hochschule mannheim

test First line
second line title

Daniel Koch

Bachelor-Thesis

Studiengang Informatik

Fakultät für Informatik

Hochschule Mannheim

22.07.2020

Betreuer

Prof. Dr. Thomas Ihme, Hochschule Mannheim

Koch, Daniel:
TEST / Daniel Koch. –
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2020. 37 Seiten.

Koch, Daniel:
TEST / Daniel Koch. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2020. 37 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 22.07.2020

Daniel Koch

Abstract

TEST

TEST.

TEST

TEST.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel der Arbeit . . . 2
- 1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **5**
- 2.1 Aufbau des Laufroboters . . . 5
- 2.2 Robotik . . . 5
  - 2.2.1 Koordinatensysteme . . . 5
  - 2.2.2 Direkte Kinematik . . . 5
  - 2.2.3 Inverse Kinematik . . . 5
  - 2.2.4 Laufplanung . . . 5
- 2.3 Frameworks . . . 6
  - 2.3.1 Robot Operating System . . . 6
  - 2.3.2 Gazebo . . . 6
  - 2.3.3 MeshLab . . . 6

3 Random Sampling **7**
- 3.1 Kriterien zur Auswahl des Algorithmus . . . 7
- 3.2 Erzeugen gültiger Lösungen . . . 9
  - 3.2.1 Festlegung des Pfades zum Ziel . . . 9
  - 3.2.2 Schrittweise Näherung an das Ziel . . . 11
  - 3.2.3 Berechnungen des Mittelpunkts und der Bewegungsdauer . . . 15
- 3.3 Bewertung gültiger Lösungen . . . 15

4 Portierung des Laufplaners nach ROS und Gazebo **17**
- 4.1 Analyse bestehender Laufplaner . . . 17
- 4.2 Vorgehensweise und Aufbau des Pakets . . . 19
- 4.3 Aufsetzen der Simulation . . . 20
  - 4.3.1 Aufsetzen des Robotermodells mittels urdf . . . 20
  - 4.3.2 Definition der Gelenkmotoren mittels ros_control . . . 24
  - 4.3.3 Aufsetzen der Umgebung mittels Gazebo . . . 24
- 4.4 Aufsetzen der Ausgangsposition . . . 26
- 4.5 Generieren und Einlesen von Bewegungen als xml-Datei . . . 27

4.6 Abspielen der Bewegungen 27
4.7 Aufsetzen des Dreifußgangs 29
4.8 Aufsetzen des Random Samplings 30
4.8.1 Abstrahierung der Fußkonfiguration 30
4.8.2 Anpassung aller Maße 30
4.8.3 Nutzung des TF-Frameworks 30
4.8.4 Veränderung der Randomisierung 31

5 Testen der Ergebnisse **33**
5.1 Testen der Fußsteuerung 33

6 Zusammenfassung **35**

7 Ausblick **37**

Abkürzungsverzeichnis **vii**

Tabellenverzeichnis **ix**

Abbildungsverzeichnis **xi**

Quellcodeverzeichnis **xiii**

Literatur **xv**

Kapitel 1

Einleitung

1.1 Motivation

Die einfachste Art einen Laufroboter zu steuern, ist die Anwendung eines *statischen Laufmusters* wie dem Dreifußgang. Diese Laufmuster funktionieren auf ebenem Untergrund für beispielsweise sechsbeinige Laufroboter sehr gut und stabil. Anders sieht es allerdings aus, wenn das Gelände Hindernisse aufweist. Dann macht es in erster Linie Sinn, zusätzlich zu dem statischen Laufmuster mit einem *reaktiven Laufmuster* auf die Umgebung zu reagieren. Ist ein Gelände allerdings stark uneben und besteht eventuell sogar aus unüberwindbaren Abschnitten, würde das reaktive Laufmuster dominieren. Der Laufroboter würde sich nur noch langsam mittels des reaktiven Laufmusters nach vorne tasten.

Planende Verfahren verbessern diesen Prozess, indem sie Lösungen für unebenes Gelände schon vor oder während des Laufens berechnen und daher nicht mehr darauf reagieren müssen, sondern aktiv einen gültigen Weg zu Ziel planen. Dies ist in Fällen wie der Bergung von Opfern in einem unebenen Gelände oder einem Kernkraftwerk-Unfall der Fall, in dem in unterschiedlichsten Umgebungen gelaufen werden muss. Aber auch bei Service-Robotern könnte eine solche Planung sinnvoll sein.

Diese Arbeit nutzt ein bestehendes planendes Verfahren und portiert dieses auf eine Umgebung im Robot Operating System (ROS) für den sechsbeinigen Laufroboter Akrobat.

1.2 Ziel der Arbeit

André Herms [1] hat initial verschiedene Algorithmen zur Laufplanung analysiert und eine Auswahl in einer bestehenden OpenInventor-Umgebung [2] für den sechsbeinigen Laufroboter *LAURON* aufgesetzt. Dabei hat sich aus einer Auswahl von sieben Algorithmen und den Auswahlkriterien Anytime-Fähigkeit, Parallelisierbarkeit, Speicherbedarf und Anwendbarkeit das Random Sampling als beste Wahl herausgestellt. Uli Ruffler [3] hat diesen Algorithmus auf eine inkrementelle Funktionsweise weiterentwickelt sowie weitere Anpassungen vorgenommen.

Das Ziel dieser Arbeit ist nun diesen Laufplaner für den *Akrobat* bereitzustellen. Da dieser Laufroboter auf ROS aufgesetzt ist, muss die OpenInventor-Umgebung migriert werden. Da es sich um ein anderes Robotermodell handelt, müssen auch weitere Anpassungen vorgenommen werden.

Als Simulationsumgebung bietet sich Gazebo an, da Gazebo eine Physik-Engine bereitstellt. Später soll es einfach möglich sein, zwischen der Simulationsumgebung und der realen Ausführung zu wechseln. Alle Ergebnisse sollen in einem ROS-Paket gebündelt werden.

1.3 Aufbau der Arbeit

Die Arbeit beginnt in Kapitel 2 mit den Grundlagen, die für die Portierung des Laufplaners in das ROS und Gazebo nötig sind. Dabei geht die Arbeit auf den Laufroboter, auf Koordinatensysteme, direkte und inverse Kinematik sowie Grundlagen zur Laufplanung ein. Außerdem werden die benötigten Frameworks wie ROS, Gazebo und MeshLab vorgestellt.

Nach den Grundlagen folgt in Kapitel 3 die konzeptionelle Darstellung des Algorithmus Random Sampling, welcher zur Laufplanung genutzt werden soll. Zunächst wird der Algorithmus mit weiteren Algorithmen verglichen und dann detailliert erklärt. Das Kapitel geht darauf ein, wie gültige Lösungen generiert, aber auch bewertet werden können.

Kapitel 4 analysiert existierende Laufplaner und stellt einen Ansatz dar, diese in ein ROS-Paket, welches Gazebo als Simulationsumgebung nutzt, zu migrieren. Da die vorherigen Laufplaner über die 3D-Bibliothek OpenInventor laufen, sind eini-

ge Anpassungen nötig, damit der Laufplaner in Gazebo funktionieren kann. Die Ergebnisse werden in Kapitel 5 getestet.

Kapitel 6 fasst alle Ergebnisse zusammen und Kapitel 7 gibt einen Ausblick darüber, wie der Laufplaner in Zukunft weiterentwickelt werden könnte.

Kapitel 2

Grundlagen

2.1 Aufbau des Laufroboters

2.2 Robotik

2.2.1 Koordinatensysteme

2.2.2 Direkte Kinematik

- Mit gegebener Stellung der Gelenke Position und Orientierung des Endeffektes zu berechnen - Transformationsmatrix, die dann aufgelöst wird und berechnet werden kann

fellmann zitieren

2.2.3 Inverse Kinematik

- Durch Position und Orientierung des Endeffektes mögliche Stellung der Gelenke zu berechnen (meist mehrere Lösungen) - analytische Berechnung - numerische Berechnung: lineare Annäherung durch Jacobi-Matrix

fellmann zitieren

2.2.4 Laufplanung

statische Laufalgorithmen, reaktive, planende Laufalgorithmen like RandomSampling

2.3 Frameworks

2.3.1 Robot Operating System

2.3.2 Gazebo

2.3.3 MeshLab

Kapitel 3

Random Sampling

Dieses Kapitel geht auf die konzeptionellen Aspekte des Random Samplings ein. Zunächst betrachtet die Arbeit, weshalb der Algorithmus die beste Wahl für den Laufplaner ist. Darauf folgt die Darstellung der Funktionsweise des Random Samplings, welches große Mengen an zufälligen und gültigen Lösungen generiert. Dabei sind zwei Fragen von großer Bedeutung. Zum einen ist das: Wie können zufällige gültige Lösungen generiert werden. Wenn diese Lösungen vorliegen ist die nächste wichtige Frage: Wie können diese Lösungen bewertet werden?

