schwerpunkt des Tendo achillis. Angegeben in Vielfachen der Länge des Starrkörpers (von Os femoris betrachtet).
ossa_metatarsalia_length	Länge der Ossa metatarsalia.
ossa_metatarsalia_mass	Masse der Ossa metatarsalia.
ossa_metatarsalia_com	Schwerpunkt der Ossa metatarsalia. Angegeben in Vielfachen der Länge des Starrkörpers (von Tendo achillis betrachtet).
sartorius_length	Initiale Länge des Sartorius.
sartorius_anfang	Anfangsposition des Sartorius in kartesischen Koordinaten.
versus_lateralis_length	Initiale Länge des Versus lateralis.
versus_lateralis_anfang	Anfangsposition des Versus lateralis in kart. Koord.
versus_lateralis_pulley_size	Größe der Umlenkrolle des Versus lateralis. Entspricht ungefähr dem $1/\sqrt{2}$-fachen Radius.
gluteus_length	Initiale Länge des Gluteus.
gluteus_anfang	Anfangsposition des Gluteus in kart. Koord.
gluteus_rolle	Position der Umlenkrolle des Gluteus in kart. Koord.
biceps_femoris_length	Initiale Länge des Biceps femoris.
biceps_femoris_anfang	Anfangsposition des Biceps femoris in kart. Koord.

Tabelle 4.1: Vollständige Parametrisierung des Hinterbeins.

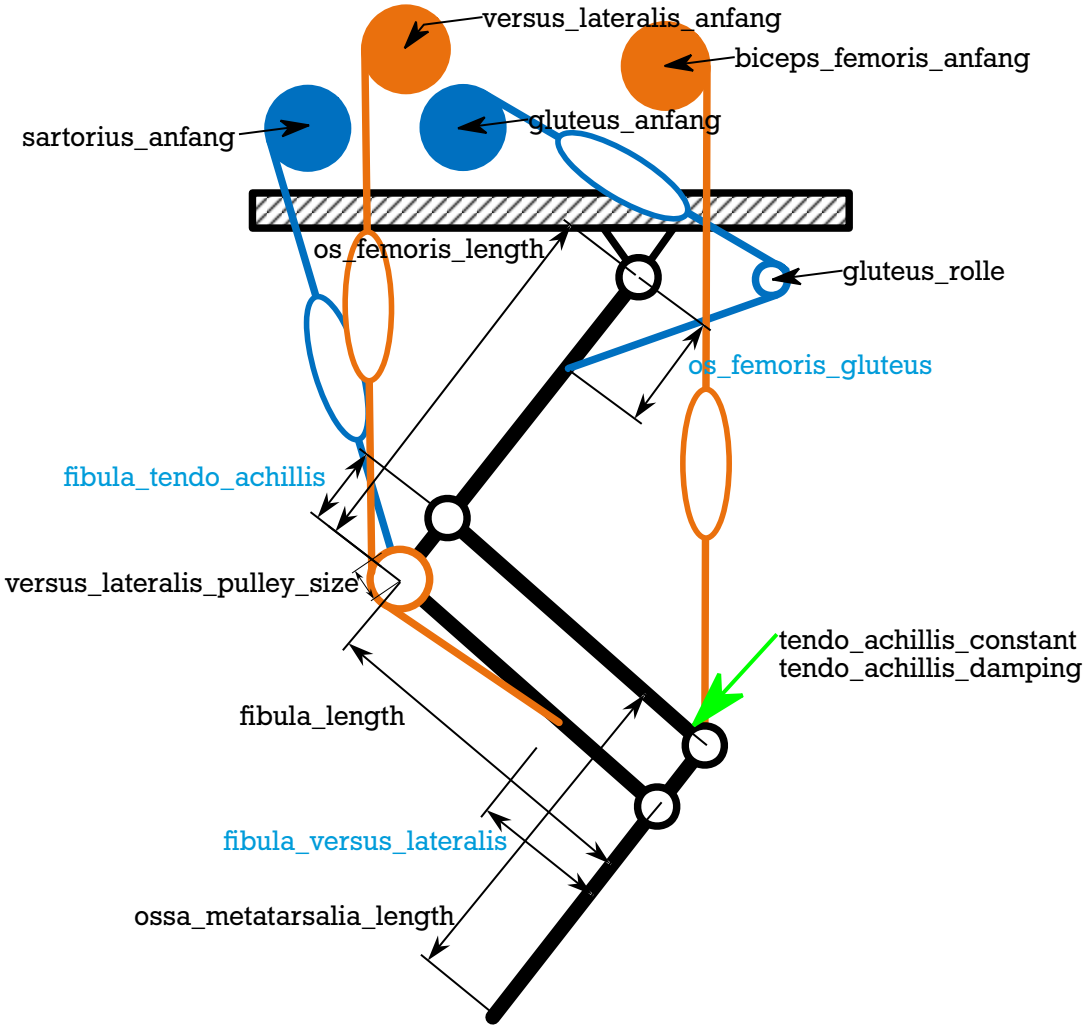

Abbildung 4.17: Parametrisierung des Hinterbeins.

4.6.3 Modellierung (als offene kinematische Kette)

Zur Modellierung des Hinterbeins als offene kinematische Kette wird die Verbindung zwischen Tendo achillis und Ossa metatarsalia (grüner Pfeil in Abbildung 4.17) durch eine Feder ersetzt. Dabei wird das eingezeichnete Drehgelenk weggelassen.

SimMechanics:

Der Aufbau des SimMechanics-Modells ist in Abbildung 4.18 dargestellt. Die Untermodule R_1 bis R_4 sind gedämpfte Drehgelenke; sie sind wie beim gedämpften mathematischen Pendel (siehe Abbildung 4.5) aufgebaut. Der Bodenkontakt entspricht dem zuvor entwickelten Modul (siehe Abschnitt 4.4). Die Kette aus Anker, R_1, Os femoris, R_2, Fibula/Tibia, R_3 und Ossa metatarsalia entspricht genau dem Aufbau in Abbildung 4.16. An den entsprechenden Stellen an Os femoris greifen die Seilzüge (rote Rechtecke) Sartorius und Gluteus an. Der Versus lateralis ist mit dem Os fermoris und Fibula/Tibia verbunden. Von Os femoris bezieht er die Position der Umlenkrolle. Auf Fibula/Tibia kann er Kraft ausüben. Der Biceps femoris ist mit der Ossa metatarsalia verbunden, die er bewegen kann. Der Bodenkontakt ist an der Ossa metatarsalia angebracht. Zwischen Os femoris und Ossa metatarsalia befindet sich die Kette aus R_4, dem Starrkörper „Tendo achillis rb“ und dem Feder-Dämpfer-Modul „Tendo achillis“. Die Module „plot line1“ und „plot line2“ protokollieren die Position der Starrkörper, damit man sie mit dem externen Visualisierungs-Programm verwenden kann.

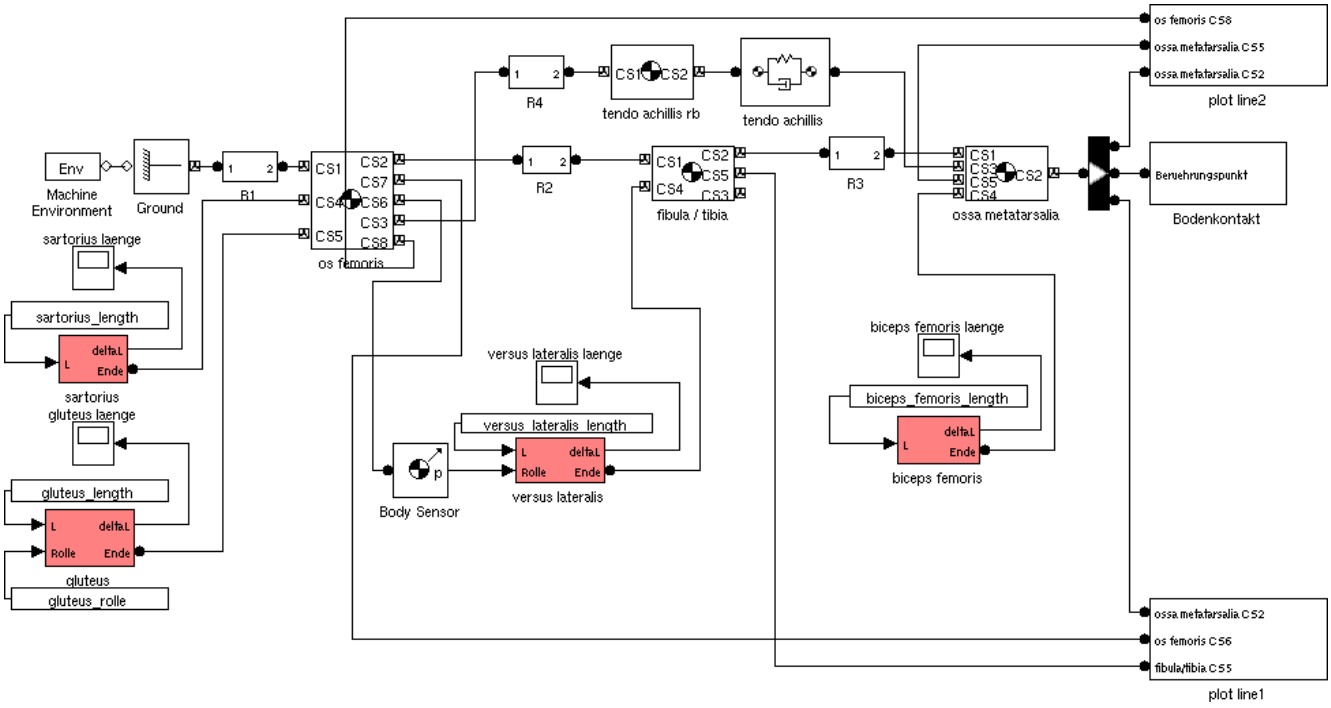

Abbildung 4.18: Modellierung des Hinterbeins in SimMechanics.

Die Seilzüge Sartorius und Biceps femoris berechnen den Abstand zwischen Anfangs- und Endpunkt. Ist dieser Abstand größer als die eingestellte Länge, wirkt eine Kraft, die analog zu einer gedämpften Feder berechnet wird, in Richtung des Verbindungsvektors zwischen Endpunkt und Anfangspunkt. Die beiden anderen Seilzüge besitzen zusätzlich eine Umlenkrolle. Hier wird die Strecke entlang des Linienzuges (vom Anfang zur Rolle und anschließend zum Ende) berechnet. Die Kraft wirkt hier vom Endpunkt zur Umlenkrolle. Alle Linienzüge protokollieren die Positionen der verwendeteten Punkte zur Visualisierung. Die Umlenkrolle des Versus lateralis wird durch einen diagonal vom Os fermoris jeweils um versus_lateralis_pulley_size nach unten links verschobenen Punkt approximiert.

Die initialen Gelenkstellungen sind so gewählt, dass Os fermoris senkrecht zum Boden steht. Die anderen Glieder stehen jeweils senkrecht zueinander. Der Boden befindet sich bei $y = -2{,}2$ m.

mbslib:

Die Modellierung des Hinterbeins in mbslib beinhaltet keine neuen Komponenten, deshalb wird das Modell nur kurz besprochen: Zuerst werden die Trägheitstensoren der Starrkörper berechnet. Danach wird der Anker erzeugt und mittles Verzweigungen und fester Verschiebungen die Anfangspunkte aller Seilzüge erstellt. Danach werden die Drehgelenke und Knochen erzeugt. Hierbei werden Endpunkte für die Feder zur Verbindung zwischen Tendo achillis und Ossa metatarsalia erzeugt. Ein Endpunkt am Ende von Ossa metatarsalia wird mit einer Instanz eines Bodenkontaktes (siehe Abschnitt 4.4) verbunden. Zuletzt werden die Feder zur Verbindung von Tendo achillis mit Ossa metatarsalia und die vier Seilzüge erstellt. Zur Modellierung der Seilzüge wird das beim Federpendel (siehe Abschnitt 4.3.3) beschriebene fnLinearSpringWithRopeModel verwendet.

Das Modell des Hinterbeins wird in mbslib in einer eigenen Klasse (hinterbein_modell) generiert. Dies erleichtert die Benutzung des Modells in verschiedenen Simulationen. Der Quelltext dieser Klasse ist in Listing 5.11 und Listing 5.12 im Anhang aufgeführt. Die Klasse hat, zusätzlich zum Konstruktor, drei Methoden: printCoordinateLines(float t), printLineSettings(), printMuscleLengths().

printCoordinateLines(float t) gibt die aktuelle Position geeigneter Punkte des Beins zur Visualisierung mit dem Visualisierungs-Programm auf der Konsole aus. Dabei wird die Zeit t angegeben. printLineSettings() gibt die Grafik-Optionen der Linienzüge für das Visualisierungs-Programm aus.

printMuscleLengths() gibt die tatsächliche geometrische Länge der Seilzüge aus. Der Konstruktor nimmt optional einen Satz an Parametern (Klasse hinterbein_parameter) entgegen. Falls keine Parameter übergeben werden, werden die Standard-Einstellungen verwendet.

Die Klasse hinterbein_parameter (siehe Listing 5.9 und 5.10 im Anhang) speichert alle in Tabelle 4.1 angegeben Parameter. Diese Art der Speicherung hat den Vorteil, dass ausgewählte Parameter vor Erstellung des Modells leicht angepasst werden können, ohne dass das Modell geändert werden muss.

Ein Beispiel für die Verwendung des Modells ist in Listing 5.13 im Anhang angegeben. Das Bein wird, wie beim SimMechanics-Modell, aus der initialen Position (Os Femoris und Ossa metatarsalia senkrecht zum Boden und Fibula/Tibia und Tendo Achillis parallel zum Boden) fallen gelassen, bis der Kontaktpunkt den Boden bei $y = -2{,}2$ m trifft.

4.6.4 Modellierung (als geschlossene kinematische Kette)

SimMechanics:

Um aus dem Modell mit offener kinematischer Kette ein Modell mit geschlossener kinematischer Kette zu erzeugen, muss nur das Feder-Dämpfer-Modul „Tendo achillis“ durch ein weiteres Drehgelenkt R_5 ausgetauscht werden. Diese Änderung ist in Abbildung 4.19 durch einen blauen Kasten markiert.

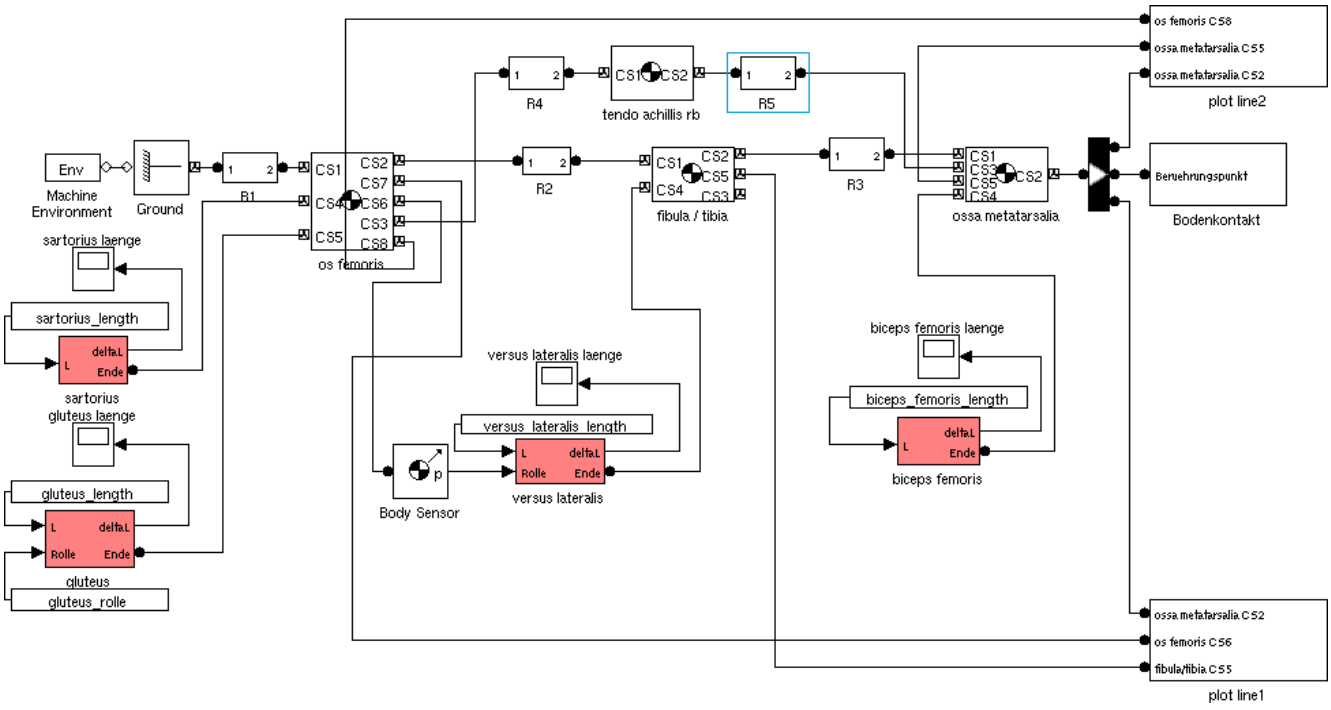

Abbildung 4.19: Modellierung des Hinterbeins in SimMechanics als geschlossene kinematische Kette.

mbslib:

Wie zuvor (siehe Abschnitt 2.1) erwähnt, können in mbslib keine geschlossenen kinematischen Ketten modelliert werden.

4.6.5 Ergebnisse der Simulation

Die Positionen der verschiedenen Modelle des Hinterbeins sind in Abbildung 4.20 abgebildet. Die Starrkörper werden durch durchgezogene schwarze Linien, die Gelenke durch blaue Punkte und die Seilzüge

durch gestrichelte rote Linien dargestellt. Man kann erkennen, dass die Positionen des SimMechanics-Modells mit offener kinematischer Kette (links), des mbslib-Modells mit offener kinematischer Kette (mitte) und des SimMechanics-Modells mit geschlossener kinematischer Kette (rechts) identisch sind.

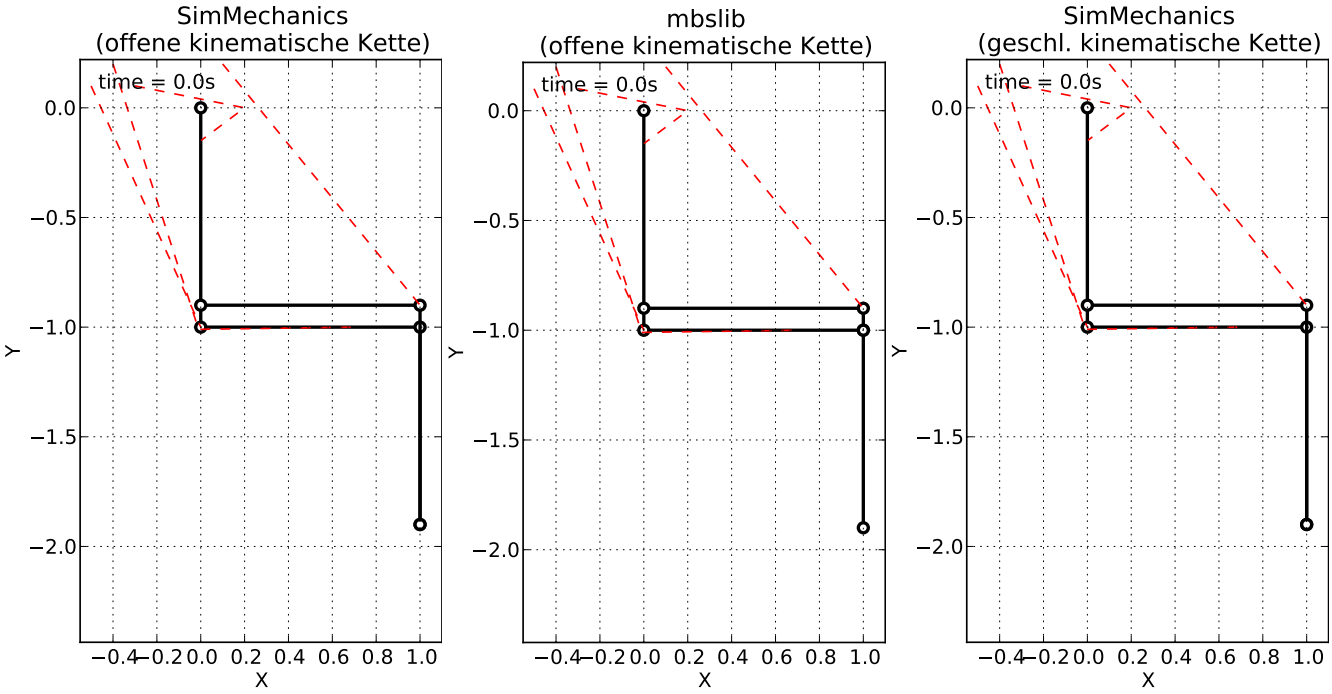

Abbildung 4.20: Visualisierung der initialen Position des Hinterbeins.

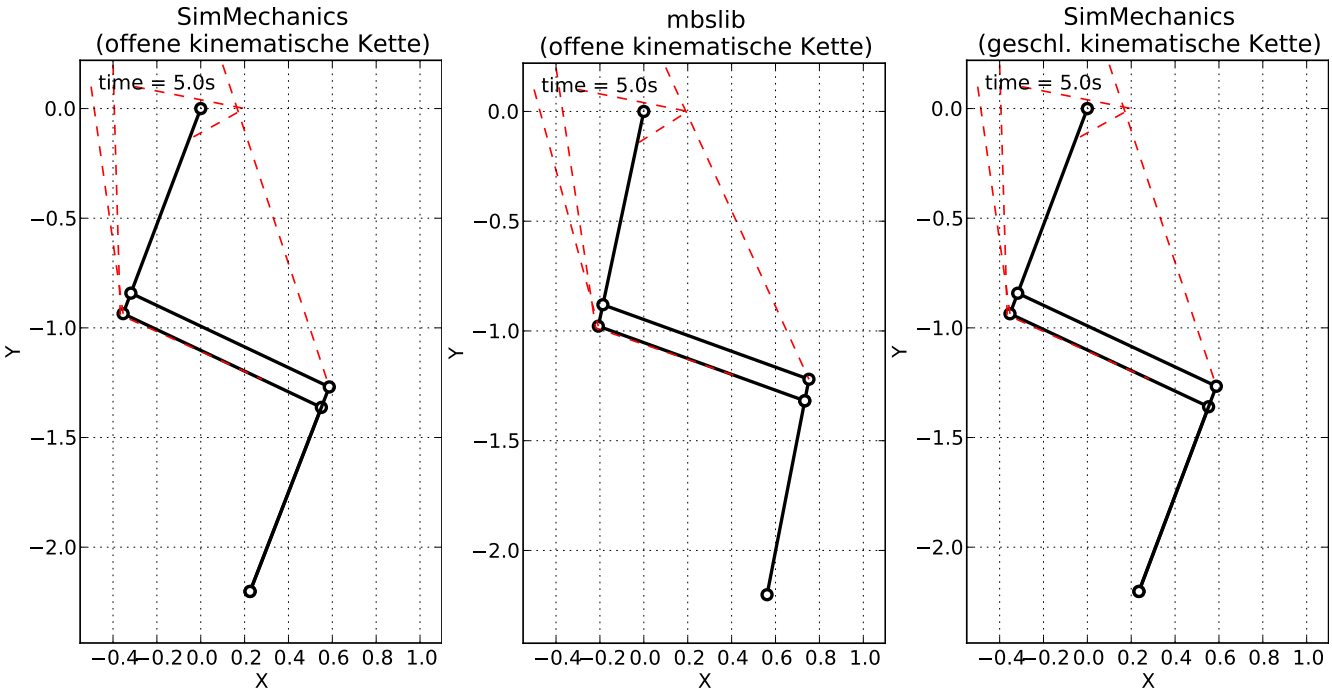

Abbildung 4.21: Visualisierung der Position des Hinterbeins nach fünf Sekunden Simulation.

In Abbildung 4.21 sind die Positionen der Hinterbeine nach fünf Sekunden Simulation aufgetragen. Die Positionen der beiden SimMechanics-Modelle (links und rechts) stimmen fast überein. Die Position des mbslib-Modells (mitte) weicht von den SimMechanics-Modellen stark ab. Der Unterschied der

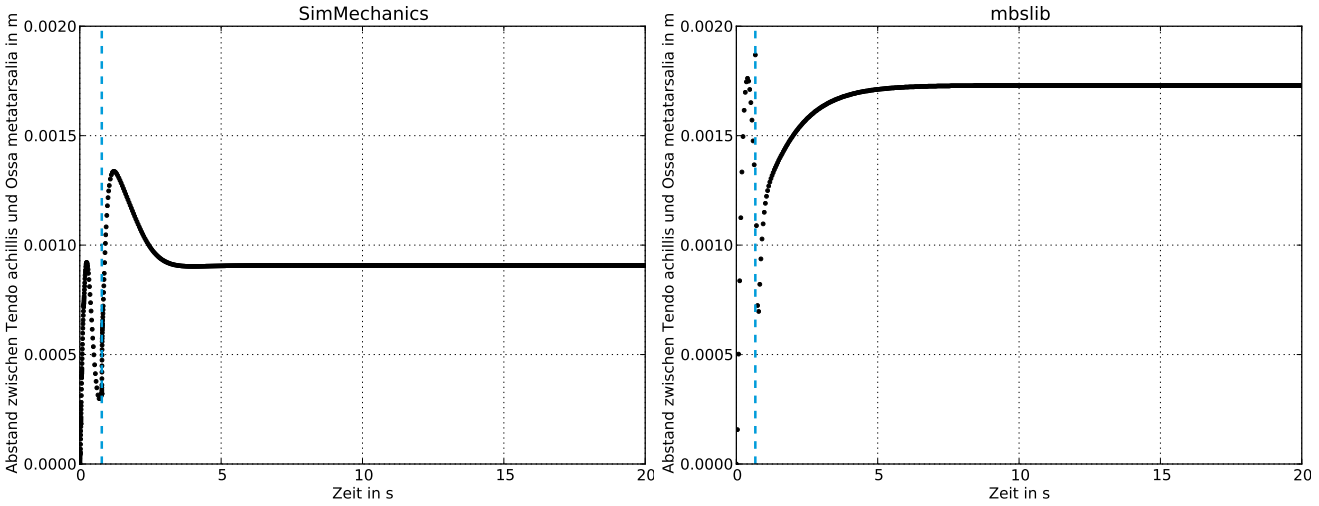

Abbildung 4.22: Hinterbein mit offener kinematischer Kette: Abstand zwischen Tendo achillis und Ossa metatarsalia. Der Zeitpunkt des Aufpralls auf den Boden ist markiert.

SimMechanics-Modelle ist auf die Modellierung als offene bzw. geschlossene kinematische Kette zurückzuführen. Bei der offenen kinematischen Kette wird die Feder gedehnt.

In Abbildung 4.22 ist der Abstand zwischen Tendo achillis und Ossa metatarsalia aufgetragen. Dabei ist der Zeitpunkt des ersten Kontakts mit dem Boden mit einer gestrichelten blauen Linie markiert. In der linken Abbildung, dem SimMechanics-Modell, kann diese Vermutung bestätigt werden. Zu Beginn der Simulation wird der Abstand zunächst oszillierend größer. Die Bodenberührung führt zu einer schnellen Vergrößerung des Abstands, danach wird er geringer, während das Bein zur Ruhe kommt. Dabei stellt sich ein Abstand ein, der knapp unter 1 mm liegt. Beim mbslib-Modell beginnt der Abstand ebenso bei 0. Danach steigt sie an. Zum Zeitpunkt des Bodenkontakts steigt der Abstand plötzlich auf bis zu 1,9 mm an (einzelner Punkt im Diagramm). Er sinkt zunächst schnell stark ab, um danach bis zur Ruheposition bei ca. 1,7 mm anzusteigen. Der größere Abstand beim mbslib-Modell ist auf die stark abweichende Ruhelage des Beins zurückzuführen.

In Abbildung 4.23 sind die Abweichungen der Gelenkwinkel aus der initialen Position aufgetragen. Bei den Modellen mit offener kinematischer Kette sind die Winkel R_2 und R_4 fast identisch. Die Abweichung entsteht durch die Feder. Bei beiden SimMechanics-Modellen ist der Winkelverlauf sehr ähnlich. R_1 kommt bei ca. -21°, R_2 und R_4 bei ca. -4° und R_3 (und R_5) bei ca. 4° zur Ruhe. Am Verlauf von R_2, R_3 und R_4 kann der Aufprall auf dem Boden als deutlicher erkennbarer Knick beobachtet werden. Der Verlauf beim mbslib-Modell ist ähnlich, allerdings kommen alle Kurven bei anderen Winkeln als beim SimMechanics-Modell zur Ruhe. Dies wird durch die abweichende Ruhelage verursacht.

Der Vergleich der beiden SimMechanics-Modelle zeigt, dass die Modellierung als offene bzw. geschlossene kinematische Kette am Beispiel des Hinterbeins nur geringe Unterschiede verursacht. Der Trick mit der Ersetzung eines Drehgelenks durch eine Feder scheint gut geeignet zu sein. Da keine Unterschiede in der Modellierung und den Parametern gefunden werden konnten, muss angenommen werden, dass die großen Unterschiede der Ruhelagen zwischen den Simulationen in SimMechanics und mbslib durch Unterschiede der Integratoren verursacht werden. Diese Annahme wird durch die zuvor beobachteten Abweichungen beim mathematischen Pendel (siehe Abschnitt 4.2.5) und dem Bodenkontakt-Testaufbau mit Pendel (siehe Abschnitt 4.2.5) gestärkt. Da der Aufbau durch die Verwendung mehrerer durch Drehgelenke verbundener Starrkörper und mehrerer Elastizitäten (in den Seilzügen und in der Feder, die als Ersatz für das Drehgelenk zwischen Tendo achillis und Ossa metatarsalia verwendet wird) komplex ist, führen kleine Veränderungen, ähnlich wie beim Dreifachpendel (siehe Abschnitt 4.5.4), zu verändertem Verhalten.