Das Kapitel basiert auf der Arbeit von André Herms [1], der verschiedene planende Verfahren vorstellt und mit geeigneten Kriterien bewertet. Da seine Arbeit zusammenfassend davon ausgeht, dass Random Sampling die nach seinen Kriterien beste Wahl darstellt, nutzt auch diese Arbeit den Algorithmus.

3.1 Kriterien zur Auswahl des Algorithmus

Herms stellt in seiner Arbeit neben dem Random Sampling die folgenden Algorithmen vor:

- *Greedy Verfahren:* Es werden nacheinander Teillösungen durch ein lokales Kriterium generiert. Teillösungen werden nicht mehr verworfen. Daher kommt auch der Begriff „greedy“, da der Algorithmus gierig ist und somit Lösungen nicht wieder verwirft.

- *Branch and Bound:* Das Verfahren liefert durch einen modifzierten Backtracking-Ansatz immer die exakte Lösung. Anders als beim Backtracking werden die Suchbäume gestutzt, so dass die Laufzeit dadurch verkürzt wird.
- *Lokale Suche:* Das Verfahren ist eine modifzierte Variante des Random Samplings. Neben der zufälligen Generierung von Lösungen wird auch iterativ in der Nachbarschaft nach einem besseren Punkt geschaut.
- *Tabu-Suche:* Das Verfahren ist eine Erweiterung der lokalen Suche. Der Nachbar muss nicht zwingend besser bewertet sein, um als Nachfolgepunkt akzeptiert zu werden. Damit bleibt der Algorithmus nicht in lokalen Maxima hängen.
- *Simulated Annealing:* Das Verfahren ist eine erneute Verbesserung der lokalen Suche und der Tabu-Suche, das sich hinsichtlich des Optimierungsproblems an dem physikalischen Prozess des langsamen Abkühlen fester Stoffe orientiert.
- *Genetische Algorithmen:* Das Verfahren nutzt die Mechanismen der natürlichen Evolution und wendet diese auf das Problem der Laufplanung an.

Die genannten Algorithmen vergleicht Herms nun unter Berücksichtigung der folgenden Kriterien:

- Parallelisierbarkeit
- Speicherbedarf
- Anytime-Fähigkeit
- Allgemeine Anwendbarkeit

Dabei stellt sich heraus, dass nur die Algorithmen *Random Sampling*, *Lokale Suche* und *Simulated Annealing* alle Kriterien mindestens erfüllen. Nach der Implementierung und erster Tests stellt sich heraus, dass das Random Sampling die Kriterien am besten erfüllt, da die Lokale Suche sowie das Simulated Annealing die Lösungen zu langsam erstellen. Tabelle 3.1 zeigt die weiteren Kriterien, die für die Bewertung des Algorithmus wichtig sind. Auch hier schneidet das Random Sampling positiv ab.

Zusammenfassend ist nun festzuhalten, dass das Random Sampling für den Laufplaner genutzt werden soll, da es zwischen allen Algorithmen am besten abschneidet.

Tabelle 3.1: Bewertung des Random Samplings

Kriterium	Erfüllung
Parallelisierbarkeit	Lösungen können ohne großen Aufwand unabhängig voneinander generiert werden. Am Ende müssen diese nur noch synchronisiert werden, so dass die Lösung mit der besten Bewertung übernommen wird.
Speicherbedarf	Der Algorithmus benötigt kaum Speicher, da immer nur eine Lösung generiert wird und diese die vorherige beste Lösung überschreibt.
Anytime	Das Kriterium ist erfüllt, sobald eine Lösung generiert ist. Allerdings kann es passieren, dass in einer bestimmten Zeit noch keine gute Lösung vorhanden ist.
Anwendbarkeit	Der Algorithmus lässt sich auf jedes Problem der Laufplanung anwenden. Die Qualität hängt allerdings vom Anteil guter Lösungen, die im Lösungsraum vorhanden sind. Außerdem spielt die konkrete Implementierung des Algorithmus eine große Rolle. Insgesamt gilt, dass je länger der Algorithmus läuft, desto besser sind potentiell die Lösungen.

3.2 Erzeugen gültiger Lösungen

Der Algorithmus zur Erzeugung gültiger Lösungen für die Gesamtstrecke wurde initial von André Herms [1] entwickelt. Uli Ruffler [3] hat diesen in seiner Arbeit unter anderem dahingehend weiterentwickelt, dass dieser auf einer inkrementellen Arbeitsweise läuft. Der folgende Abschnitt geht nun auf das grundlegende Konzept der Generierung von gültigen Lösungen ein.

Wichtig ist, dass alle möglichen Lösungen generiert werden können und nur Lösungen ausgeschlossen werden, die außerhalb des gültigen Bereichs der Fußwinkel und Fußgeschwindigkeiten liegen, die auf instabilem Untergrund wie einem zu starkem Gefälle liegen und die den Roboter zum Umkippen bringen würden.

Das Generieren von Lösungen erfolgt nun in mehreren Schritten. Zu Beginn müssen einige Berechnungen für die Festlegung des Pfades zum Ziel ablaufen, bevor der Hauptteil des Algorithmus abläuft, welcher sich schrittweise an das Ziel nähert, bis der Abstand zum Ziel annähernd null beträgt.

3.2.1 Festlegung des Pfades zum Ziel

Eine erste Methode ist es, den Weg vom Start zum Ziel auf direktem Weg zu erreichen. Da allerdings nicht immer Lösungen auf direktem Weg möglich sind, müssen auch andere Pfade generiert werden. Beispielsweise könnte ein unüberwindbarer Graben oder eine hohe Mauer dafür sorgen, dass kein direkter Weg möglich ist. In diesem Fall müsste der Laufroboter den Weg um das Hindernis planen, um eine

gültige Lösung zu finden. Um nun alle Fälle abzudecken, benötigt man eine zufällige Streckenplanung. Diese erfolgt, indem zuerst die Anzahl der Pfadsegmente festgelegt wird und dann genau so viele Wegpunkte gesetzt werden.

Festlegung der Anzahl der Pfadsegmente

Die Festlegung der Anzahl der Pfadsegmente erfolgt über eine *geometrische Verteilung*. Diese Verteilung wird genutzt, da geringe Streckenanzahlen häufig sein sollen, da davon ausgangen wird, dass der direkte Weg der schnellste Weg ist. Höhere Streckenanzahlen sollen weniger oft auftreten. Durch die Veränderung des Parameters p lassen sich die Wahrscheinlichkeiten dahingehend variieren, dass entweder höhere oder niedrigere Streckenanzahlen bevorzugt werden. Standardmäßig geht man von $p = 0.5$ aus, was bedeutet, dass die Wahrscheinlichkeit genau einen Streckenabschnitt zu erhalten 50% beträgt.

Festlegung der Wegpunkte

Nun ist definiert, wie viele Segmente der Pfad haben soll. Jetzt müssen die Wegpunkte auf dem Pfad zufällig generiert werden. Für jeden Pfadpunkt P wird einzeln eine Lösung generiert. Zunächst wird der Mittelpunkt $m(x, y)$ sowie die Länge l zwischen Start und Ende berechnet. Auf den Mittelpunkt wird nun ein zufälliger Vektor $r(x, y)$ addiert, dessen generierter Wert vom negativen bis zum positiven Betrag der Länge l reicht. Gleichung 3.1 und Gleichung 3.2 zeigen mögliche Ergebnisse für drei Pfadsegmente P_1, P_2 und P_3 für den Startpunkt $S(0, 0)$ und dem Zielpunkt $Z(10, 0)$. Der letzte Pfadpunkt P_3 entspricht immer genau der Zielposition, damit der Roboter dieses Ziel in jedem Fall erreicht. Abbildung 3.1 zeigt die generierten Punkte und damit den Pfad des Roboters grafisch dargestellt.

$$P_1 = \begin{pmatrix} x_m + x_r \\ y_m + y_r \end{pmatrix} = \begin{pmatrix} 5 - 2.31 \\ 0 - 4.1 \end{pmatrix} = \begin{pmatrix} 3.31 \\ -4.1 \end{pmatrix} \tag{3.1}$$

$$P_2 = \begin{pmatrix} x_m + x_r \\ y_m + y_r \end{pmatrix} = \begin{pmatrix} 5 + 3.1 \\ 0 + 0.5 \end{pmatrix} = \begin{pmatrix} 8.1 \\ 0.5 \end{pmatrix} \tag{3.2}$$