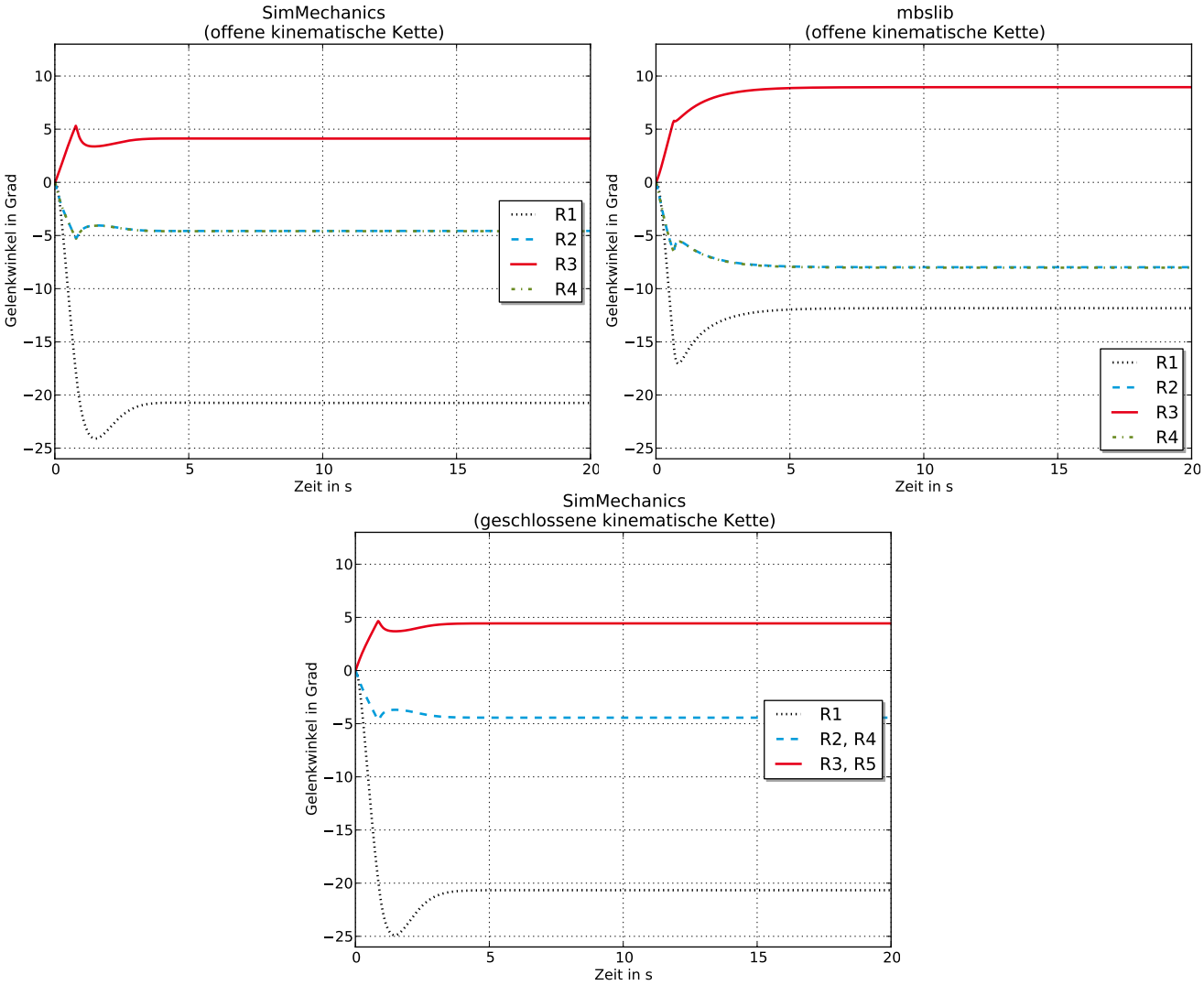

Abbildung 4.23: Hinterbein: Abweichung der Winkel der vier (bzw. fünf) Drehgelenke aus initialer Position.

4.7 Seilzüge

Auf Grund der Beobachtungen beim Vergleich des Verhalten des Hinterbeins in SimMechanics und mbslib scheint ein Vergleich des Verhaltens eines Seilzuges sinnvoll.

4.7.1 Aufbau

Die Seilzüge sind wie im vorigen Abschnitt beschrieben aufgebaut. Es werden zwei verschiedene Situationen verglichen: ohne und mit Umlenkrolle. Ein mathematisches Pendel ist an einem Drehgelenk mit Reibungskoeffizient 0,75 Nms/rad im Ursprung des Koordinatensystems befestigt. Es hat genau wie zuvor eine Länge von $l = 1$ m und eine Masse von $m = 1$ kg. Zu Beginn ist die Stange parallel zum Boden ausgerichtet. Der Massenpunkt befindet sich somit bei $x = 1$ m und $y = 0$ m. Der Seilzug hat eine Länge von 0,5 m, eine Federkonstante von $k = 100$ N/m und eine Dämpfung von $\mu = 100$ Ns/m.

Für den Aufbau ohne Umlenkrolle verbindet der Seilzug den Punkt $x = 1$ m und $y = 0$ m mit dem Massenpunkt. Der Aufbau eine Sekunde nach Beginn der Simulation in mbslib ist in Abbildung 4.24 links dargestellt. Beim Aufbau mit Umlenkrolle verbindet der Seilzug den Punkt $x = 0$ m und $y = 0$ m über

eine Umlenkrolle an $x = 1$ m und $y = 0$ m mit dem Massenpunkt. Der Aufbau ist in Abbildung 4.24 rechts abgebildet.

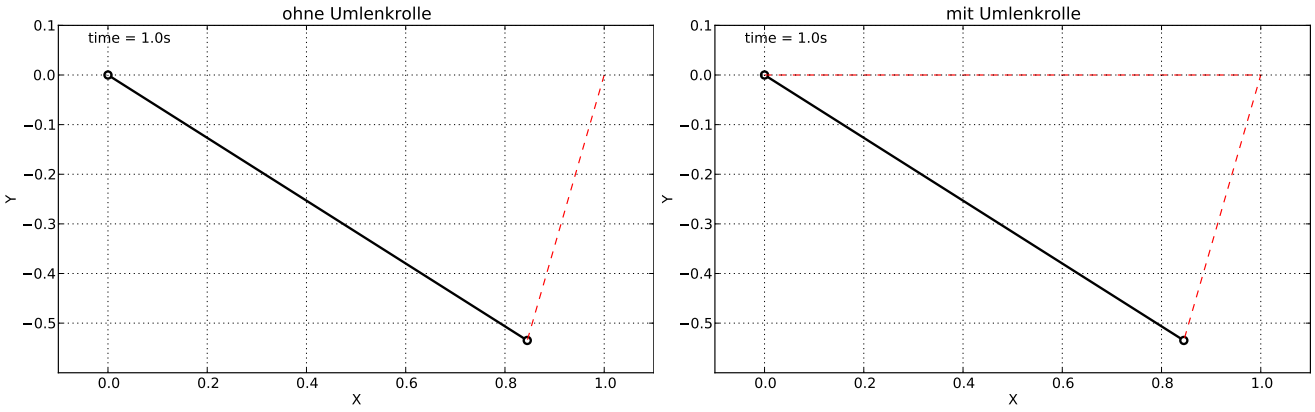

Abbildung 4.24: Mathematisches Pendel an einem Seilzug (nach einer Sekunde Simulation in mbslib).

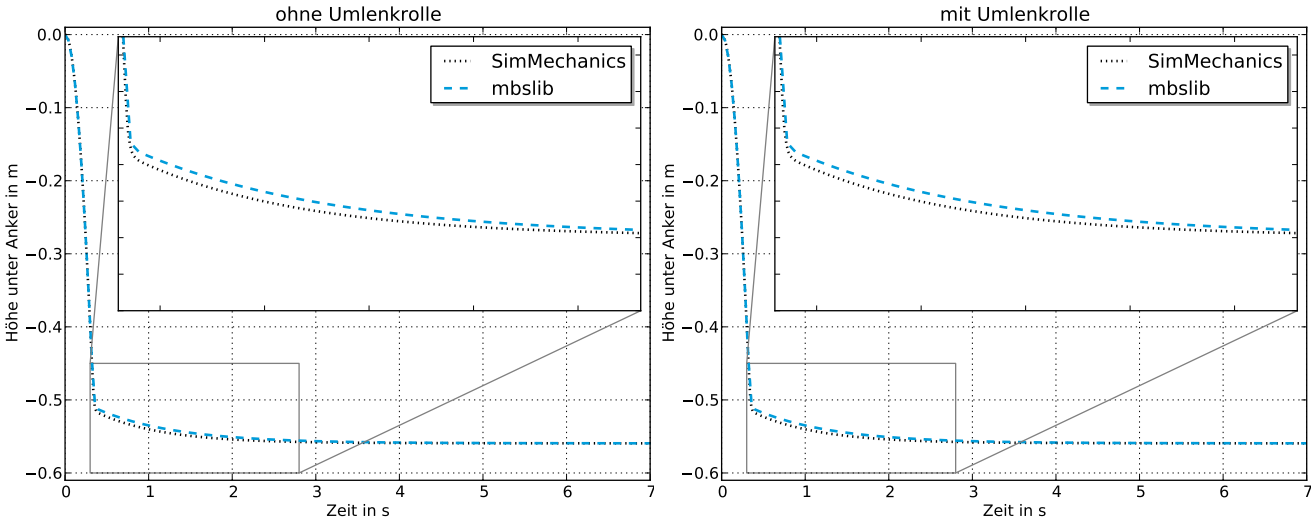

Abbildung 4.25: Höhe der Punktmasse des mathematischen Pendels an einem Seilzug unter dem Anker.

4.7.2 Ergebnisse der Simulation

Die Simulationsergebnisse sind in Abbildung 4.25 dargestellt. Hier ist wieder links das Verhalten ohne und rechts mit Umlenkrolle abgebildet. Die Höhe des Massenpunktes unter dem Anker in SimMechanics ist als schwarze gestrichelte Linie und die in mbslib als blau gestrichelte Linie aufgetragen. Wie man sehen kann, ist das Verhalten mit und ohne Umlenkrolle exakt gleich (siehe Abbildung 4.24). In der Vergrößerung (siehe Abbildung 4.25) kann man erkennen, dass der Verlauf in SimMechanics und mbslib abweicht. In mbslib wird das Pendel schneller abgebremst als in SimMechanics. Das ist darauf zurückzuführen, dass in SimMechanics die Schrittweitensteuerung zu Beginn des Aufpralls (plötzliches Auftreten großer abbremsender Kräfte) die Schrittweite verkleinert, wodurch die großen Kräfte kürzere Zeit wirken, während die in mbslib (mit konstanter Schrittweite) längere Zeit wirken. Dies ist ein weiterer Faktor, der zu den Unterschieden bei der Simulation des Hinterbeins führt.

4.8 Optimierung

Als Beispiel für eine Optimierung wurde die Maximierung der auf den Boden ausgeübten Kraft abhängig von den Endpunkten von Versus lateralis und Gluteus gewählt. Im folgenden Abschnitt wird betrachtet, wie eine solche Optimierung eines SimMechanics- und eines mbslib-Modells mittels der Optimierungstoolbox von Matlab vorgenommen werden kann.

4.8.1 Vorüberlegungen

Die Hinterbeine werden zu Beginn der Optimierung in die Ruhelage des SimMechanics-Modells mit offener kinematischer Kette gebracht (siehe Abbildung 4.21 links). Die Anfangswinkel sind $\varphi_1 = -20{,}7°$, $\varphi_2 = -4{,}57°$, $\varphi_3 = 4{,}12°$ und $\varphi_4 = -4{,}59°$. Im Vergleich zur Situation, in der diese Position die Ruhelage ist, ist in dieser Situation sowohl der Seilzug Versus lateralis um 0,5 m als auch der Seilzug Gluteus um 0,1 m verkürzt.

Die Zielfunktion ist eine Funktion, die angibt, wie gut die gewählten Parameter geeignet sind. Matlab benötigt eine Funktion, die die Simulation mit den richtigen Einstellungen startet und danach den Wert der Zielfunktion berechnet. Diese Auswertungsfunktion (in der Matlab-Anleitung allgemein `fun` oder `fitnessfcn` genannt) muss somit die Anfangsbedingungen der Simulation setzen, sie laufen lassen und danach die Simulationsergebnisse bewerten. Da bei einer Optimierung in Matlab die Zielfunktion minimiert wird, wird in diesem Fall das multiplikative Inverse des Maximums der während der Simulation auf den Boden ausgeübten Kräfte als Gütekriterium verwendet.

Auf Grund der Parametrisierung des Hinterbeins (siehe Abschnitt 4.6.2), sind beide Parameter (`fibula_versus_lateralis` und `os_femoris_gluteus`) nur im Intervall $[0, 1]$ physikalisch sinnvoll. Somit sollte ein Algorithmus zur Lösung eines beschränkten Optimierungsproblems verwendet werden. Falls die Ableitungen bekannt sind oder die Anzahl der Funktionsauswertungen egal ist, ist im Allgemeinen die Optimierung mittels eines gradientenbasierten Verfahrens wie z.B. *BFGS* (*Broyden-Fletcher-Goldfarb-Shanno*[13]) empfehlenswert. Da die Ausführung der Auswertungsfunktion hinsichtlich der Laufzeit teuer ist und die Hessematrix des Problems nicht bekannt ist, bietet sich ein Algorithmus an, der nur mit den tatsächlichen Funktionswerten arbeitet und somit keine Ableitungen numerisch nähern muss, was viele zusätzliche Funktionsauswertungen erfordert. Die Optimierungstoolbox von Matlab (optimtool) bietet für solche Probleme z.B. einen genetischen Algorithmus (*genetic algorithm*, Matlab-Funktion `ga`).

Die verwendete Auswertungsfunktion ist nicht fair, da zwar die Endpunkte der Seilzüge verändert werden, dabei aber nicht die Seillängen an die neue Situation angepasst werden.

4.8.2 Implementierung

SimMechanics:

Das SimMechanics-Modell, das zur Optimierung verwendet wird, wird analog zum Modell des Dreifachpendels (siehe Abschnitt 4.5.3) angepasst: alle Module werden in ein Untermodul verschoben und die zu variierenden Parameter werden zur Maske hinzugefügt. In der in Listing 4.6 abgedruckten Auswertungsfunktion wird das Modell zunächst geladen, danach werden die Parameter in der Maske des Modells eingesetzt. Zuletzt werden die im Bodenkontakt auftretenden Kräfte aus einer Datei geladen und das Maximum bestimmt. Der Rückgabewert der Funktion ist das multiplikative Inverse dieses Maximums. Die verwendete Auswertungsfunktion ist nicht parallelisierbar, da die Daten aus einer Datei gelesen werden müssen.

Der Quelltext zur Durchführung der eigentlichen Optimierung ist in Listing 4.7 dargestellt. Zunächst werden die Standard-Parameter eingelesen, danach werden die Seilzüge angespannt und die Anfangswinkel gesetzt. Zuletzt wird die Optimierung gestartet. Dazu wird die Funktion `ga` mit der Auswertungsfunktion, der Dimension des Parameterraums, der unteren Schranke, der oberen Schranke und dem

zuvor erstellten Einstellungsobjekt aufgerufen. Die Rückgabewerte werden auf verschiedene Variablen verteilt.

Für die Optimierung am Modell des Hinterbeins mit geschlossener kinematischer Kette unterscheiden sich beide Dateien nur geringfügig. Der Hauptunterschied ist, dass nur die Werte der ersten drei Drehgelenke gesetzt werden können, da das System sonst überbestimmt ist.

Listing 4.6: hinterbein_eval_simmechanics.m

```
function p=hinterbein_eval_simmechanics(q)
  load_system('hinterbein_opt_simmechanics');

  set_param('hinterbein_opt_simmechanics/subsys', ...
            'fibula_versus_lateralis', num2str(q(1)));
  set_param('hinterbein_opt_simmechanics/subsys', ...
            'os_femoris_gluteus', num2str(q(2)));

  sim('hinterbein_opt_simmechanics');

  p = load('hinterbein_bodenkontakt_kraft.mat', '-mat', 'ans');
  p = p.ans(3,:);
  p = 1.0/max(p);
end
```

Listing 4.7: hinterbein_opt_simmechanics_run.m

```
%Achtung: nicht parallelisierbar!
%Parameter laden
hinterbein_parameter
%Seilzuege verkuerzen
versus_lateralis_length = versus_lateralis_length - 0.5
gluteus_length = gluteus_length - 0.1
%Startwinkel
r1_q = -20.7
r2_q = -4.57
r3_q = 4.12
r4_q = -4.59

opt = gaoptimset()

[x, fval, exitcode, out, pop, scores] = ...
    ga(@hinterbein_eval_simmechanics, ...
       2, [], [], [], [], [0, 0], [1, 1], [], opt)
exit
```

mbslib:

Für die Optimierung des mbslib-Modells wird der in Listing 5.14 im Anhang abgedruckte Quelltext verwendet. Zuerst wird eine Instanz der Klasse hinterbein_parameter erstellt. In dieser Instanz werden die Seilzuglängen und die Position der Endpunkte von Versus lateralis und Gluteus gesetzt. Die beiden zu variierenden Parameter werden als Kommandozeilenparameter übergeben. Danach werden die Gelenkwinkel gesetzt und die Simulation gestartet. Während der Simulation wird das Maximum der im Bodenkontakt auftretenden Kraft bestimmt. Falls dieser Wert ungleich 0 ist, wird das Inverse dieses Wertes ausgegeben, ansonsten gibt man einen beliebigen großen Wert (hier $2 \cdot 10^6$) aus.

Somit muss die in Listing 4.8 abgedruckte Auswertungsfunktion nur noch das Modell starten und den Rückgabewert des externen Programms in eine Zahl umwandeln. Da das mbslib-Modell beliebig oft unabhängig voneinander gestartet werden kann, ist diese Auswertungsfunktion problemlos parallelisierbar.

Listing 4.8: hinterbein_eval_mbslib.m

```
function p=hinterbein_eval_mbslib(q)
  [status, result] = system(['./hinterbein_opt_mbslib ', ...
                             num2str(q(1)), ' ', ...
                             num2str(q(2))]);
  p = str2num(result);
end
```

Der in Listing 4.9 dargestellte Quelltext zur Durchführung der Optimierung unterscheidet sich hauptsächlich durch die Verwendung von Parallelisierung. Dazu wird mit matlabpool open 4 die Anzahl der Prozessoren eingestellt. Mit den Optionen von gaoptimset wird die Parallelisierung aktiviert und festgelegt, dass die Auswertungsfunktion keine Vektoren akzeptiert. Nach der Optimierung wird mit matlabpool close die Parallelisierung beendet.

Listing 4.9: hinterbein_opt_mbslib_run.m

```
matlabpool open 4

opt = gaoptimset('UseParallel', 'always', ...
                 'Vectorized', 'off')

[x, fval, exitcode, out, pop, scores] = ...
    ga(@hinterbein_eval_mbslib, ...
       2, [], [], [], [], [0, 0], [1, 1], [], opt)
matlabpool close
exit
```

Art	Laufzeit[2]	Parameter[3]	Wert der Zielfunktion
SimMechanics (offene kin. Kette)	24 Min	(0,4418/0,2345)	0,0265
SimMechanics (geschlossene kin. Kette)	20 Min	(0,4502/0,2156)	0,0295
mbslib	137 Min	(0,0340/0,0424)	0,0186
mbslib (parallelisiert)	57 Min	(0,0340/0,0424)	0,0186

Tabelle 4.2: Ergebnisse der Optimierung mittles ga.

4.8.3 Optimierung mit genetischem Algorithmus

Die Ergebnisse der Optimierung mit dem genetischen Algorithmus sind in Tabelle 4.2 aufgetragen. In allen Fällen wurde die Optimierung wegen Unterschreiten der Änderungstoleranz nach 51 Generationen abgebrochen. Die bestimmten Parameter und Zielfunktionswerte unterscheiden sich alle. Die durch die unterschiedliche Modellierung verursachte Abweichung zwischen den beiden SimMechanics-Modellen ist gering (ungefähr 10%). Die Unterschiede der Ausführungszeiten (24 Minuten bei offener kinematischer Kette und 20 Minuten bei geschlossener kinematischer Kette) sind ebenfalls ziemlich klein. Da diese Zeiten mit dem Kommandozeilenprogramm time gemessen wurden, können Abweichungen dieser Größenordnung durch viele Faktoren verursacht werden. Die am mbslib-Modell erhaltenen Parameter weichen, wie wegen der vorherigen Beobachtungen erwartet, deutlich ab. Wegen der fehlenden Schrittweitensteuerung des Integrators in mbslib ist die Ausführungszeit deutlich größer (137 Minuten). Wird die Optimierung in vier Threads parallelisiert durchgeführt, ist die Ausführungzeit immer noch um den

[2] Gemessen auf einem AMD Phenom™ 9650 Quad-Core-Prozessor.
[3] Notation der Parameter: (fibula_versus_lateralis/os_femoris_gluteus).

Faktor drei im Vergleich zu SimMechanics größer (57 Minuten). Die Optimierung mit und ohne Parallelisierung liefert, wie erwartet, die exakt gleichen Ergebnisse.

Die Werte der Zielfunktion (ca. 0,03 beim SimMechanics-Modell und ca. 0,02 beim mbslib-Modell) suggerieren, dass die für das mbslib-Modell gefundenen Parameter besser sind als die für das SimMechanics-Modell. Dies stimmt nicht. Wegen der unterschiedlichen Integratoren sind die Werte nicht vergleichbar. Beide Zielfunktionswerte befinden sich jedoch in einem plausiblen Bereich.

5 Diskussion

5.1 Vergleich

Modellierung:

Es ist schwierig, die unterschiedlichen Modellierungsansätze (grafisch und objektorientiert) objektiv miteinander zu vergleichen. Am Beispiel der Modellierung des mathematischen Pendels (siehe Abschnitt 4.2) kann der Leser selbst eine Präferenz entwickeln.

Kleine Systeme lassen sich von einer Person, die mit den Frameworks nicht vertraut ist, schneller in SimMechanics modellieren. Werden die zu modellierenden Systeme komplexer, kann ein grafisches Modell (wie in SimMechanics) sehr schnell unübersichtlich werden. Allerdings kann die Übersicht durch Verwendung von Untermodulen erhöht werden. Dies geschieht aber immer auf Kosten des Überblicks über die Funktion, da diese dadurch versteckt wird. Die Einstellung von Parametern über Dialoge und die Möglichkeit, selbst soche Dialog für eigene Untermodule zu erstellen, ist komfortabel. Ein großer Nachteil ist die Tatsache, dass es durch diese vielen verschiedenen Einstellungsdialoge nicht möglich ist, eine Übersicht über alle Parameter zu bekommen. Diese Art der Modellierung schränkt die Verwendbarkeit von Abbildungen von SimMechanics-Modellen stark ein, da, im Gegensatz zu einem mbslib-Modell, alle Implementierungsdetails versteckt sind. In mbslib ist es wesentlich einfacher zu überprüfen, ob alle Details korrekt sind, da sie auf einen Blick betrachtet werden können.

Die objektorientierte mbslib fördert durch Instanzierung und Vererbung die Wiederverwendung von Klassen. Es ist möglich, Modell und Simulations-Einstellungen zu trennen (siehe Abschnitt 4.6). Die SimMechanics-Modelle des Hinterbeins mit offener und geschlossener kinematischer Kette unterscheiden sich nur in einem Modul. Um die Version mit geschlossener kinematischer Kette zu erstellen, muss eine Kopie angefertig werden. Änderungen müssen danach an beiden Versionen parallel durchgeführt werden. Falls viele Änderungen gleichzeitig erfolgen, ist es weniger Aufwand, eine neue Kopie zu erstellen und die Anpassung auf die geschlossene Kette erneut durchzuführen. Es ist zwar möglich, ein eigenes Modul, das mehrfach verwendet wird, in eine Simulink-Bibliothek zu verwandeln, aber die Ersetzung von Blöcken durch Vererbung ist nicht möglich. Da das Überschreiben von Parametern kompliziert ist, ist es in vielen Fällen einfacher, eine Kopie anzufertigen und die Änderungen per Hand durchzuführen (mehr dazu im übernächsten Abschnitt).

Ein Nachteil bei der Modellierung in mbslib, der bei der Modellierung des Federpendels (siehe Abschnitt 4.3) sichtbar wird, ist die geringe Zahl der vorgefertigten Komponenten.

Erweiterbarkeit:

Am Beispiel des Bodenkontaktes (siehe Abschnitt 4.4) wurde in beiden Frameworks eine eigene Komponente erstellt.

In SimMechanics können durch Erstellung eigener Untermodule aus Kombinationen von vorhandenen Modulen oder eigenem Quelltext (in Matlab oder C) eigene Komponenten erstellt werden. Durch Bereitstellung von Modulen wie dem *Custom Joint*, einem Gelenk, dessen Freiheitsgrade durch Auswahl aus einer Liste von möglichen Freiheitsgraden selbst beliebig zusammengestellt werden können, werden dem Benutzer viele Erweiterungsmöglichkeiten gegeben.

In mbslib können, wegen der Objektorientierung, durch Vererbung mit geringem Aufwand beliebige fehlende Komponenten in C++ realisiert werden. Da der Quelltext von mbslib verfügbar ist, können auch grundlegende Algorithmen (wie z.B. der Integrator) ergänzt oder ausgetauscht werden. Dies ist in SimMechanics nicht möglich.

Automatisierbarkeit:

Am Beispiel des Dreifachpendels (siehe Abschnitt 4.5) und der Auswertungsfunktion bei der Optimierung (siehe Abschnitt 4.8) werden verschiedene Möglichkeiten zur Automatisierung von Simulationen in beiden Frameworks gezeigt. Dabei ist deutlich geworden, dass es wesentlich schwieriger ist, die Simulation eines SimMechanics-Modells zu automatisieren als die Simulation eines mbslib-Modells. In SimMechanics treten zwei Schwierigkeiten auf: Zum einen ist das automatisierte Ändern von Parametern über die Maske sehr umständlich und zum anderen ist es schwierig, die Daten aus Simulink in Matlab direkt zu verwenden, da dazu entweder eine Datei oder eine globale Variable des Workspaces verwendet werden muss. Leider ist es nicht möglich, beim Aufruf der Funktion sim zum Start der Simulation eines Simulink-Modells einen Rückgabewert zu erhalten. Ein mbslib-Modell kann beliebige Optionen per Kommandozeile entgegennehmen und kann ebenso beliebige Daten zurückgeben. Zudem können, im Gegensatz zu SimMechanics/Simulink, Daten nicht nur in eine Datei geschrieben werden, sondern auch an eine Datei angehängt werden. Ein großer Vorteil von mbslib ist außerdem, dass zum Speichern jedes beliebige Format verwendet werden kann, während SimMechanics/Simulink auf das von Matlab verwendete Binärformat beschränkt ist.

Dokumentation:

SimMechanics verfügt über eine eingebaute Dokumentation; mbslib über Doxygen-Kommentare, mit denen eine Dokumentation generiert werden kann. Die SimMechanics-Dokumentation ist ausführlicher als die mbslib-Dokumentation. Trotzdem sind einige wichtige Themen nicht ausführlich genug beschrieben. Es ist z.B. nicht einfach, durch Suche in der Dokumentation herauszufinden, wie die Simulation eines SimMechanics-Modells automatisiert werden kann. SimMechanics hat viele Benutzer, deshalb lassen sich im Internet viele Seiten finden, auf denen bei Problemen Hilfestellungen gegeben werden. Da mbslib wenige Benutzer hat, ist es schwierig, andere Personen zu finden, die auf gleiche Probleme bei der Bedienung gestoßen sind.

Zusätzlich zu den Doxygen-Kommentaren in mbslib gibt es eine Reihe von sehr hilfreichen Anwendungsbeispielen und Tutorials, in denen die Arbeit mit der Bibliothek vorgeführt wird. Ein weiterer Vorteil von mbslib hinsichtlich der Dokumentation ist die Verfügbarkeit des Quelltextes. Im Zweifelsfall, falls z.B. eine Funktion unklar oder ein Ergebnis merkwürdig ist, kann die tatsächlich verwendete Implementierung angesehen werden.

Verwendung:

SimMechanics ist, genau wie Matlab und Simulink, eine kommerzielle Software. Für die Verwendung werden Lizenzen benötigt. Diese werden für Systeme mit mehreren Nutzern, wie z.B. an der TU-Darmstadt, durch einen Lizenzserver verwaltet, der beim Programmstart kontaktiert wird und dynamisch Lizenzen zuteilt. Dadurch kann es zu Situationen kommen, in denen nicht genügend Lizenzen vorhanden sind und somit nicht mit SimMechanics gearbeitet werden kann. Diese Situation ist im Laufe dieser Arbeit mehrere Male aufgetreten. Für jede verwendete Toolbox ist eine separate Lizenz erforderlich. Bei Verwendung von Parallelisierung durch das *Parallel Computing Toolkit* reicht eine Lizenz pro verwendeter Toolbox aus[1]. Für parallele Ausführung der Auswertungsfunktion bei der Optimierung mit dem genetischen Algorithmus benötigt man z.B. genau eine Matlab-, eine Simulink-, eine Parallelisierungs-, eine Optimierungstoolbox- und eine SimMechanics-Lizenz.

Bei Verwendung von mbslib entfallen die oben beschriebenen Probleme. Ein weiterer Vorteil von mbslib gegenüber SimMechanics ist der geringere Speicherplatzbedarf der Bibliothek selbst (ca. 30 MB gegenüber ca. 3 GB). Allerdings werden für die Verwendung von mbslib mindestens ein C++-Compiler, make und cmake zusätzlich benötigt, die weiteren Speicherplatz verbrauchen. Trotzdem ist der Speicherplatzbedarf von mbslib deutlich geringer.

Ein Vorteil bei der Verwendung von mbslib ist die Verwendung von C++-Code zur Modellierung, da dieser Quelltext in einem normalen Texteditor bearbeitet werden kann und es somit möglich ist, sehr

[1] Aussage der MathWorks-Hotline vom 20.03.2012.

schnell kleine Änderungen vorzunehmen. SimMechanics-Modelle werden im Simulink-Modell-Format gespeichert. Dieses ist prinzipiell auch in einem Texteditor lesbar, aber eine Arbeit ohne Simulink ist quasi unmöglich, da es sich um eine stark verschachtelte Struktur handelt, die selbst bei kleinen Modellen sehr viele Zeilen umfasst. Das Modell des mathematischen Pendels benötigt in SimMechanics 1207 Zeilen, während das gleiche Modell in mbslib 85 Zeilen (inklusive einiger Leerzeilen und Kommentare zur besseren Lesbarkeit) umfasst. Ein weiterer Vorteil, den eine Verwendung eines übersichtlichen Text-Formats zur Speicherung von Modellen hat, ist die einfachere Untersuchung von Unterschieden zwischen Modellen durch Programme wie *diff*, die automatisch geänderte Textstellen markieren.

Visualisierung und Speichern von Ergebnissen:

Die genauen Eigenschaften sowie Vor- und Nachteile wurden schon in Abschnitt 4.1 ausführlich besprochen. Hier folgt eine kurze Zusammenfassung: SimMechanics besitzt eine interne Visualisierung, die den Vorteil hat, direkt auf die Daten des Modells zugreifen zu können. Sie hat den Nachteil, dass sie eigene Komponenten, wie z.B. die Seilzüge, nicht darstellen kann. Außerdem ist es nicht möglich fremde Daten zu visualisieren.

mbslib hat keine interne Visualisierung, aber es gibt verschiedene Ansätze, dieses Problem zu lösen: Das während dieser Arbeit erstellte Programm mbsplot kann beliebige Situationen in der x-y-Ebene animieren, hat aber den Nachteil, dass es die Daten nicht direkt aus dem Modell auslesen kann, sondern darauf angewiesen ist, dass es die Daten entweder aus einer Datei ausliest oder per Stdin erhält. Diese Daten können in mbslib einfach zur Verfügung gestellt werden. Es ist auch möglich, SimMechanics-Modelle zu visualisieren, allerdings ist das Sammeln und Speichern der Ergebnisse in SimMechanics aufwändig, da für jeden zu visualisierenden Eckpunkt die x- und y-Koordinate gespeichert werden müssen. Mit mbsplot ist es sehr einfach, Videos und einzelne Frames der Simulation anzuzeigen. Bei Darstellung einzelner Frames kann entweder nach einem bestimmten Zeitindex oder einer Framenummer gesucht werden. Beispiele für die mit mbsplot erstellten Ansichten sind z.B. in den Abbildungen 4.20, 4.21 oder 4.24 zu finden. Zur Visualisierung in mbslib ist ein weiteres Programm in der Entwicklung, das dreidimensionale Ansichten unter Verwendung von OpenGL erstellt.