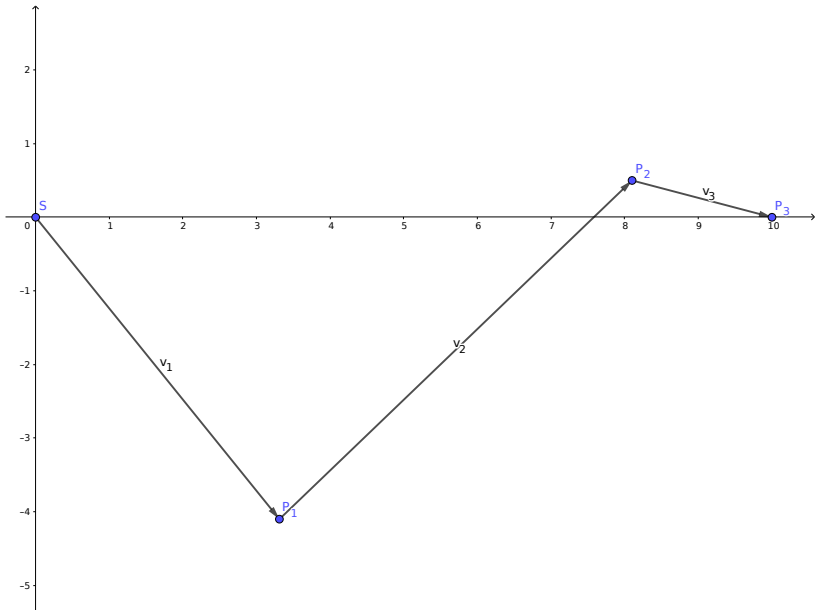


Abbildung 3.1: Zufällig generierte Wegpunkte

Mit dieser Berechnung sind nun für den Mittelpunkt des Körpers des Laufroboters ausschließlich Bewegungen auf diesem Pfad möglich. Bewusst wird hier noch nicht mit der z-Koordinate gearbeitet, da diese je nach Fußstellung und Terrain variiert.

3.2.2 Schrittweise Näherung an das Ziel

Die schrittweise Näherung an das Ziel erfolgt solange, bis der Abstand zum Ziel minimal ist. Ein exakter Vergleich mit dem Abstand null ist nicht effektiv, da der Roboter sonst fortwährend versuchen würde auf seine Zielposition zu gelangen, aber ununterbrochen minimale Abweichungen zum Ziel hätte, was den Prozess erneut anstoßen würde. Der Ablauf, um die Füße anzuheben oder abzusetzen sowie den Körper zu bewegen, läuft folgendermaßen ab:

1. Berechnung des zulässigen Bereichs
2. Auswahl der nächsten Fuß-Konfiguration

Neben den zwei Schritten nennt André Herms [1] noch zwei weitere Schritt zur Berechnung der Mittelpunktpositionen und der Festlegung der Dauer der Bewegungen. Dieser werden erst nach der schrittweisen Näherung an das Ziel durchgeführt.

Berechnung des zulässigen Bereichs

Der zulässige Bereich des Mittelpunkts ist auf Grund von zwei Kriterien eingeschränkt. Zum einen durch der maximalen Reichweite der Fußpunkte vom Mittelpunkt aus. Zum anderen dadurch, dass der minimale *Stability Margin* nicht unterschritten werden darf. Der Stability Margin ist der kleinste Abstand zum Mittelpunkt der konvexen Hülle der Fußpunkte vom Mittelpunkt aus. Dieser ist ein Maß dafür, wie stabil der Roboter steht. Ist der Stability Margin kleiner als null, so ist der Mittelpunkt außerhalb der konvexen Hülle, was ein Kippen des Roboters verursacht. Je größer der Stability Margin, desto stabiler steht der Roboter.

Der zulässige Bereich kann nur an zwei Punkten verlassen werden. Damit wird ein Start- und ein Endpunkt der aktuellen Fußstellung gebildet. Diese sind analytisch schwer zu berechnen. Daher muss mit einem numerischen Verfahren, der *binären Suche*, mit einem inneren zulässigen Punkt und einem äußerem unzulässigen Punkt der Schnittpunkt mit dem zulässigen Bereich gesucht werden. Dazu genügt eine Funktion welche angibt, ob der Mittelpunkt zulässig oder unzulässig ist.

Auswahl der nächsten Fuß-Konfiguration

Der Mittelpunkt des Roboterkörpers kann sich in diesem aktuellen Zustand nur noch vom zuletzt berechneten Start- und Endpunkt des zulässigen Bereichs bewegen. Dies geschieht durch das Verschieben der Körpermitte. Dazu müssen die Füße gehoben und umgesetzt werden. Auch hier müssen die zu Beginn genannten Kriterien beachtet werden. Diese sind unter anderem, dass sich mindestens drei Füße auf dem Boden befinden und dass mindestens ein Fuß auf jeder Seite den Körper stützt, damit der Stability Margin positiv ist. Mit diesen Regeln lassen sich 40 mögliche Fußkonfigurationen, auch Stützzustände genannt, darstellen, welche in Tabelle 3.2 abgebildet sind.

Mit jedem Übergang eines Stützzustands in einen anderen Stützzustands wird exakt ein Fuß entweder angehoben oder abgesetzt. Es können auch mehrere Füße gleichzeitig angehoben oder abgesetzt werden, indem in dem aktuellen Übergang eine Zeitdauer von $t = 0s$ angenommen wird. Damit wird der nächste Stützzustand mit dem aktuellen Stützzustand ausgeführt. Tabelle 3.3 gibt alle möglichen Stützzustände an.

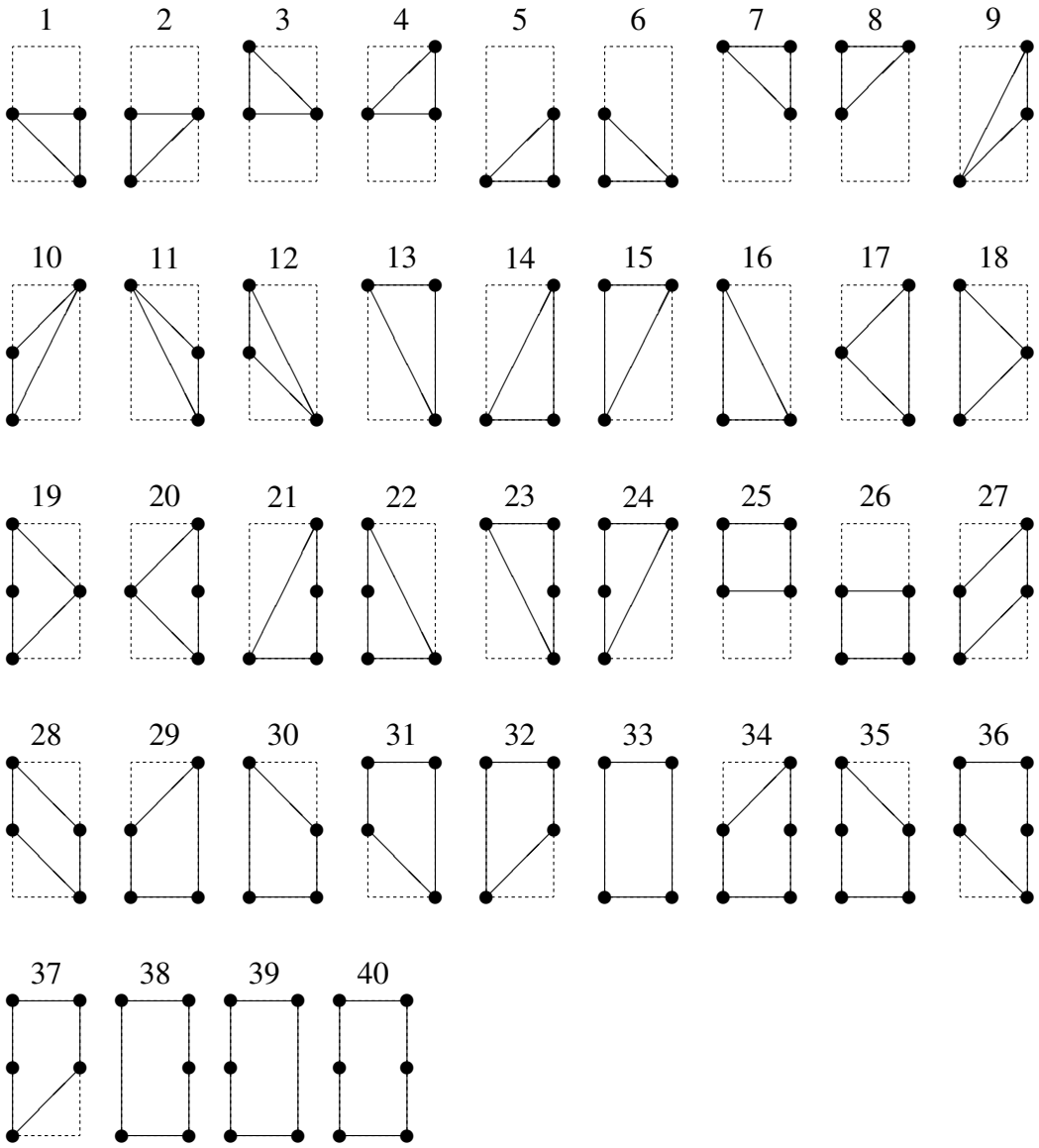


Tabelle 3.2: 40 zulässige Stützzustände [1]

Von dem aktuellen Stützzustand wird per Zufall bestimmt, wie der Nachfolgezustand sein soll. Dazu werden alle möglichen nächsten Zustände bewertet. Wird ein Fuß angehoben und verletzt dabei die Stabilitätskriterien, wird dieser Übergang verworfen. Aus den verbleibenden zulässigen Übergängen wird nun zufällig ein Übergang ausgewählt.