Simulation:

Die Simulation selbst ist, wie schon zuvor beobachtet (siehe Abschnitte 4.2, 4.4, 4.6, 4.7 und 4.6), eine Schwäche von mbslib. Der einfache Euler-Integrator, den mbslib verwendet, hat mehrere Probleme: Es besteht vor allem bei zu kleiner Schrittweite die Gefahr, dass bei einer Simulation durch den Integrator Energie in das System gebracht wird. Ein einfaches Beispiel, um dieses Verhalten zu provozieren, ist das ungedämpfte Pendel (siehe Abschnitt 4.2.3) mit einer Schrittweite von $\Delta t = 0{,}01$ s. Im Verlauf der Simulation nimmt die Amplitude (siehe Abbildung 5.1 links) und die maximale Geschwindigkeit (rechts) der Schwingung des Pendels immer weiter zu.

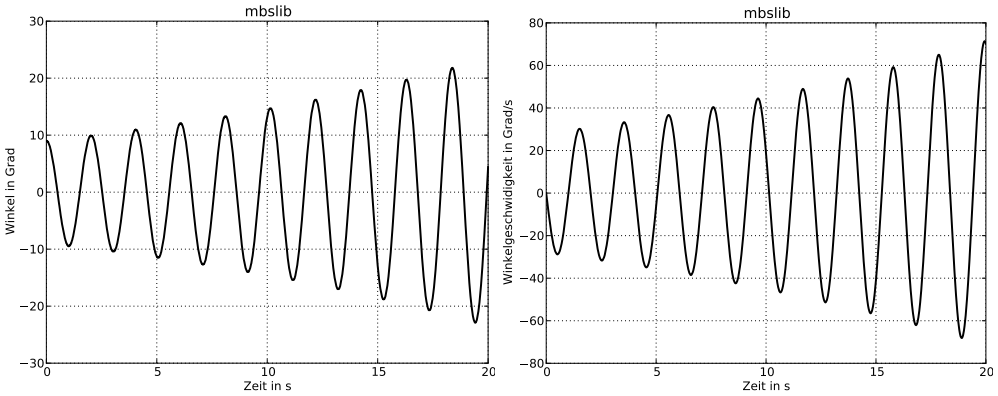

Abbildung 5.1: Schwingung eines gedämpften Pendels bei zu kleinem Δt. Der Euler-Integrator erhöht die Energie des Systems. Links: Position über Zeit. Rechts: Geschwindigkeit über Zeit.

Dieses Problem wird durch eine fehlende Schrittweitensteuerung verstärkt. Der Integrator ist nicht in der Lage, eine geeignete Schrittweite zu wählen, die das Auftreten von Fehlern während der Simulation minimiert. Vor allem an Zeitpunkten, an denen große Kräfte wirken, ist eine kleine Schrittweite nötig. Damit bleiben für den Benutzer zwei Optionen: entweder eine Schrittweite zu wählen, mit der die Simulation in einer akzeptablen Zeit durchläuft, und dabei die dadurch verursachten Fehler in Kauf zu nehmen oder eine so kleine Schrittweite zu wählen, dass die Simulation plausiblere Ergebnisse[2] liefert, aber sehr lange dauert.

Wenn die Feder des ungedämpften Federpendels (siehe Abschnitt 4.3) sehr steif wird, also z.B. eine Federkonstante k=1000 N/m bei einer Masse von m=1 kg verwendet wird, sind sehr kleine Schrittweiten ($\Delta t \approx 0{,}000001$ s) nötig, damit die Amplitude der Schwingung bei der Simulation nicht innerhalb von 20 Sekunden vervielfacht wird. Die Simulation dauert in diesem Fall dann ca. 84 Minuten[3].

Diese Probleme treten in SimMechanics nicht auf, da SimMechanics über eine Vielzahl von Integratoren mit Schrittweitensteuerung verfügt. Zum einen sind dies bessere Algorithmen, die auch bei größeren Schrittweiten brauchbare Ergebnisse liefern und zum andern gibt es eine Schrittweitensteuerung. Dies ist der Grund, warum die Simulationen und vor allem die Optimierungen (siehe Abschnitt 4.8) in SimMechanics schneller ausgeführt werden und trotzdem in Momenten, in denen große Kräfte auftreten, bessere Ergebnisse liefern. Wie im Abschnitt „Erweiterbarkeit“ erwähnt, ist es aber möglich, den in mbslib verwendeten Integrator auszutauschen, um die soeben beschriebenen Probleme zu lösen.

Weitere Funktionalität:

mbslib verfügt über weitere Funktionalitäten, die in SimMechanics nicht verfügbar sind: Es können mittels der Bibliothek *ADOL-C* über die Klasse `DeriveOMat` automatisch Ableitungen durchgeführt werden. Wenn ADOL-C verwendet wird, verlangsamt die Bibliothek die Ausführung des Programms. Die Verwendung von ADOL-C in mbslib ist völlig optional und, durch Verwendung gleicher Typenbezeichner, transparent gegenüber dem Modell. Zudem kann mbslib die Jakobimatrix und die Sensitivität eines Manipulators berechnen. Die Jakobimatrix eines Manipulators wird zur Umrechnung zwischen seinen Gelenkgeschwindigkeiten und seinen kartesischen Geschwindigkeiten benötigt. Sie kann mit der Methode `calculateJacobian()` der Klasse `MbsCompound` berechnet werden. Die Sensitivität gibt an, wie stark die Kräfte/Drehmomente τ oder die Gelenkbeschleunigungen $\ddot{q}$ auf eine Veränderung der Kontrollparameter reagieren. Die jeweilige Sensitivitäts-Matrix kann mit `calculateDtauDcontrol()` bzw. `calculateDddqDcontrol()` (verwendet Ergebnis aus `calculateDtauDcontrol()`) oder `calculateDddqDcontrol2()` analytisch berechnet werden.

5.2 Zusammenfassung

Im Rahmen dieser Arbeit wurden verschiedene physikalische Systeme in SimMechanics und mbslib modelliert und dabei die Unterschiede in der Modellierung, der Durchführung der Simulation und den Simulationsergebnissen untersucht.

Die beiden unterschiedlichen Modellierungs-Arten haben Vor- und Nachteile, die im vorigen Abschnitt beschrieben sind. Falls ein Anwender mit beiden Frameworks nicht vertraut ist, noch nie mit einer objektorientierten Programmiersprache gearbeitet hat und möglichst schnell ein einfaches Modell erstellen will, wird er mit SimMechanics schneller Erfolg haben. Wenn die Modelle komplexer werden, kann die grafische Programmierung in SimMechanics schnell unübersichtlich werden. mbslib hat hier den Vorteil größerer Übersicht durch direkte Einsicht in die eingestellen Parameter und durch die Möglichkeit, das Modell komplett zu drucken und zwischen verschiedenen Versionen zu vergleichen.

In SimMechanics ist es sehr einfach, eine Simulation zu starten und die interne Visualisierung zu verwenden, um das Simulationsergebnis zu betrachten. Die Verwendung der erzeugten Daten ist in mbslib

[2] Keine Energiezunahme bei einer ungedämpften Bewegung in einem abgeschlossenen System.
[3] Auf einem Intel® Core™ i5-2400S.

wesentlich einfacher, da direkt über Verwendung entsprechender Methoden auf die Daten zugregriffen werden kann und diese über die üblichen Funktionen in C++ verwendet werden können (z.B. mit `std::cout`). In SimMechanics sind für jeden Wert mindestens zwei Module erforderlich: ein Sensor und das Modul *tofile*, das eine Datei im Matlab-Binärformat erzeugt. Falls mehrere unterschiedliche Werte als Tabelle geschrieben werden sollen, kommt ein *Muxer* hinzu, an den alle verwendeten Sensoren angeschlossen sind. Zur Aufzeichnung der kontinuierlichen Gelenkwinkel und Geschwindigkeiten des Dreifachpendels werden fast so viele Module und Signalleitungen (die durch das ganze Modell gehen) benötigt wie zur Modellierung des Dreifachpendels selbst. Die Veränderung von Parametern der Simulation ist in mbslib durch Verwendung der Kommandozeile deutlich einfacher, da, im Gegensatz zu SimMechanics, nicht der Umweg über die Erstellung eines Untermoduls und über die Änderung der Einträge der Maske vorgenommen werden muss.

Der einzige richtige Nachteil, den die Verwendung von mbslib statt SimMechanics zum aktuellen Zeitpunkt hat, ist die Simulation selbst. Wie zuvor beschrieben, hat der in mbslib verwendete explizite Euler-Integrator (ohne Schrittweitensteuerung) starke Nachteile hinsichtlich Genauigkeit und Ausführungszeit im Vergleich zu den von SimMechanics verwendeten Integratoren. Dieses Problem sollte durch bessere Integratoren mit Schrittweitensteuerung gelöst werden, die aktuell in mbslib implementiert werden. Wenn diese Integratoren die angesprochenen Probleme lösen, würde ich die Verwendung von mbslib der Arbeit mit SimMechanics vorziehen, da ich diese einfacher finde.

Das im Verlauf der Arbeit erstellte Visualisierungsprogramm mbsplot vereinfacht die Arbeit in mbslib und ermöglicht es in beiden Frameworks mit geringem Aufwand Videos und Bilder zu beliebigen Simulationszeiten zu erstellen. Die Möglichkeit eigene Komponenten darzustellen (im Verlauf der Arbeit hauptsächlich Seilzüge) hilft, die korrekte Funktion dieser Komponenten mit allen ihren Parametern zu überprüfen.

5.3 Ausblick

Sobald die neuen Intregratoren der mbslib fertiggestellt sind, sollte eine weitere genaue Untersuchung der Simulationsergebnisse durchgeführt werden. Diese Untersuchung könnte anhand der im Verlauf dieser Arbeit erstellten Modelle und Auswertungsskripte durchgeführt werden. So könnte, mit relativ geringem Aufwand, mit ihnen ein Benchmark für verschiedene Integratoren erzeugt werden. Dabei könnte vor allem der Winkelverlauf des Dreifachpendels bei gleichen Startbedingungen und verschiedenen Integratoren und Integrator-Einstellungen aufschlussreich sein. Ich fände es interessant zu wissen, ob es durch geschickte Parameterwahl möglich ist, über längere Simulationszeiten eine ähnliche Bewegung in SimMechanics und mbslib zu erhalten.

Mit den erstellten Modellen des Hinterbeins könnten weitere Untersuchungen, wie z.B. das Gehen des Beins, durchgeführt werden. Dazu müsste eine Steuerung oder Regelung der Seilzuglängen erstellt werden. Wenn diese vorhanden wäre, könnten Optimierungen zur Maximierung der Schrittgeschwindigkeit durchgeführt werden. Für Optimierungen, die über die Maximierung der Schrittweite hinausgehen, müssten Haft- und Gleitreibung in den Bodenkontakt eingebaut werden.

Um Anwendern, die noch nicht mit einer objektorientierten Programmiersprache gearbeitet haben, den Einstieg in die Arbeit mit mbslib zu erleichtern, könnten Teile dieser Arbeit als Dokumentation übernommen werden. Dazu wäre eine englische Übersetzung wünschenswert. Außerdem wäre es hilfreich, wenn die mbslib-Beispiele stärker kommentiert wären. Die Verwendung der Beispiele wäre einfacher, wenn es eine Liste zur Übersicht über die vorhandenen Beispiele gäbe.

Zudem wäre es interessant, die Funktionen von mbslib, die unter dem Abschnitt „Weitere Funktionalität“ des Kapitels 5.1 erwähnt wurden, genauer zu untersuchen, da diese für die Optimierung von Bewegungsabläufen (mit gradientenbasierten Verfahren) sehr nützlich sind.

Abbildungsverzeichnis

2.1 Erste bis dritte Abbildung: Aufbau eines Hinterbeins eines Schäferhundes (aus [3]). Vierte Abbildung: Bio-inspiriertes Hinterbein. 4

3.1 Beispiel für die Baumstruktur eines Roboterbeins (aus [4, 5]). Links: Aufbau. Mitte: Details. Rechts: Baumstruktur in mbslib. 5

4.1 Aufbau des mathematischen Pendels. 7
4.2 Trajektorien eines reibungsfreien Pendels im Phasenraum (aus [11]). 8
4.3 Modellierung des reibungsfreien mathematischen Pendels in SimMechanics. 9
4.4 Eigenschaften des Starrkörpers in SimMechanics. 9
4.5 Modellierung des gedämpften mathematischen Pendels in SimMechanics. Änderungen gegenüber der reibungsfreien Version sind blau markiert. 11
4.6 Trajektorie eines Pendels im Phasenraum. Links: reibungsfrei. Rechts: mit Reibung. 11
4.7 Aufbau des Federpendels. 12
4.8 Modell des Federpendels in SimMechanics. 13
4.9 Baumstruktur des Federpendels (mbslib). 14
4.10 Auslenkung des Federpendels aus der Ruhelage. Links: ungedämpft. Rechts: gedämpft. . . 14
4.11 Modellierung des Bodenkontakts in SimMechanics. 16
4.12 Höhe des Kontaktpunktes über dem Boden. 17
4.13 Aufbau des Dreifachpendels. Das blau markierte Drehgelenk ist verankert. 18
4.14 Positionen des Dreifachpendels nach 20 Sekunden Simulation. Die initialen Winkel sind per Pseudozufallszahlen zwischen 0 rad und 10^{-8} rad 100 Mal variiert. 20
4.15 Verlauf der Winkel des Dreifachpendels innerhalb von 20 Sekunden. 21
4.16 Aufbau des Hinterbeins (Seitenansicht: links ist vorne). 22
4.17 Parametrisierung des Hinterbeins. 24
4.18 Modellierung des Hinterbeins in SimMechanics. 25
4.19 Modellierung des Hinterbeins in SimMechanics als geschlossene kinematische Kette. . . . 26
4.20 Visualisierung der initialen Position des Hinterbeins. 27
4.21 Visualisierung der Position des Hinterbeins nach fünf Sekunden Simulation. 27
4.22 Hinterbein mit offener kinematischer Kette: Abstand zwischen Tendo achillis und Ossa metatarsalia. Der Zeitpunkt des Aufpralls auf den Boden ist markiert. 28
4.23 Hinterbein: Abweichung der Winkel der vier (bzw. fünf) Drehgelenke aus initialer Position. 29
4.24 Mathematisches Pendel an einem Seilzug (nach einer Sekunde Simulation in mbslib). . . . 30
4.25 Höhe der Punktmasse des mathematischen Pendels an einem Seilzug unter dem Anker. . . 30

5.1 Schwingung eines gedämpften Pendels bei zu kleinem Δt. Der Euler-Integrator erhöht die Energie des Systems. Links: Position über Zeit. Rechts: Geschwindigkeit über Zeit. 37

Tabellenverzeichnis

2.1 Versionen der verwendeten Matlab/Simulink-Komponenten. 3

4.1 Vollständige Parametrisierung des Hinterbeins. 23
4.2 Ergebnisse der Optimierung mittles ga. 33

Listingverzeichnis

4.1 Ausschnitt aus GroundContact.cpp (siehe Listing 5.4 im Anhang). 16
4.2 Ausschnitt aus bodenkontakt_mp.cpp (siehe Listing 5.5 im Anhang). 17
4.3 makeInertiaTensorCylinderX.m 18
4.4 dreifachpendel_20s.m 19
4.5 dreifachpendel_run.m 19
4.6 hinterbein_eval_simmechanics.m 32
4.7 hinterbein_opt_simmechanics_run.m 32
4.8 hinterbein_eval_mbslib.m 33
4.9 hinterbein_opt_mbslib_run.m 33

5.1 mathematisches_pendel.cpp 44
5.2 federpendel.cpp 45
5.3 GroundContact.h 47
5.4 GroundContact.cpp 49
5.5 bodenkontakt_mp.cpp 50
5.6 dreifachpendel_rnd.py 51
5.7 dreifachpendel_20s.cpp 52
5.8 dreifachpendel_run.py 53
5.9 hinterbein_parameter.h 54
5.10 hinterbein_parameter.h 55
5.11 hinterbein_modell.h 56
5.12 hinterbein_modell.cpp 57
5.13 hinterbein.cpp 64
5.14 hinterbein_opt.cpp 65

Literaturverzeichnis

[1] T. Lens, K. Radkhah, and O. von Stryk, "Simulation of dynamics and realistic contact forces for manipulators and legged robots with high joint elasticity," in *Proc. 15th International Conference on Advanced Robotics (ICAR)*, pp. 34–41, 2011.

[2] M. Hardt, M. Stelzer, and O. von Stryk, "Modellierung und Simulation der Dynamik des Laufens bei Roboter, Tier und Mensch," in *thema Forschung*, vol. 2/2002, pp. 56–63, Technische Universität Darmstadt, 2002.

[3] K. Radkhah, S. Kurowski, and O. von Stryk, "Design considerations for a biologically inspired compliant four-legged robot," in *Proc. 2009 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 598–603, Dec 19-23 2009.

[4] M. Friedmann, "Übersicht mbslib." Vortrag (annual SIM-Workshop, Überlingen), Oktober 2011.

[5] M. Friedmann, C. Reinl, D. Scholz, and O. von Stryk, "Efficient simulation and parameter estimation for a biologically inspired biped robot," 2011.

[6] K. Radkhah, C. Maufroy, M. Maus, D. Scholz, A. Seyfarth, and O. von Stryk, "Concept and design of the biobiped1 robot for human-like walking and running," *International Journal of Humanoid Robotics*, vol. 8, no. 3, pp. 439–458, 2011.

[7] D. Arndt, J. E. Bobrow, S. Peters, K. Iagnemma, and S. Dubowsky, "Self-balancing control of a four wheeled vehicle," April 2009.

[8] O. Unver, A. Uneri, A. Aydemir, and M. Sitti, "Geckobot: a gecko inspired climbing robot using elastomer adhesives," in *Proc. IEEE International Conference on Robotics and Automation*, pp. 2329–2335, 2006.

[9] M. Stelzer, O. von Stryk, E. Abele, J. Bauer, and M. Weigold, "High speed cutting with industrial robots: Towards model based compensation of deviations," in *Proceedings of Robotik 2008, 11-12 June*, pp. 143–146, 2008.

[10] E. Abele, J. Bauer, C. Bertsch, R. Laurischkat, H. Meier, S. Reese, M. Stelzer, and O. von Stryk, "Comparison of implementations of a flexible joint multibody dynamic system model for an industrial robot," in *6th CIRP International Conference on Intelligent Computation in Manufacturing Engineering*, July 2008.

[11] B. Drossel, "Vorlesung Komplexe Dynamische Systeme," Wintersemester 2009/10.

[12] W. Demtröder, *Experimentalphysik 1: Mechanik und Wärme*. Springer, 2006.

[13] J. E. Dennis, Jr. and J. J. Morée, "Quasi-newton methods, motivation and theory," *SIAM Review*, vol. 19, no. 1, 1977.

Anhang

Quelltext

Modellierung des reibungsfreien mathematisches Pendels in mbslib:

Listing 5.1: mathematisches_pendel.cpp

```
#include <mbs/MbsCompoundWithBuilder.h>

#include <iostream>
#include <iomanip>

int main(void)
{
    mbslib::Joint1DOF * j1;
    mbslib::Endpoint * e1;

    mbslib::MbsCompoundWithBuilder mbs("Mathematisches Pendel");

    mbs.addFixedBase("Anker");

    j1 = mbs.addRevoluteJoint(mbslib::TVector3(0,0,1),
                              "Drehgelenk");
    mbs.addRigidLink(mbslib::TVector3(0,-1,0),
                     mbslib::TVector3(0,0,0),
                     0,
                     mbslib::TMatrix3x3::Zero());
    e1 = mbs.addEndpoint(1,
                         mbslib::TMatrix3x3::Zero(),
                         "Punktmasse");

    assert(mbs.isValid());
    mbs.setGravitation(mbslib::TVector3(0, -9.81, 0));

    // Anfangsbedingungen:
    j1->setJointPosition(9*M_PI/180);
    j1->setJointVelocity(0);
    j1->setJointAcceleration(0);
//    j1->setParameter(0, 1); // Gelenkreibung
    j1->setParameter(0, 0); // keine Gelenkreibung

    mbs.doDirkin();

    double dt = 0.0001;
    double t = 20;
    double fps = 25;
    std::cout << "!PLfps=25!\n";
    int skip = 1/fps/dt;

    std::cout << std::setiosflags(std::ios::fixed)
              << std::setprecision(6);

```

```
46      std::cout << "\n#t\tq\tdq\n";
47
48      for (int i = 0; i <= t/dt; i++) {
49          if (i%skip==0) {
50              std::cout << (i*dt) << "\t"
51                        << (j1->getJointPosition()/M_PI*180) << "\t"
52                        << (j1->getJointVelocity()/M_PI*180)
53                        << std::endl;
54              // print coordinates for visualization
55              std::cout << "!PL!" << (i*dt) << "\t"
56                        << j1->getCoordinateFrame().r.x() << "\t"
57                        << j1->getCoordinateFrame().r.y() << "\t"
58                        << e1->getCoordinateFrame().r.x() << "\t"
59                        << e1->getCoordinateFrame().r.y()
60                        << std::endl;
61          }
62          mbs.doABA();
63          mbs.integrate(dt);
64      }
65
66
67      std::cout << (t) << "\t"
68                << (j1->getJointPosition()/M_PI*180) << "\t"
69                << (j1->getJointVelocity()/M_PI*180)
70                << std::endl;
71
72      // print coordinates for visualization
73      std::cout << "!PL!" << (t) << "\t"
74                << j1->getCoordinateFrame().r.x() << "\t"
75                << j1->getCoordinateFrame().r.y() << "\t"
76                << e1->getCoordinateFrame().r.x() << "\t"
77                << e1->getCoordinateFrame().r.y()
78                << std::endl;
79
80      std::cout << "!PLtitle=mathematisches␣Pendel␣in␣mbslib!\n";
81      std::cout << "!PLxlim=[-0.5,0.5]!\n";
82
83      return 0;
84  }
```

Modellierung des reibungsfreien Federpendels in mbslib:

Listing 5.2: federpendel.cpp

```
1   #include <mbs/MbsCompoundWithBuilder.h>
2   #include <mbs/LinearSpringModel.h>
3
4   #include <iostream>
5   #include <iomanip>
6
7   int main(void)
8   {
9       mbslib::Joint1DOF * pris;
10      mbslib::Endpoint * e1;
11      mbslib::Endpoint * e2;
12      mbslib::Spring3D * feder;
```

```
13
14      mbslib::MbsCompoundWithBuilder mbs("Federpendel");
15
16      mbslib::FixedBase * base = mbs.addFixedBase("Basis");
17
18      // Zusaetzlicher Freiheitsgrad:
19      mbs.addRevoluteJoint(mbslib::TVector3(0, 0, 1));
20      mbs.addFork();
21
22      // Trick zur Verlaengerung der Feder:
23      mbs.addFixedTranslation(mbslib::TVector3(0, -1, 0));
24      e1 = mbs.addEndpoint("Anfang der Feder");
25
26      // Freier Massenpunkt
27      pris = mbs.addPrismaticJoint(mbslib::TVector3(0, -1, 0));
28      e2 = mbs.addEndpoint(1,
29                           mbslib::TMatrix3x3::Zero(),
30                           "Punktmasse");
31
32      // Feder (Daempfung = 1 in SimMechanics):
33      mbslib::LinearSpringModel federModell(10, 0);
34      feder = mbs.addSpring(federModell, "Feder");
35      (* feder) << e1 << e2;
36
37      assert(mbs.isValid());
38      mbs.setGravitation(mbslib::TVector3(0, -9.81, 0));
39
40      // Anfangsbedingungen:
41      pris->setJointPosition(1.0);
42
43      mbs.doDirkin();
44
45      double dt = 0.0001;
46      double t = 20;
47      double fps = 25;
48      std::cout << "!PLfps=25!\n";
49      int skip = 1/fps/dt;
50
51      std::cout << std::setiosflags(std::ios::fixed)
52                << std::setprecision(6);
53
54      std::cout << "\n#t\tq\tdq\n";
55
56      for (int i = 0; i <= t/dt; i++) {
57          if (i%skip==0) {
58              std::cout << (i*dt) << "\t"
59                        << (pris->getJointPosition()) << "\t"
60                        << (pris->getJointVelocity())
61                        << std::endl;
62
63              std::cout << "!PL!" << (i*dt) << "\t"
64                        << base->getCoordinateFrame().r.x() << "\t"
65                        << base->getCoordinateFrame().r.y() << "\t"
66                        << e2->getCoordinateFrame().r.x() << "\t"
67                        << e2->getCoordinateFrame().r.y()
```

```
                        << std::endl;

        }

        mbs.doABA();
        mbs.integrate(dt);
    }

    std::cout << (t) << "\t"
              << (pris->getJointPosition()) << "\t"
              << (pris->getJointVelocity())
              << std::endl;

    std::cout << "!PL!" << (t) << "\t"
              << base->getCoordinateFrame().r.x() << "\t"
              << base->getCoordinateFrame().r.y() << "\t"
              << e2->getCoordinateFrame().r.x() << "\t"
              << e2->getCoordinateFrame().r.y()
              << std::endl;

    std::cout << "!PLtitle=Federpendel in mbslib!\n";
    std::cout << "!PLxlim=[-1.1, 1.1];ylim=[-3.1, 0.1]!\n";
    std::cout << "!PLOset_marker=o,set_markerfacecolor=b!\n";
    std::cout << "!PLOset_linewidth=1,set_color=r,set_linestyle=dashed!\n";

    return 0;
}
```

Bodenkontakt in mbslib:

Listing 5.3: GroundContact.h

```
#ifndef GROUNDCONTACT_H
#define GROUNDCONTACT_H
#include <mbs/IForceGenerator.h>
#include <mbs/MbsCompoundWithBuilder.h>

namespace mbslib {
/**
 * \brief The GroundContact is an IForceGenerator which is used to simulate
 * the interactions of a point with the ground.
 *
 * A simple spring-damper model is used.
 */
class GroundContact : public IForceGenerator {
public:
    /**
    * \brief Constructor.
    *
    * \param joint The joint to limit.
    * \param limit The position limit.
    * \param direction The limiting direction: +1 == up, -1 == down.
    */
    GroundContact(Endpoint & point, TScalar position, TScalar direction,
                  TScalar springConstant, TScalar dampingConstant);

```

```
25      /**
26      * \brief Apply force of this IForceGenerator to the mbs.
27      */
28      virtual void applyForce();
29
30      /**
31      * \brief Get force of this IForceGenerator to the mbs.
32      */
33      TScalar getForce();
34
35      /**
36      * \brief Reset force introduced by this IForceGenerator.
37      */
38      virtual void resetForce();
39
40      /**
41      * \brief Get name of this IForceGenerator.
42      */
43      virtual const std::string & getName()const;
44
45      /**
46      * \brief Get Integrator. We have no state, so we need none and
47      * return NULL.
48      *
49      * \return Always NULL.
50      */
51      IIntegrate * getIntegrator(){return NULL;}
52
53  protected:
54      /// The joint.
55      Endpoint & point;
56
57      /// Ground position.
58      TScalar position;
59
60      /// Limiting direction.
61      TScalar direction;
62
63      /// Spring constant for rebound.
64      TScalar springConstant;
65
66      /// Damping constang.
67      TScalar dampingConstant;
68
69      /// Name of this limiter.
70      std::string name;
71
72      /// The last calculated force
73      TScalar force;
74  };
75  } // namespace mbslib
76  #endif // GROUNDCONTACT_H
```

Listing 5.4: GroundContact.cpp

```
#include "GroundContact.h"

namespace mbslib {
GroundContact::GroundContact(Endpoint & point, TScalar position,
                             TScalar direction, TScalar springConstant,
                             TScalar dampingConstant)
    :point(point)
    ,position(position)
    ,springConstant(springConstant)
    ,dampingConstant(dampingConstant)
{
    name = "GroundContact on ";
    name.append(point.getName());

    assert((direction == 1) || (direction == -1));

    if(direction > 0){
        this->direction = 1;
    } else {
        this->direction = -1;
    }
}

void GroundContact::applyForce()
{
    TScalar depth, speed, springforce;

    depth = direction * (point.getCoordinateFrame().r.y() - position);
    speed = point.getCoordinateFrame().v.y();
    springforce = depth * springConstant - (speed * dampingConstant);

    force = 0;
    // condassign fuer Kompatibilitaet mit ADOC_C
    // codassign (a, b, c, d) entspricht a = (b > 0) ? c : d;
    condassign(force, depth, force + springforce, force);

    point.setExternalForceTorque(TVector3(0, force, 0),
                                 TVector3::Zero());
}

TScalar GroundContact::getForce()
{
    return force;
}

void GroundContact::resetForce()
{
    point.setExternalForceTorque(TVector3::Zero(),
                                 TVector3::Zero());
}

const std::string & GroundContact::getName()const
{
    return name;
```

```
}
} // namespace mbslib
```

Bodenkontakt in mbslib: Testmodell

Listing 5.5: bodenkontakt_mp.cpp

```
#include <mbs/MbsCompoundWithBuilder.h>
#include "GroundContact.h"
#include <mbs/modeltools.h>

#include <iostream>
#include <iomanip>

int main(void)
{
    mbslib::Joint1DOF * j1;
    mbslib::Endpoint * boden;
    mbslib::GroundContact * gc;

    mbslib::MbsCompoundWithBuilder mbs("Bodenkontakt");

    mbs.addFixedBase("Anker");


    j1 = mbs.addPrismaticJoint(mbslib::TVector3(0, -1, 0));


    boden = mbs.addEndpoint(1,
                            mbslib::TMatrix3x3::Zero(),
                            "Bodenkontaktpunkt");


    // Bodenkontakt:
    gc = new mbslib::GroundContact(*boden, -1.0, -1.0, 5000.0, 100.0);
    mbs.addForceGenerator(*gc);

    assert(mbs.isValid());

    mbs.setGravitation(mbslib::TVector3(0, -9.81, 0));

    // Anfangsbedingungen:
    j1->setJointPosition(0);
    j1->setJointVelocity(0);
    j1->setJointAcceleration(0);

    mbs.doDirkin();

    double dt = 0.0005;
    double t = 3; //20
    double fps = 50;
    std::cout << "!PLfps=25!\n";
    int skip = 1/fps/dt;

    std::cout << std::setiosflags(std::ios::fixed)
              << std::setprecision(6);
```

```
50
51      std::cout << "\n#t\ty\tdy\n";
52
53      for (int i = 0; i <= t/dt; i++) {
54          if (i%skip==0) {
55              std::cout << "!PL!" << (i*dt) << "\t"
56                        << j1->getCoordinateFrame().r.x() << "\t"
57                        << j1->getCoordinateFrame().r.y() << "\t"
58                        << boden->getCoordinateFrame().r.x() << "\t"
59                        << boden->getCoordinateFrame().r.y()
60                        << std::endl;
61
62
63              std::cout << (i*dt) << "\t"
64                        << boden->getCoordinateFrame().r.y() << "\t"
65                        << boden->getCoordinateFrame().v.y() << std::endl;
66
67          }
68
69          mbs.doABA();
70          mbs.integrate(dt);
71      }
72
73      std::cout << "!PL!" << t << "\t"
74                << j1->getCoordinateFrame().r.x() << "\t"
75                << j1->getCoordinateFrame().r.y() << "\t"
76                << boden->getCoordinateFrame().r.x() << "\t"
77                << boden->getCoordinateFrame().r.y() //<< "\t"
78                << std::endl;
79
80      std::cout << (t) << "\t"
81                << boden->getCoordinateFrame().r.y() << "\t"
82                << boden->getCoordinateFrame().v.y() << std::endl;;
83
84      std::cout << "!PLO0set_marker=o,set_markerfacecolor=b!\n";
85      std::cout << "!PLtitle=Bodenkontakt␣in␣mbslib␣(bei␣-0.5)!\n";
86      std::cout << "!PLxlim=[-1.1,1.1];␣ylim=[-1.1,␣0.1]!\n";
87      return 0;
88  }
```