Zustandswechsel können theoretisch sofort wieder aufgehoben werden, was dazu führt, dass der Laufroboter sich nicht nach vorne bewegt. Daher müssen vorher veränderte Füße weniger gewichtet werden und Füße, die länger nicht verändert wur-

Stützzustand	*mögliche Nachfolger*
1	28 20 26
2	19 27 26
3	25 19 28
4	25 27 20
5	30 21 26
6	22 29 26
7	25 32 23
8	25 24 31
9	32 27 21
10	24 27 29
11	23 28 30
12	31 28 22
13	31 23 33
14	33 29 21
15	24 32 33
16	33 22 30
17	31 20 29
18	32 19 30
19	3 18 2 37 35
20	4 17 1 36 34

Stützzustand	*mögliche Nachfolger*
21	9 14 5 38 34
22	12 16 6 39 35
23	7 13 11 36 38
24	8 15 10 37 39
25	8 7 3 4 37 36
26	2 1 6 5 35 34
27	4 10 9 2 37 34
28	3 12 11 1 36 35
29	10 17 14 6 39 34
30	18 11 16 5 38 35
31	8 13 12 17 36 39
32	7 15 18 9 37 38
33	15 13 16 14 39 38
34	27 20 29 21 26 40
35	19 28 22 30 26 40
36	25 31 23 28 20 40
37	25 24 32 19 27 40
38	32 23 33 30 21 40
39	24 31 33 22 29 40
40	37 36 39 38 35 34

Tabelle 3.3: Transitionstabelle der Stützzustände [1]

den, höher gewichtet werden. Dies wird über einen Bonus geregelt. Die Berechnng läuft wie folgt ab:

- Ist ein Fuß verändert worden, wird sein Bonus auf null gesetzt.
- Ist ein Fuß nicht verändert worden, wird sein Bonus um eins erhöht.

Die Auswahl des nächsten Übergangs erfolgt über eine *Glücksradauswahl*. Dabei wird über eine Bewertungsfunktion $f(b_i) = (b_i)^2$ für jeden Fuß die Summe aller Boni gebildet. Die Wahrscheinlichkeit, dass ein Übergang, sprich eine Fußänderung, ausgewählt wird, hängt von der eigenen Bewertung des Fußes verglichen mit der Summe aller Bewertungen ab. Nach der zufälligen Auswahl des nächsten Zustands ergibt sich daraus entweder ein Anheben oder ein Absetzen des gewählten Fußes.

Beim Anheben eines Fußes kann die konvexe Hülle der Fußpositionen vom Mittelpunkt aus kleiner werden. Ist sie zu gering, kommt es zum Kippen des Roboters.

Das Absetzen eines Fußes ist in der Regel immer möglich, da die konvexe Hülle nur größer werden kann. Ausnahmen ergeben sich, wenn sich beispielsweise eine tiefe Klippe oder eine Mauer in der Nähe der Fußposition befindet. Beim Absetzen muss ein zufälliger Punkt in der Nähe des Fußes bestimmt werden. Als Erwartungswert wird hier ein in Richtung Ziel verschobener Mittelpunkt addiert. Sollte die Fußposition nicht gültig sein, läuft eine spiralförmige Suche rund um den Punkt ab. Laut André Herms passiert es nur sehr selten, dass dabei keine gültigen Lösungen gefunden wird.

Hat der Algorithmus nun schon die maximale Anzahl an Durchläufen erreicht, ist die Lösung ungültig. Ansonsten beginnt der Algorithmus nun wieder bei der Berechnung des zulässigen Bereichs für die neue Roboterposition, bis der Mittelpunkt des Roboters das Ziel erreicht hat.

3.2.3 Berechnungen des Mittelpunkts und der Bewegungsdauer

Da nun jeder Übergang, eingeschlossen der Fußposition, der Fußkonfiguration und des zulässigen Bereiches des Mittelpunkts definiert ist, können nun konkrete Werte für den zulässigen Bereich des Mittelpunkts definiert werden, damit auch Bewegungen mit konkreten Werten möglich sind. Da die zulässigen Bereiche der Übergänge sich überlappen, kann ein Mittelwert für die Bereiche des vorherigen und des nächsten Übergangs definiert werden. Dies gilt sowohl für Übergänge, bei denen ein Fuß angehoben als auch abgesetzt wird. Da eine Gleichverteilung ungünstige Ergebnisse erzielt, wird hier eine Dreiecksverteilung genutzt, die den vorherigen Mittelpunkt als Erwartungswert annimmt. Dadurch haben kürzere Bewegungen eine höhere Wahrscheinlichkeit.

Zum Abschluss wird noch die Zeit zwischen den einzelnen Übergängen benötigt. Die minimal mögliche Zeit hängt davon ab, wie lange der Mittelpunkt zur neuen Position benötigt und wie lange ein Fuß zum Absetzen benötigt. Um beide Kriterien einzuhalten ist das Maximum beider Werte die Zeit zwischen dem Übergang.

3.3 Bewertung gültiger Lösungen

Da das Random Sampling nach jedem Durchlauf nur die beste Lösung übernimmt, benötigt der Algorithmus Kriterien für die Bewertung. André Herms erstellt daraus

eine Bewertungsfunktion, welche für jede Lösung ausgeführt werden kann. Dabei nutzt er die folgenden Bewertungskriterien:

- Dauer der Bewegung
- Kippstabilität
- Untergrundstabilität
- Zulässigkeit der Lösung

Es sind noch weitere Kriterien denkbar wie der Energieverbrauch oder die Fehlertoleranz beim Ausfall eines Beins, welche allerdings nicht für die Umsetzung dieses Laufplaners eingesetzt wurden.

Kapitel 4

Portierung des Laufplaners nach ROS und Gazebo

Dieses Kapitel geht auf die Einzelheiten und die Abläufe der Portierung des Laufplaners nach ROS und Gazebo ein. Zunächst werden die wesentlichen Aspekte des bestehenden Laufplaner analysiert und mögliche Herausforderungen dargestellt. Dann wird ein eigenes ROS-Paket entworfen, welches die Basis für den neuen Laufplaner darstellt. Das Kapitel stellt alle nötigen Schritte dar, um die Simulation aufzusetzen und den Roboter dazu zu bringen, Fußbewegungen auszuführen.

4.1 Analyse bestehender Laufplaner

Der von André Herms [1] und von Uli Ruffler [3] weiterentwickelte Laufplaner für den *LAURON* basiert auf der 3D-Bibliothek OpenInventor [2]. Dieses ToolKit basiert wiederum auf OpenGL und ist in C++ implementiert. Des Weiteren ist es objektorientiert und daher orientiert sich auch der Laufplaner an diesem Programmierstil. Auf Grund dieser Tatsache existieren Klassen bzw. Objekte, die nun beschrieben werden:

- *World*: speichert die Objekte der Sicht auf die Landschaft und den Roboter.
- *Terrain*: enthält die Informationen über die Höhenkarte.
- *Eyes*: stellt die Sicht auf die Landschaft dar.
- *Robot*: ist verantwortlich für die Robotersteuerung und definiert außerdem die inneren Objekte *Head* und *Leg*.