Dreifachpendel: Generierung der Gelenkwinkel

Listing 5.6: dreifachpendel_rnd.py

```
1  #!/usr/bin/env python2
2  # -- coding: utf-8 --
3  import random
4  from numpy import genfromtxt
5
6  min_ = 0
7  max_ = 0.00000001
8
9  logfilename = "dreifachpendel.rnd"
10 log = open(logfilename, "w")
11
12 def pr(str_):
```

```
    log.write(str_)
    log.write("\n")
    print str_

for i in range(0, 100):
    q1 = str(random.uniform(min_, max_))
    q2 = str(random.uniform(min_, max_))
    q3 = str(random.uniform(min_, max_))
    pr("%d %s %s %s"%(i, q1, q2, q3))

log.close()
```

Dreifachpendel in mbslib: Position nach 20 s

Listing 5.7: dreifachpendel_20s.cpp

```
#include <mbs/MbsCompoundWithBuilder.h>
#include <mbs/modeltools.h>

#include <iostream>
#include <iomanip>

int main(int argc, char* argv[])
{
    mbslib::TMatrix3x3 I = mbslib::makeInertiaTensorCylinderY(0.05, 1, 1);

    mbslib::MbsCompoundWithBuilder mbs("Dreifachpendel");

    mbs.addFixedBase("Anker");

    // Gelenk und Starrkoerper
    mbslib::Joint1DOF * j1 = mbs.addRevoluteJoint(mbslib::TVector3(0, 0, 1));
    mbs.addRigidLink(mbslib::TVector3(0, 1, 0),
                     mbslib::TVector3(0, -0.5, 0),
                     1,
                     I);

    mbslib::Joint1DOF * j2 = mbs.addRevoluteJoint(mbslib::TVector3(0, 0, 1));
    mbs.addRigidLink(mbslib::TVector3(0, 1, 0),
                     mbslib::TVector3(0, -0.5, 0),
                     1,
                     I);

    mbslib::Joint1DOF * j3 = mbs.addRevoluteJoint(mbslib::TVector3(0, 0, 1));
    mbs.addRigidLink(mbslib::TVector3(0, 1, 0),
                     mbslib::TVector3(0, -0.5, 0),
                     1,
                     I);

    // Endpunkt
    mbslib::Endpoint * e1 = mbs.addEndpoint("Ende");

    assert(mbs.isValid());

    mbs.setGravitation(mbslib::TVector3(0, -9.81, 0));

```

```
41      if (argc < 4)
42      {
43          std::cout << "Dieses␣Programm␣erwartet␣die␣Positionen␣der␣drei␣"
44                    << "Gelenke␣in␣Rad␣als␣Argument.";
45          return 1;
46      }
47
48      j1->setJointPosition(atof(argv[1]));
49      j2->setJointPosition(atof(argv[2]));
50      j3->setJointPosition(atof(argv[3]));
51
52      mbs.doDirkin();
53
54      double dt = 0.0005;
55      double t = 20;
56
57      for (int i = 0; i <= t/dt; i++) {
58          mbs.doABA();
59          mbs.integrate(dt);
60      }
61
62      std::cout << std::setiosflags(std::ios::fixed)
63                << std::setprecision(6);
64
65      // Koordinaten fuer Visualisierung
66      std::cout << j1->getCoordinateFrame().r.x() << "\t"
67                << j1->getCoordinateFrame().r.y() << "\t"
68                << j2->getCoordinateFrame().r.x() << "\t"
69                << j2->getCoordinateFrame().r.y() << "\t"
70                << j3->getCoordinateFrame().r.x() << "\t"
71                << j3->getCoordinateFrame().r.y() << "\t"
72                << e1->getCoordinateFrame().r.x() << "\t"
73                << e1->getCoordinateFrame().r.y()
74                << std::endl;
75
76      return 0;
77  }
```

Dreifachpendel in mbslib: Parallelisierte Aufrufe

Listing 5.8: dreifachpendel_run.py

```
1  #!/usr/bin/env python2
2  # -- coding: utf-8 --
3  import random
4  import subprocess
5  import threading
6
7  retval = []
8
9  log = open("dreifachpendel_run.dat", "w")
10 rnd = open("dreifachpendel.rnd", "r")
11
12 class Th(threading.Thread):
13   def __init__(self, params):
14     self.sp = None
```

```
15      self.params = params
16      threading.Thread.__init__(self)
17    def run(self):
18        self.sp = subprocess.check_output(self.params)
19    def get(self):
20      return self.sp.strip()
21
22  for i in range(0, 25):
23    th = []
24    for j in range(0, 4):
25      args = rnd.readline().split()
26      args[0] = '../qtcreator-build/src/mk/dreifachpendel_20s'
27
28      print "running", args
29      th.insert(j, Th(args))
30      th[j].start()
31
32    for j in range(0, 4):
33      th[j].join()
34      retval += [th[j].get()]
35      log.write(th[j].get())
36      log.write("\n")
37
38  print "Results:"
39  for rv in retval:
40      print rv
41
42  log.close()
```

Hinterbein in mbslib:

Listing 5.9: hinterbein_parameter.h

```
1  #ifndef HINTERBEIN_PARAMETER_H
2  #define HINTERBEIN_PARAMETER_H
3  #include <mbs/types.h>
4  #include <mbs/modeltools.h>
5
6  class hinterbein_parameter
7  {
8  public:
9      hinterbein_parameter();
10
11     mbslib::TScalar friction;
12     mbslib::TScalar muscle_damping;
13     mbslib::TScalar muscle_constant;
14     mbslib::TScalar radius;
15
16     mbslib::TScalar ground_constant;
17     mbslib::TScalar ground_damping;
18     mbslib::TScalar ground_pos;
19
20     mbslib::TScalar os_femoris_length;
21     mbslib::TScalar os_femoris_mass;
22     mbslib::TScalar os_femoris_com;
23     mbslib::TScalar os_femoris_gluteus;
```

```
    mbslib::TScalar fibula_tendo_achillis;

    mbslib::TScalar fibula_length;
    mbslib::TScalar fibula_mass;
    mbslib::TScalar fibula_com;
    mbslib::TScalar fibula_versus_lateralis;

    mbslib::TScalar tendo_achillis_constant;
    mbslib::TScalar tendo_achillis_damping;
    mbslib::TScalar tendo_achillis_mass;
    mbslib::TScalar tendo_achillis_com;

    mbslib::TScalar ossa_metatarsalia_length;
    mbslib::TScalar ossa_metatarsalia_mass;
    mbslib::TScalar ossa_metatarsalia_com;

    mbslib::TScalar sartorius_length;
    mbslib::TVector3 sartorius_anfang;

    mbslib::TScalar versus_lateralis_length;
    mbslib::TVector3 versus_lateralis_anfang;
    mbslib::TScalar versus_lateralis_pulley_size;

    mbslib::TScalar gluteus_length;
    mbslib::TVector3 gluteus_anfang;
    mbslib::TVector3 gluteus_rolle;

    mbslib::TScalar biceps_femoris_length;
    mbslib::TVector3 biceps_femoris_anfang;
};
#endif // HINTERBEIN_PARAMETER_H
```

Listing 5.10: hinterbein_parameter.h

```
#include "hinterbein_parameter.h"

hinterbein_parameter::hinterbein_parameter()
{
    //% Achtung: com immer in Vielfachen der Starrkoerperlaenge:
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    //% Gelenkreibung:
    friction = 20;
    //% Muskelstaerke
    muscle_constant = 100;
    //% Muskeldaempfung
    muscle_damping = 20;
    //% Fuer die Traegheitstensoren:
    radius = 0.05;
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    //% Boden
    ground_constant = 5000;
    ground_damping = 100;
    ground_pos = -2.2;
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    os_femoris_length = 1;
    os_femoris_mass = 1;
```

```
    os_femoris_com = 0.5;
    //%Position des Endes des Gluteus an Os femoris in Vielfachen der Laenge
    //%des Os femoris:
    os_femoris_gluteus = 0.15;
    //%Abstand von Fibula/Tibia und Tendo achillis in Vielfachen der Laenge
    //%des Os femoris:
    fibula_tendo_achillis = 0.1;
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    fibula_length = 1; //% auch Tendo achillis
    fibula_mass = 1;
    fibula_com = 0.5;
    //%Abstand des Endpunktes des Versus lateralis an Fibula/Tibia in
    //%Vielfachen der Laenge von Fibula/Tibia in Prozent (auf Seite des
    //%Ossa metatarsalia):
    fibula_versus_lateralis = 0.3;
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    //% Tendo Achillis
    tendo_achillis_constant = 10000;
    tendo_achillis_damping = 100;
    tendo_achillis_mass = 1;
    tendo_achillis_com = 0.5;
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    ossa_metatarsalia_length = 1;
    ossa_metatarsalia_mass = 1;
    ossa_metatarsalia_com = 0.5;
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sartorius_length = 1.2;
    sartorius_anfang = mbslib::TVector3(-0.5, 0.1, 0);
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    versus_lateralis_length = 1.98;
    versus_lateralis_anfang = mbslib::TVector3(-0.4, 0.2, 0);
    //% Groesse der Umlenkrolle:
    versus_lateralis_pulley_size = 0.01;
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    gluteus_length = 0.76;
    gluteus_anfang = mbslib::TVector3(-0.3, 0.1, 0);
    gluteus_rolle = mbslib::TVector3(0.2, 0, 0);
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    biceps_femoris_length = 1.42;
    biceps_femoris_anfang = mbslib::TVector3(0.1, 0.2, 0);
}
```

Listing 5.11: hinterbein_modell.h

```
#ifndef HINTERBEIN_MODELL_H
#define HINTERBEIN_MODELL_H
#include <mbs/MbsCompoundWithBuilder.h>
#include <mbs/LinearSpringModel.h>
#include <mbs/LinearSpringWithRopeModel.h>
#include "GroundContact.h"
#include <iostream>
#include <iomanip>
#include "hinterbein_parameter.h"

class hinterbein_modell
{
```

```
13  public:
14      hinterbein_modell(hinterbein_parameter p = hinterbein_parameter());
15
16      void printCoordinateLines(float t);
17      void printLineSettings();
18      void printMuscleLengths();
19
20      mbslib::MbsCompoundWithBuilder * mbs;
21      mbslib::Joint1DOF * R1;
22      mbslib::Joint1DOF * R2;
23      mbslib::Joint1DOF * R3;
24      mbslib::Joint1DOF * R4;
25
26      mbslib::Spring3D * bicepsFemoris;
27      mbslib::Endpoint * bicepsFemorisAnfang;
28      mbslib::Endpoint * bicepsFemorisEnde;
29
30      mbslib::Spring3D * versusLateralis;
31      mbslib::Endpoint * versusLateralisAnfang;
32      mbslib::Endpoint * versusLateralisMitte;
33      mbslib::Endpoint * versusLateralisEnde;
34
35      mbslib::Spring3D * sartorius;
36      mbslib::Endpoint * sartoriusAnfang;
37      mbslib::Endpoint * sartoriusEnde;
38
39      mbslib::Endpoint * tendoAchillisAnfang;
40      mbslib::Endpoint * tendoAchillisEnde;
41
42      mbslib::Spring3D * gluteus;
43      mbslib::Endpoint * gluteusAnfang;
44      mbslib::Endpoint * gluteusMitte;
45      mbslib::Endpoint * gluteusEnde;
46
47      mbslib::Endpoint * ossaMetatarsaliaAnfang;
48
49      mbslib::Endpoint * bodenkontakt;
50      mbslib::GroundContact * boden;
51  };
52
53  #endif // HINTERBEIN_MODELL_H
```

Listing 5.12: hinterbein_modell.cpp

```
1   #include "hinterbein_modell.h"
2
3   hinterbein_modell::hinterbein_modell(hinterbein_parameter p)
4   {
5       // Lokale Hilfsvariablen
6       mbslib::TScalar fibula_com_x;
7       mbslib::TScalar os_femoris_com_y;
8       mbslib::TScalar gluteus_y;
9       mbslib::TScalar dist_tendo_achillis_fibula;
10      mbslib::TScalar versus_lateralis_x;
11      mbslib::TScalar ossa_y;
12      mbslib::TScalar ossa_com_y;
```

```
13      mbslib::TVector3 pulley;
14
15      // Traegheitstensoren
16      mbslib::TMatrix3x3 os_femoris_I, fibula_I,
17              tendo_achillis_I, ossa_metatarsalia_I;
18
19
20      os_femoris_I = mbslib::makeInertiaTensorCylinderY(
21                              p.radius,
22                              p.os_femoris_length,
23                              p.os_femoris_mass);
24      fibula_I = mbslib::makeInertiaTensorCylinderX(
25                              p.radius,
26                              p.fibula_length,
27                              p.fibula_mass);
28      tendo_achillis_I = mbslib::makeInertiaTensorCylinderX(
29                              p.radius,
30                              p.fibula_length,
31                              p.tendo_achillis_mass);
32      ossa_metatarsalia_I = mbslib::makeInertiaTensorCylinderY(
33                              p.radius,
34                              p.ossa_metatarsalia_length,
35                              p.ossa_metatarsalia_mass);
36
37
38      mbs = new mbslib::MbsCompoundWithBuilder("Hinterbein");
39      mbs->addFixedBase("Anker");
40
41      // ****************************************
42      // Anfangspunkte der Seilzuege:
43      // ****************************************
44
45      // Anfang biceps femoris:
46      mbs->addFork();
47      mbs->addFixedTranslation(p.biceps_femoris_anfang);
48      bicepsFemorisAnfang = mbs->addEndpoint("Anfang biceps femoris");
49
50      // Anfang gluteus:
51      mbs->addFork();
52      mbs->addFixedTranslation(p.gluteus_anfang);
53      gluteusAnfang = mbs->addEndpoint("Anfang gluteus");
54
55
56      // Anfang versus lateralis:
57      mbs->addFork();
58      mbs->addFixedTranslation(p.versus_lateralis_anfang);
59      versusLateralisAnfang = mbs->addEndpoint("Anfang versus lateralis");
60
61      // Anfang sartorius:
62      mbs->addFork();
63      mbs->addFixedTranslation(p.sartorius_anfang);
64      sartoriusAnfang = mbs->addEndpoint("Anfang sartorius");
65
66      // Umlenkrolle gluteus:
67      mbs->addFork();
```

```
68      mbs->addFixedTranslation(p.gluteus_rolle);
69      gluteusMitte = mbs->addEndpoint("Mitte gluteus");
70
71      // ****************************************
72      // Gelenke und Knochen:
73      // ****************************************
74      // Drehgelenk
75      R1 = mbs->addRevoluteJoint(mbslib::TVector3(0,0,1), "R1");
76      R1->setParameter(0, p.friction); // jointFriction
77
78      // ============
79      // os femoris:
80      // ============
81      os_femoris_com_y = p.os_femoris_length * (1 - p.os_femoris_com);
82      mbs->addRigidLink(mbslib::TVector3(0, -1 * p.os_femoris_length, 0),
83                        mbslib::TVector3(0, os_femoris_com_y, 0),
84                        p.os_femoris_mass,
85                        os_femoris_I,
86                        "os femoris");
87
88      // Ende gluteus an os femoris:
89      mbs->addFork();
90
91      gluteus_y = (1-p.os_femoris_gluteus)*p.os_femoris_length;
92      mbs->addFixedTranslation(mbslib::TVector3(0, gluteus_y, 0));
93      gluteusEnde = mbs->addEndpoint("Endpunkt gluteus");
94
95      // Anfang tendo achillis an os femoris:
96      mbs->addFork();
97      dist_tendo_achillis_fibula = p.fibula_tendo_achillis*p.os_femoris_length;
98      mbs->addFixedTranslation(mbslib::TVector3(0,
99                                                dist_tendo_achillis_fibula,
100                                               0));
101
102     // Trick mit Drehgelenk und Feder (offene kinematische Kette):
103     R4 = mbs->addRevoluteJoint(mbslib::TVector3(0, 0, 1), "R2");
104     R4->setParameter(0, p.friction); // jointFriction
105
106     fibula_com_x = p.fibula_length * (p.tendo_achillis_com - 1);
107     mbs->addRigidLink(mbslib::TVector3(p.fibula_length, 0, 0),
108                       mbslib::TVector3(fibula_com_x, 0, 0),
109                       p.tendo_achillis_mass,
110                       tendo_achillis_I,
111                       "tendo achillis");
112
113     tendoAchillisAnfang = mbs->addEndpoint("Anfang tendo achillis");
114
115
116     // Mitte versus lateralis an os femoris:
117     mbs->addFork();
118
119     mbs->addFork();
120     sartoriusEnde = mbs->addEndpoint("Ende sartorius");
121
122     pulley = mbslib::TVector3(-1*p.versus_lateralis_pulley_size,
```

```
123                                 -1*p.versus_lateralis_pulley_size,
124                                 0);
125     mbs->addFixedTranslation(pulley, "Radersatz");
126     versusLateralisMitte = mbs->addEndpoint("Mitte versus lateralis");
127
128     R2 = mbs->addRevoluteJoint(mbslib::TVector3(0, 0, 1), "R2");
129     R2->setParameter(0, p.friction); // jointFriction
130
131     // ============
132     // fibula/tibia
133     // ============
134     mbs->addRigidLink(mbslib::TVector3(p.fibula_length, 0, 0),
135                       mbslib::TVector3(p.fibula_length * (p.fibula_com - 1),
136                                        0,
137                                        0),
138                       p.fibula_mass,
139                       fibula_I,
140                       "fibula/tibia");
141
142     // Ende versus lateralis an fibula/tibula:
143     mbs->addFork();
144     versus_lateralis_x = -1 * p.fibula_versus_lateralis * p.fibula_length;
145     mbs->addFixedTranslation(mbslib::TVector3(versus_lateralis_x, 0, 0));
146
147     versusLateralisEnde = mbs->addEndpoint("Endpunkt versus lateralis");
148
149
150     R3 = mbs->addRevoluteJoint(mbslib::TVector3(0, 0, 1), "R3");
151     R3->setParameter(0, p.friction); // jointFriction
152
153     // ============
154     // ossa metatarsalia
155     // ============
156     // Fuer die Visualisierung wird der folgende Punkt benoetigt:
157     mbs->addFork();
158     ossaMetatarsaliaAnfang = mbs->addEndpoint("ossa_Anfang");
159
160     // Ende tendo achillis an ossa metatarsalia:
161     mbs->addFork();
162     mbs->addFixedTranslation(mbslib::TVector3(0,
163                                               dist_tendo_achillis_fibula,
164                                               0));
165     tendoAchillisEnde = mbs->addEndpoint("Ende tendo achillis");
166     bicepsFemorisEnde = tendoAchillisEnde;
167
168
169     ossa_y = -1 * p.ossa_metatarsalia_length + dist_tendo_achillis_fibula;
170     ossa_com_y = (1 - p.ossa_metatarsalia_com) * p.ossa_metatarsalia_length;
171     mbs->addRigidLink(mbslib::TVector3(0, ossa_y, 0),
172                       mbslib::TVector3(0, ossa_com_y, 0),
173                       p.ossa_metatarsalia_mass,
174                       ossa_metatarsalia_I,
175                       "ossa metatarsalia");
176
177
```

```
178
179     // Endpunkt
180     bodenkontakt = mbs->addEndpoint("Bodenkontaktpunkt");
181
182     assert(mbs->isValid());
183
184     // ============
185     // Boden:
186     // ============
187     boden = new mbslib::GroundContact(*bodenkontakt,
188                                       p.ground_pos,
189                                       -1.0,
190                                       p.ground_constant,
191                                       p.ground_damping);
192     mbs->addForceGenerator(*boden);
193
194
195     // ============
196     // tendo achillis (Verbindung)
197     // ============
198     mbslib::LinearSpringModel tendoAchillis_springModel(
199                                  p.tendo_achillis_constant,
200                                  p.tendo_achillis_damping);
201     mbslib::Spring3D * tendoAchillis = mbs->addSpring(
202                                           tendoAchillis_springModel,
203                                           "tendo achillis");
204     (* tendoAchillis) << tendoAchillisAnfang << tendoAchillisEnde;
205
206
207     // ****************************************
208     // Muskeln:
209     // ****************************************
210
211     // ============
212     // sartorius
213     // ============
214     mbslib::LinearSpringWithRopeModel muscleModel(p.muscle_constant,
215                                                   p.muscle_damping);
216     sartorius = mbs->addSpring(muscleModel, "sartorius");
217     (* sartorius) << sartoriusAnfang
218                  << sartoriusEnde;
219     sartorius->setParameter(2, p.sartorius_length);
220
221     // ============
222     // versus lateralis
223     // ============
224     versusLateralis = mbs->addSpring(muscleModel, "versus lateralis");
225     (* versusLateralis) << versusLateralisAnfang
226                         << versusLateralisMitte
227                         << versusLateralisEnde;
228     versusLateralis->setParameter(2, p.versus_lateralis_length);
229
230     // ============
231     // gluteus
232     // ============
```

```
233     gluteus = mbs->addSpring(muscleModel, "gluteus");
234     (* gluteus) << gluteusAnfang
235               << gluteusMitte
236               << gluteusEnde;
237     gluteus->setParameter(2, p.gluteus_length);
238 
239     // ============
240     // biceps femoris
241     // ============
242     bicepsFemoris = mbs->addSpring(muscleModel, "biceps femoris");
243     (* bicepsFemoris) << bicepsFemorisAnfang
244                     << bicepsFemorisEnde;
245     bicepsFemoris->setParameter(2, p.biceps_femoris_length);
246 
247     // ****************************************
248     // Sonstiges:
249     // ****************************************
250     mbs->setGravitation(mbslib::TVector3(0, -9.81, 0));
251 }
252 
253 void hinterbein_modell::printCoordinateLines(float t)
254 {
255 
256     std::cout << "!PL0!" << t << "\t"
257               << std::setprecision(6)
258               << R1->getCoordinateFrame().r.x() << "\t"
259               << R1->getCoordinateFrame().r.y() << "\t"
260               << R2->getCoordinateFrame().r.x() << "\t"
261               << R2->getCoordinateFrame().r.y() << "\t"
262               << R3->getCoordinateFrame().r.x() << "\t"
263               << R3->getCoordinateFrame().r.y() << "\t"
264               << bodenkontakt->getCoordinateFrame().r.x() << "\t"
265               << bodenkontakt->getCoordinateFrame().r.y() //<< "\t"
266               << std::endl;
267 
268     std::cout << "!PL1!" << t << "\t"
269               << std::setprecision(6)
270               << R4->getCoordinateFrame().r.x() << "\t"
271               << R4->getCoordinateFrame().r.y() << "\t"
272               << tendoAchillisEnde->getCoordinateFrame().r.x() << "\t"
273               << tendoAchillisEnde->getCoordinateFrame().r.y() << "\t"
274               << ossaMetatarsaliaAnfang->getCoordinateFrame().r.x() << "\t"
275               << ossaMetatarsaliaAnfang->getCoordinateFrame().r.y()
276               << std::endl;
277 
278     std::cout << "!PL2!" << t << "\t"
279               << std::setprecision(6)
280               << sartoriusAnfang->getCoordinateFrame().r.x() << "\t"
281               << sartoriusAnfang->getCoordinateFrame().r.y() << "\t"
282               << sartoriusEnde->getCoordinateFrame().r.x() << "\t"
283               << sartoriusEnde->getCoordinateFrame().r.y()
284               << std::endl;
285 
286     std::cout << "!PL3!" << t << "\t"
287               << std::setprecision(6)
```

```
288                 << bicepsFemorisAnfang->getCoordinateFrame().r.x() << "\t"
289                 << bicepsFemorisAnfang->getCoordinateFrame().r.y() << "\t"
290                 << bicepsFemorisEnde->getCoordinateFrame().r.x() << "\t"
291                 << bicepsFemorisEnde->getCoordinateFrame().r.y()
292                 << std::endl;
293
294     std::cout << "!PL4!" << t << "\t"
295               << std::setprecision(6)
296               << versusLateralisAnfang->getCoordinateFrame().r.x() << "\t"
297               << versusLateralisAnfang->getCoordinateFrame().r.y() << "\t"
298               << versusLateralisMitte->getCoordinateFrame().r.x() << "\t"
299               << versusLateralisMitte->getCoordinateFrame().r.y() << "\t"
300               << versusLateralisEnde->getCoordinateFrame().r.x() << "\t"
301               << versusLateralisEnde->getCoordinateFrame().r.y()
302               << std::endl;
303
304     std::cout << "!PL5!" << t << "\t"
305               << std::setprecision(6)
306               << gluteusAnfang->getCoordinateFrame().r.x() << "\t"
307               << gluteusAnfang->getCoordinateFrame().r.y() << "\t"
308               << gluteusMitte->getCoordinateFrame().r.x() << "\t"
309               << gluteusMitte->getCoordinateFrame().r.y() << "\t"
310               << gluteusEnde->getCoordinateFrame().r.x() << "\t"
311               << gluteusEnde->getCoordinateFrame().r.y()
312               << std::endl;
313 }
314
315 void hinterbein_modell::printLineSettings()
316 {
317     std::cout << "!PLO0set_marker=o,set_markerfacecolor=b!\n";
318     std::cout << "!PLO1set_marker=o,set_markerfacecolor=b!\n";
319     std::cout << "!PLO2set_linewidth=1,set_color=r,set_linestyle=dashed!\n";
320     std::cout << "!PLO3set_linewidth=1,set_color=r,set_linestyle=dashed!\n";
321     std::cout << "!PLO4set_linewidth=1,set_color=r,set_linestyle=dashed!\n";
322     std::cout << "!PLO5set_linewidth=1,set_color=r,set_linestyle=dashed!\n";
323 }
324
325 void hinterbein_modell::printMuscleLengths()
326 {
327     mbslib::TScalar sartorius, versus, gluteus, biceps;
328
329     sartorius = (sartoriusAnfang->getCoordinateFrame().r -
330                  sartoriusEnde->getCoordinateFrame().r ).norm();
331
332     versus = (versusLateralisAnfang->getCoordinateFrame().r -
333               versusLateralisMitte->getCoordinateFrame().r).norm() +
334              (versusLateralisMitte->getCoordinateFrame().r -
335               versusLateralisEnde->getCoordinateFrame().r).norm();
336
337     gluteus = (gluteusAnfang->getCoordinateFrame().r -
338                gluteusMitte->getCoordinateFrame().r).norm() +
339               (gluteusMitte->getCoordinateFrame().r -
340                gluteusEnde->getCoordinateFrame().r).norm();
341
342     biceps = (bicepsFemorisAnfang->getCoordinateFrame().r -
```

```
343                 bicepsFemorisEnde->getCoordinateFrame().r).norm();
344
345     std::cout << "\ngeometrische Laengen der Muskeln:\n";
346     std::cout << "  Sartorius: " << sartorius << std::endl;
347     std::cout << "  Versus lateralis: " << versus << std::endl;
348     std::cout << "  Gluteus: " << gluteus << std::endl;
349     std::cout << "  biceps femoris: " << biceps << std::endl;
350 }
```

Listing 5.13: hinterbein.cpp

```
1  #include <iostream>
2  #include <iomanip>
3
4  int main(int argc, char* argv[])
5  {
6      hinterbein_parameter param;
7      hinterbein_modell hb = hinterbein_modell(param);
8
9      hb.mbs->doDirkin(); // damit der erste Frame korrekt ist.
10
11     double dt = 0.0005;
12     double t = 1;//20;
13     double fps = 25;
14     std::cout << "!PLfps=25!\n";
15     int skip = 1/fps/dt;
16
17     std::cout << std::setiosflags(std::ios::fixed)
18               << std::setprecision(6);
19     std::cout << "\n#t\tq1\tdq1\tq2\tdq2\tq3\tdq3\tq4\tdq4\tdist\n";
20
21     float dist;
22
23     for (int i = 0; i <= t/dt; i++) {
24         if (i%skip==0) {
25             dist = (hb.tendoAchillisAnfang->getCoordinateFrame().r -
26                     hb.tendoAchillisEnde->getCoordinateFrame().r).norm();
27             std::cout << (i*dt) << "\t"
28                       << (hb.R1->getJointPosition()/M_PI*180) << "\t"
29                       << (hb.R1->getJointVelocity()/M_PI*180) << "\t"
30                       << (hb.R2->getJointPosition()/M_PI*180) << "\t"
31                       << (hb.R2->getJointVelocity()/M_PI*180) << "\t"
32                       << (hb.R3->getJointPosition()/M_PI*180) << "\t"
33                       << (hb.R3->getJointVelocity()/M_PI*180) << "\t"
34                       << (hb.R4->getJointPosition()/M_PI*180) << "\t"
35                       << (hb.R4->getJointVelocity()/M_PI*180) << "\t"
36                       << dist
37                       << std::endl;
38
39             hb.printCoordinateLines(i*dt);
40         }
41         // Simuliere die Dynamik:
42         hb.mbs->doABA();
43         hb.mbs->integrate(dt);
44     }
45     dist = (hb.tendoAchillisAnfang->getCoordinateFrame().r -
```

```
            hb.tendoAchillisEnde->getCoordinateFrame().r).norm();

    std::cout << (t) << "\t"
              << (hb.R1->getJointPosition()/M_PI*180) << "\t"
              << (hb.R1->getJointVelocity()/M_PI*180) << "\t"
              << (hb.R2->getJointPosition()/M_PI*180) << "\t"
              << (hb.R2->getJointVelocity()/M_PI*180) << "\t"
              << (hb.R3->getJointPosition()/M_PI*180) << "\t"
              << (hb.R3->getJointVelocity()/M_PI*180) << "\t"
              << (hb.R4->getJointPosition()/M_PI*180) << "\t"
              << (hb.R4->getJointVelocity()/M_PI*180) << "\t"
              << dist
              << std::endl;
    hb.printCoordinateLines(t);

    hb.printLineSettings();

    std::cout << "!PLtitle=Hinterbein␣in␣mbslib!\n"
              << "!PLtitle2=(offene␣kinematische␣Kette)!\n";
    return 0;
}
```

Optimierung:

Listing 5.14: hinterbein_opt.cpp

```
#include "hinterbein_modell.h"

#include <iostream>
#include <iomanip>

int main(int argc, char* argv[])
{
    if (argc < 3)
    {
        std::cout << "Dieses␣Programm␣erwartet␣die␣Werte␣fuer␣"
                  << "fibula_versus_lateralis␣und␣os_femoris_gluteus.";
        return 1;
    }

    hinterbein_parameter param;
    param.fibula_versus_lateralis = atof(argv[1]);
    param.os_femoris_gluteus = atof(argv[2]);
    param.versus_lateralis_length -= 0.5;
    param.gluteus_length -= 0.1;

    hinterbein_modell hb = hinterbein_modell(param);

    hb.R1->setJointPosition(-20.7 * M_PI / 180);
    hb.R2->setJointPosition(-4.57 * M_PI / 180);
    hb.R3->setJointPosition(4.12 * M_PI / 180);
    hb.R4->setJointPosition(-4.59 * M_PI / 180);

    hb.mbs->doDirkin();

    double dt = 0.0005;
```

```
31      double t = 2;
32      double fps = 25;
33      int skip = 1/fps/dt;
34
35      std::cout << std::setiosflags(std::ios::fixed)
36                << std::setprecision(6);
37
38      mbslib::TScalar max = 0;
39
40      for (int i = 0; i <= t/dt; i++) {
41          // Simuliere die Dynamik:
42          hb.mbs->doABA();
43          hb.mbs->integrate(dt);
44
45          if (i%skip==0) {
46              if (hb.boden->getForce() > max)
47                  max = hb.boden->getForce();
48          }
49      }
50
51      if (max!=0)
52          std::cout << 1/max;
53      else
54          std::cout << 2000000;
55      return 0;
56  }
```