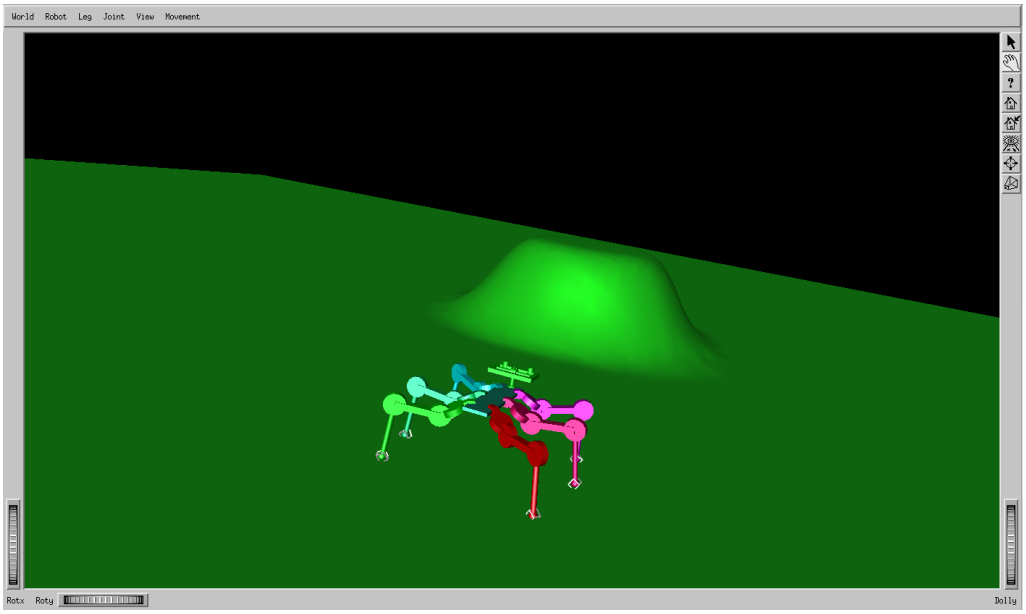


Abbildung 4.1: Screenshot aus der OpenInventor-Umgebung

Für jedes dieser Darstellungsobjekt in OpenInventor wird ein *SoSeperator* benötigt. Da der Name dieses Objekts einzigartig sein muss, sind die Objekte auch nur einmalig nutzbar. Dies erschwert die Aufgabe, später einmal mehrere Roboter gleichzeitig in der Simulation zu testen. Code-technisch sollte das dem Design-Pattern *Singleton* folgen. Der Laufplaner sichert diese allerdings nicht derart ab.

Des Weiteren stellt das Terrain-Objekt die Höhenkarte der Landschaft in OpenInventor zur Verfügung. Dies entspricht nicht dem, was dem Roboter tatsächlich an Kartenmaterial zur Verfügung hat. Diese Tatsache erschwert die Portierung auf ein anderes System, da eine Möglichkeit geschaffen werden muss, dem Roboter beizubringen, was er sieht, um weiterhin mit Höhenangaben arbeiten zu können.

Eine Konfigurationsdatei existiert bereits in dieser Umgebung, die es ermöglichen soll, auch andere Robotermodelle zu testen. Genau dieser Fall ist an dieser Stelle interessant, da die Portierung des Laufplaners auf den Akrobat geschehen soll. Diese wird für den Laufalgorithmus allerdings nicht genutzt, da dort Roboterangaben direkt im Quelltext definiert sind.

Weiterhin wird die aktuelle Fußkonfiguration sowie die Fußposition getrennt voneinander gespeichert. Die Fußkonfiguration ist in eine Bitlogik gespei-

chert. Dies ist sehr nützlich, um Speicherplatz zu sparen, macht auf der anderen Seite das Finden von Fehlern schwieriger.

4.2 Vorgehensweise und Aufbau des Pakets

Der zuvor analysierte Laufplaner muss nun in das ROS portiert werden. Ferner muss der Algorithmus zur Laufplanung extrahiert und mit ROS kompatibel gemacht werden. Dabei gibt es einige Herausforderungen, die gelöst werden müssen:

- Code-technisch ist der Laufplaner *fest verdrahtet mit dem Terrain der OpenInventor-Oberfläche*, während das neue Paket auf Gazebo laufen soll.
- Der Laufplaner nutzt für geometrische Berechnungen *Klassen aus der OpenInventor-Bibliothek*, während ROS auf die Klassen des TF-Frameworks zurückzugreifen soll.
- Durch *unterschiedliche Robotermodelle* können Maße innerhalb des Codes anders definiert sein und müssen angepasst werden.

Da eine komplette Kopie des vorherigen Projekts auf Grund verschiedener Umgebungen nicht möglich ist, ist die Vorgehensweise schrittweise wichtige Codestellen zu übertragen und zu testen. Dies hat den Vorteil, dass Funktionen unabhängig voneinander getestet werden können. Damit ergibt sich der folgende Gesamtablauf für die Portierung:

1. Aufsetzen des ROS-Pakets
2. Aufsetzen der Simulation
   a) Aufsetzen des Roboter-Modells
   b) Aufsetzen der Gelenkmotoren
3. Aufsetzen der Fußsteuerung
4. Testen der Fußsteuerung
5. Portierung des Laufalgorithmus

Während der Arbeit hat es sich als sinnvoll herausgestellt, die Portierung erst zum Schluss zu beginnen. Der Grund dafür ist, dass es damit während der Por-

tierung möglich ist schon Artefakte in der Simulation zu testen. Damit lässt sich wesentlich besser einschätzen, ob das Übertragene auch funktioniert. Außerdem basiert der Laufplaner auf Funktionalitäten wie dem Roboter-Modell und der Definition der Gelenkmotoren. Mit dieser Vorgehensweise kann das Paket nun schrittweise implementiert werden.

Abbildung 4.2 definiert die Ordnerstruktur des ROS-Pakets. Die Ordnerstruktur ist typisch für ROS-Pakete und findet sich ähnlich in vielen weiteren Paketen der ROS-Community. Der Aufbau des Pakets ist an das Akrobat-Paket [4] sowie das Hopper-Paket [5] angelehnt. Beide Pakete nutzen wichtige Konzepte für das Robotermodelle und die Gelenksteuerung, die auch für dieses Projekt wichtig sind.

4.3 Aufsetzen der Simulation

Das erste Ziel ist es nun, den Akrobat in Gazebo anzuzeigen. Dazu muss das Robotermodell integriert werden, die Gelenkmotoren definiert werden und die Gazebo-Welt aufgesetzt werden.

4.3.1 Aufsetzen des Robotermodells mittels urdf

Als erstes wird das Robotermodell in das neue ROS-Paket integriert. Die vorliegende Roboterbeschreibung liegt im *urdf*-Format vor. Desweiteren existieren die 3D-Modelle im *stl*-Format, welche im Robotermodell eingebunden sind. Damit das Robotermodell auch bei späteren Änderungen noch wartbar bleibt, wird die zusammenhängende Datei aufgeteilt, so dass nun eine Datei für die Robotermitte und eine Datei für jedes einzelne Bein existiert. Dabei ist das Wrapper-Format von urdf mit dem Namen *xacro* sehr von Vorteil, da es mehr Flexibilität in urdf-Dateien bringt. Dadurch lassen sich unter anderem andere xacro-Dateien per *include* einbinden, was sich in diesem Fall als nützlich herausgestellt hat. Mit diesem Aufbau könnte das Robotermodell nun schon über ein Launch-File im 3D Visualisierungs-Tool „rviz“ angezeigt werden. Abbildung 4.3 zeigt stellt dies sowie die Koordinatensystemen des Roboters dar.

```
hexapod
├── config
│   ├── config.rviz
│   └── hexapod.yaml
├── urdf
│   ├── hexapod.xacro
│   ├── leg-1.xacro
│   ├── leg-2.xacro
│   ├── leg-3.xacro
│   ├── leg-4.xacro
│   ├── leg-5.xacro
│   └── leg-6.xacro
├── launch
│   ├── rviz.launch
│   ├── gazebo.launch
│   ├── model.launch
│   ├── akrobat_walk.launch
│   └── control.launch
├── include
│   ├── akrobat
│   │   ├── akrobat_init.h
│   │   ├── JointStateToGazebo.h
│   │   ├── ControlRandomSampling.h
│   │   ├── JointStateToDynamixel.h
│   │   ├── FootConfiguration.h
│   │   └── Akrobat.h
│   └── pugixml
├── worlds
│   └── default.world
├── stl
│   ├── hexapod_link.stl
│   ├── coxa_r_link.stl
│   ├── coxa_l_link.stl
│   ├── femur_link.stl
│   └── tibia_link.stl
└── src
    ├── akrobat
    │   ├── akrobat_main.cpp
    │   ├── JointStateToGazebo.cpp
    │   ├── FootConfiguration.cpp
    │   ├── JointStateToDynamixel.cpp
    │   ├── Akrobat.cpp
    │   └── ControlRandomSampling.cpp
    └── pugixml
```

Abbildung 4.2: Dateibaum des ROS-Pakets

Für das spätere Hinzufügen des Akrobats in die *Gazebo*-Simulation müssen neben der Visualisierung, die für den rviz ausreichend war, noch weitere Anpassungen am Roboter-Modell vorgenommen werden:

- Aufsetzen des Kollisionsmodells durch das Attribut <collission>
- Aufsetzen der Massenträgheit durch das Attribut <inertia>

Aufsetzen des Kollisionsmodells

Das Kollisionsmodell muss für jedes Körperteil einzeln definiert werden. Es wird durch ein Ursprung und einem geometrischen Objekt definiert. Das ROS stellt hierfür eine Schnittstelle zur Verfügung, die zwei Möglichkeiten anbietet. Zum einen können einfache geometrische Objekte wie ein Zylinder oder ein Quader als Kollisionsmodell definiert werden. Dies ist in unserem Fall nicht möglich, da sich die Beinsegmente nicht exakt durch einzelne geometrische Objekte modellieren lassen. Daher ist es sinnvoll auf die zweite Variante zurückzugreifen und die exakten 3D-Modelle über die stl-Dateien als Kollisionsmodelle anzugeben. Da diese viel detaillierter sind, wird daher zwar mehr Rechenleistung benötigt, aber auch die physikalischen Bewegungen sehen realer aus. Eine Möglichkeit wäre es die 3D-Modelle mittels MeshLab

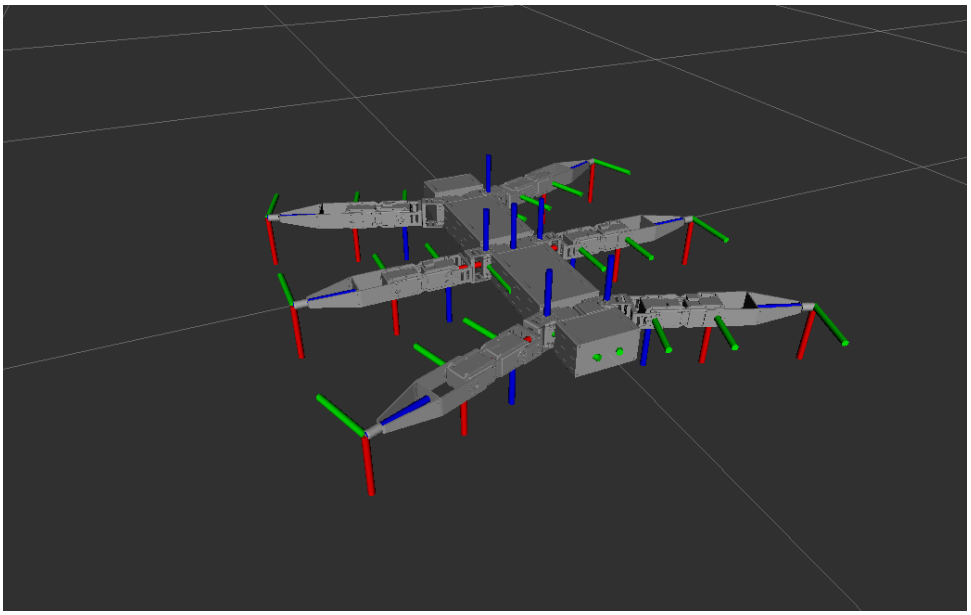


Abbildung 4.3: Darstellung des Akrobats im rviz

[6] runterzurechnen, so dass sie später besser von Gazebo verarbeitet werden können, da sie optimiert sind.

Aufsetzen der Massenträgheit

Ebenfalls muss die Massenträgheit für jedes Körperteil einzeln definiert werden. Dabei wird eine Masse sowie das Trägheitsmoment angegeben. Die Masse lässt sich durch Wiegen der einzelnen Körperteile herausfinden. Um das Trägheitsmoment für das jeweilige Körperteil herauszufinden, bietet Gazebo die Möglichkeit die Werte über MeshLab auszulesen und umzuwandeln [7]. Abbildung 4.4 zeigt die Extraktion der Trägheitsmomente aus MeshLab. Dies erfolgt über mehrere Schritte, um möglichst genaue Ergebnisse zu erhalten:

- Skalierung in MeshLab mit Faktor 10 bis 100
- Automatische Berechnung der Trägheitsmomente in MeshLab
- Teilen des Ergebnisses durch den zuvor skalierten Faktor
- Multiplizieren des Ergebnisses mit der berechneten Masse
- Teilen des Ergebnisses durch das berechnete Volumen
- Eintragen der berechneten Werte in ROS

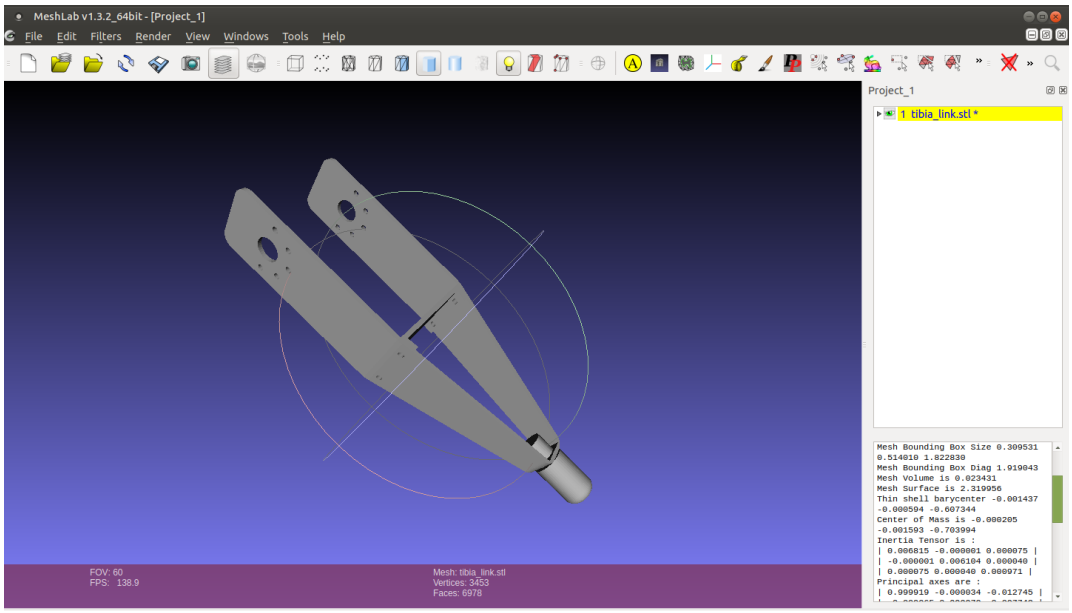


Abbildung 4.4: Auslesen der Trägheitsmomente mit MeshLab

4.3.2 Definition der Gelenkmotoren mittels ros_control

Das zuvor aufgesetzte Robotermodell könnte nun in Gazebo angezeigt werden. Allerdings ist dieses noch nicht durch Eingaben von außen beweglich, da keine Gelenkmotoren definiert sind. Diese werden in diesem Abschnitt behandelt.

Hierbei hat sich das Paket *ros_control* als geeignet herausgestellt. Für das Aufsetzen des Pakets wird eine Konfigurationsdatei für die Definition aller Controller benötigt. Außerdem muss im Robotermodell ein Gelenk auf einen Controller abgebildet werden, damit das Gelenk angesteuert werden kann. Außerdem müssen zwei wesentliche Plugins eingebunden und konfiguriert werden:

- gazebo_ros_control
- p3d_base_controller

Abschließend muss die Konfigurationsdatei geladen und zwei ROS-Nodes gestartet werden. Dies ist zum einen der *robot_state_publisher*, der die publizierten Bewegungen an den Roboter weitergibt, sowie ein *controller_manager*, der letztendlich die einzelnen Gelenke in der Simulation bewegt. Hier lässt sich statt dem *controller_manager* auch direkt ein *dynamixel_manager* anbinden, so dass später leicht zwischen Simulation und dem realen Roboter gewechselt werden kann. Durch die Einbindung existieren nun zur Laufzeit einige wichtige ROS-Topics, wie in Abbildung 4.5 zu sehen ist.

4.3.3 Aufsetzen der Umgebung mittels Gazebo

Nun fehlt nur noch die Gazebo-Umgebung. Diese lässt sich mit einem von Gazebo bereitgestellten Launch-File starten. Es besteht die Möglichkeit Parameter mitzugeben. Die folgende Auflistung beschreibt einige wichtige Parameter:

- *world_name*: Dateipfad zur gewünschten Welt
- *debug*: Gibt Informationen zur Analyse während der Laufzeit aus
- *gui*: Startet die grafische Oberfläche
- *paused*: Pausiert die Simulation

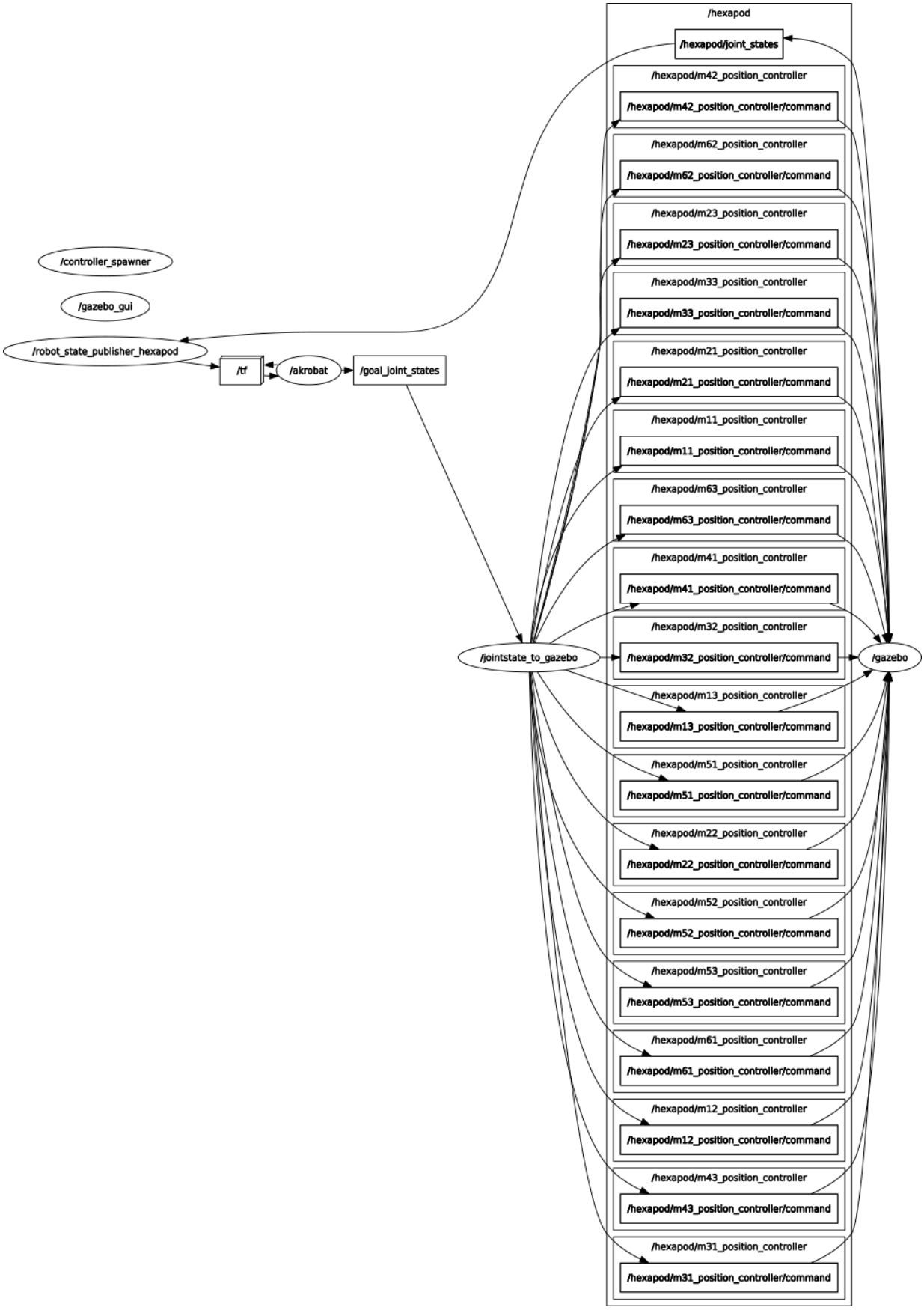


Abbildung 4.5: ROS-Topics der Simulation

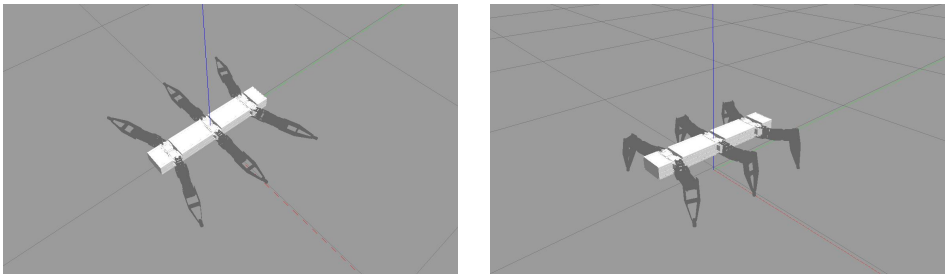
Damit ist das Aufsetzen des Roboters sowie der Simulationsumgebung vollständig. Der Roboter wird nun in Gazebo angezeigt, wie in Abbildung 4.6a zu sehen ist.

4.4 Aufsetzen der Ausgangsposition

Der Roboter wird nun zwar im Gazebo angezeigt, allerdings liegt dieser nun flach auf dem Boden. Das Ziel ist es daher, die Fußsteuerung aufzusetzen, so dass der Roboter sich zu Beginn in seine Ausgangsstellung begibt. Von dort aus wartet der Roboter auf weitere Befehle für die Bewegung.

Da ROS-Nodes allgemein in einer Schleife laufen, wird der folgende Ablauf kontinuierlich ausgeführt. Der Ablauf sorgt wie in Abbildung 4.6b zu sehen, dafür, dass der Roboter in den nächsten Durchläufen in der Ausgangsposition steht. Dies erfolgt in mehreren Schritten:

1. Zunächst wird über das *TF-Framework* mit Hilfe von Transformationen und Rotationen die Ausgangsposition vom obersten Gelenk zum Endeffektor berechnet. Dies wird im weiteren Verlauf als Ausgangsposition definiert und eingelesene Bewegungen sind relativ zu dieser Position.
2. Die dazugehörigen Winkel, die benötigt werden, um in die Ausgangsposition zu kommen, werden nun mittels *inverser Kinematik* berechnet.
3. Die drei Winkel werden nun in das Topic *goal_joint_states* geschrieben, was ein tatsächliches Verändern der Winkel in der Simulation oder am echten Roboter verursacht. Dabei wird vorher geprüft, ob die Winkel gültig sind, d.h. dass sie sich über dem minimal und unter dem maximal



(a) Vor Aufsetzen der Ausgangsposition **(b)** Nach Aufsetzen der Ausgangsposition

Abbildung 4.6: Aufstehen des Akrobat in Gazebo

erlaubten Winkel befinden. Ist das nicht der Fall, wird eine Warnung ausgegeben.

4.5 Generieren und Einlesen von Bewegungen als xml-Datei

Da der Laufplaner und die Simulation voneinander getrennt sein sollen, damit das System flexibel für den Austausch von Komponenten bleibt, benötigt der Laufplaner ebenso wie bei André Herms [1] und Uli Ruffler [3] eine xml-Schnittstelle, welche Bewegungen repräsentiert.

Die hier entwickelte Schnittstelle ist eine Abwandlung der vorherigen Schnittstellen, da diese nicht die Dauer von Bewegungen speichert, sondern lediglich für jeden Schritt die Fußpositionen relativ zur definierten Ausgangsposition.

Die Generierung erfolgt durch die Iteration über die von einem Algorithmus generierten Schritte für die Fußpositionen. Statt für die Generierung eine xml-Bibliothek zu nutzen, reicht hier eine einfache Ausgabe in einer Datei mittels *ofstream*.

Listing 4.1 zeigt eine abgespeckte XML-Bewegung der ersten beiden Füße, welche den Fuß an der 1. Stelle ein wenig in y-Richtung verschieben würde, was einer Verschiebung der Robotermitte verursacht, sofern die anderen Füße, welche auf dem Boden sind, das auch tun. Des Weiteren wird der Fuß an der 2. Stelle ein Stück angehoben.

Für das Einlesen der XML-Datei wird Pugixml [8] verwendet, welches unkompliziert und schnell xml-Dateien generieren oder einlesen kann. Die eingelesene Bewegung wird als Vektor in C++ gespeichert und an den Akrobat weitergegeben. Dieser ist dann für das Abspielen der Bewegung zuständig.

4.6 Abspielen der Bewegungen

Das ROS läuft in einer Schleife, bei der die Periode definiert werden kann. Als geeignet hat sich ein Wert von 20 Millisekunden herausgestellt. Auf Grund der Tatsache, dass die xml-Datei keine Bewegungsdauern speichert, muss ei-

```
<?xml version="1.0"?>
<movement>
  <foot number="0">
    <step>
      <x>0</x>
      <y>0.03992</y>
      <z>0</z>
    </step>
  </foot>
  <foot number="1">
    <step>
      <x>0</x>
      <y>-0.04</y>
      <z>0.04</z>
    </step>
    </step>
  </foot>
</movement>
```

Listing 4.1: Aufbau der Bewegungsdatei

ne Synchronisation zwischen dem ROS-Loop und dem Abspielen der Bewegung aufgesetzt werden. Damit die Bewegungen nicht unrealistisch aussehen und linear von Position zu Position wechseln, wird eine Interpolation zwischen den einzelnen Punkten mittels einer trigonometrischen Funktion aufgesetzt.

$$f(x) = -0,5 * cos(x + 0,5) + 0,5 \tag{4.1}$$

$$f(y) = -0,5 * cos(y + 0,5) + 0,5 \tag{4.2}$$

$$f(z) = -0,5 * cos(z + 0,5) + 0,5 \tag{4.3}$$

Der in Gleichung 4.1, Gleichung 4.2 sowie Gleichung 4.3 verwendete Cosinus ist so modelliert, dass er von 0 bis 1 reicht. Geht man davon aus, dass eine Bewegung in acht Durchläufen der Schleife abgelaufen sein soll, ergibt sich also für eine Bewegung eine Zeitdauer von mindestens 160 Millisekunden. Diese Zahl kann weiter optimiert werden, so dass die maximale Geschwindigkeit der Beinregler nicht überschritten wird. In der Simulation lässt sich mit diesem Wert allerdings sehr gut testen.

Code-technisch ist das mit drei Variablen umgesetzt, die die aktuelle Zwischenposition bestimmen. Die erste Variable gibt die maximale Zahl der Zwi-

schenpositionen an (*maxTicks*). Die zweite Variable gibt die aktuelle Stelle (*tick*) an, so dass der Algorithmus über die Funktion ausrechnen kann, was die nächste Position sein wird. Die Variable *diff* ist der Vektor von der Start- zur Zielposition. Listing 4.2 zeigt die vollständige Berechnung einer Zwischenposition.

```
1 double x = diff.x() * (-0.5 * cos(M_PI * tick / maxTicks) + 0.5);
2 double y = diff.y() * (-0.5 * cos(M_PI * tick / maxTicks) + 0.5);
3 double z = diff.z() * (-0.5 * cos(M_PI * tick / maxTicks) + 0.5);
```

Listing 4.2: Interpolation der Zwischenpositionen

In jedem Schritt wird der *tick* um eins nach oben gesetzt, bis er bei der Grenze von *maxTicks* angekommen ist. Dann wird dieser zurückgesetzt und die nächste Bewegung wird ausgeführt.

4.7 Aufsetzen des Dreifußgangs

Bevor das Random Sampling aufgesetzt wird, ist es sinnvoll die gesamte Simulation zunächst mit einem statischen Laufmuster zu prüfen. Das macht es später einfacher zwischen einem Fehler in der Simulation und einem Fehler in der Laufplanung zu unterscheiden.

Beim Dreifußgang kann ein Fuß einen von zwei Stati annehmen:

1. Der Fuß wird angehoben und umgesetzt.
2. Der Fuß ist für das Verschieben der Körpermitte verantwortlich.

Zunächst werden die Stati für jeden Fuß in einem Array in C++ gesetzt. Die Füße 1, 4 und 5 starten in der ersten Phase, die Füße 2, 3 und 6 starten in der zweiten Phase. Mit jeder Bewegung wird dies abgewechselt sowie die Positionen relativ zum Fußpunkt gesetzt. Für Phase 1 muss die z-Koordinate positiv gesetzt werden, damit der Fuß angehoben wird. In Phase 2 muss die z-Koordinate auf null gesetzt werden, damit der Fuß die Körpermitte verschieben und stützen kann sowie die Position ein wenig nach hinten verschoben werden, damit der Körper sich nach vorne bewegt. Die stützenden Füße sind so gewählt, dass der Stability Margin größer als null ist, damit der Roboter nicht umkippt. Das Ergebnis wird als Bewegungsdatei exportiert und kann vom Roboter eingelesen werden.

4.8 Aufsetzen des Random Samplings

Wie auch bei den Vorgängern ist die Implementierung mit C++ durchgeführt. Im folgenden Abschnitt werden einige wesentliche Unterschiede bei der Implementierung dargestellt.

4.8.1 Abstrahierung der Fußkonfiguration

Während bei dem vorherigen Laufplaner die aktuelle Fußkonfiguration und die Fußposition getrennt voneinander gespeichert wurden, speichert der neue Laufplaner diese in dem Klassenobjekt *FootConfiguration*. Dies hat den Vorteil, dass diese gesamte Fußlogik in einer Klasse gekapselt und unabhängig vom Laufplaner getestet werden könnte.

Außerdem macht dies das Debugging einfacher, da sich in einer gekapselten Klasse beispielsweise Methoden zur Ausgabe definieren lassen. Dadurch lassen sich auch einige redundante Code-Zeilen sparen, da wiederkehrende Funktionen abstrahiert werden.

4.8.2 Anpassung aller Maße

Da die OpenInventor-Simulation alle Angaben in Millimeter gespeichert hat und ROS bzw. Gazebo diese als Meterangaben hinterlegt, muss der Algorithmus ebenfalls Meterangaben liefern.

Des Weiteren müssen fest definierte Längenangaben wie die angepeilte Fußposition beim Absetzen eines Fußes verändert werden.

4.8.3 Nutzung des TF-Frameworks

Alle 2D und 3D-Berechnungen werden nun mit dem TF-Framework aus ROS berechnet. Dieses nutzt die Klasse *Vector3*, um Berechnungen durchzuführen. Diese Klasse bietet unter anderem folgende wichtige Methoden für Vektoren:

- Grundlegende Rechenarten (+, -, *, /)
- Länge der Vektors mittels *length*

- Normalisierung von Vektoren mittels *normalise*
- Distanz zweier Punkte mittels *distance*

4.8.4 Veränderung der Randomisierung

Für die Generierung von Zufallszahlen steigt das System auf den Pseudozufallszahlengenerator Mersenne-Twister [9] um. Außerdem wird beispielsweise die geometrische Verteilung nicht mehr selbst programmiert, sondern mit einer existierenden Funktion über den Pseudozufallszahlengenerator geregelt, die dies bereits implementiert. Der Aufruf erfolgt über die Standardbibliothek wie in Listing 4.3.

```
1 std::random_device rd;
2 std::mt19937 generator(rd());
3 std::geometric_distribution<int> geometricDistribution(0.5);
4
5 int result = randomDistribution(generator);
```

Listing 4.3: Geometrische Verteilung mittels C++

Kapitel 5

Testen der Ergebnisse

5.1 Testen der Fußsteuerung

- mit Hilfe von Tripod Gait - Setup Tripod Gait - 4 Screenshots wo man ein paar Bewegungen sieht

Kapitel 6

Zusammenfassung

todo

Kapitel 7

Ausblick

todo

Abkürzungsverzeichnis

ROS Robot Operating System

Tabellenverzeichnis

3.1 Bewertung des Random Samplings 9
3.2 40 zulässige Stützzustände [1] 13
3.3 Transitionstabelle der Stützzustände [1] 14

Abbildungsverzeichnis

3.1 Zufällig generierte Wegpunkte 11

4.1 Screenshot aus der OpenInventor-Umgebung 18
4.2 Dateibaum des ROS-Pakets 21
4.3 Darstellung des Akrobats im rviz 22
4.4 Auslesen der Trägheitsmomente mit MeshLab 23
4.5 ROS-Topics der Simulation 25
4.6 Aufstehen des Akrobat in Gazebo 26

Listings

4.1 Aufbau der Bewegungsdatei 28
4.2 Interpolation der Zwischenpositionen 29
4.3 Geometrische Verteilung mittels C++ 31

Literatur

[1] A. Herms, „Entwicklung eines verteilten Laufplaners basierend auf heuristischen Optimierungsverfahren“, Magisterarb., Otto-von-Guericke-Universität Magdeburg, Jan. 2004.

[2] J. Wernecke u. a., *The Inventor mentor: programming object-oriented 3D graphics with Open Inventor, release 2*. Citeseer, 1994, Bd. 1.

[3] U. Ruffler, „Laufplanung basierend auf realitätsnahen Umgebungsdaten für einen sechsbeinigen Laufroboter“, Magisterarb., Institut für Robotik der Fakultät für Informatik an der Hochschule Mannheim, Aug. 2006.

[4] *Akrobat - Control and visualization of a six-legged walking robot based on ROS.* Adresse: https://github.com/informatik-mannheim/akrobat.

[5] *Hopper Project for Gazebo using ROS Developement Studio*. Adresse: https://bitbucket.org/theconstructcore/hopper.

[6] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli und G. Ranzuglia, „MeshLab: an Open-Source Mesh Processing Tool“, in *Eurographics Italian Chapter Conference*, V. Scarano, R. D. Chiara und U. Erra, Hrsg., The Eurographics Association, 2008. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[7] *Inertial parameters of triangle meshes*. Adresse: http://gazebosim.org/tutorials?tut=inertia&cat=build_robot.

[8] *Pugixml - Light-weight, simple and fast XML parser for C++ with XPath support*. Adresse: https://pugixml.org.

[9] M. Matsumoto und T. Nishimura, „Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator“, *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, Jg. 8, Nr. 1, S. 3–30, 1998.