hochschule mannheim

Portierung eines Laufplaners für sechsbeinige Laufroboter in das Robot Operating System

Daniel Koch

Bachelor-Thesis

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Informatik

Fakultät für Informatik

Hochschule Mannheim

22.07.2020

Betreuer

Prof. Dr. Thomas Ihme, Hochschule Mannheim

Dipl.-Phys. Ute Ihme, Hochschule Mannheim

Koch, Daniel:
Portierung eines Laufplaners für sechsbeinige Laufroboter in das Robot Operating System / Daniel Koch. –
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2020. 57 Seiten.

Koch, Daniel:
Transferring of a gait planner for a six-legged walking robot into the robot operating system / Daniel Koch. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2020. 57 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

Portierung eines Laufplaners für sechsbeinige Laufroboter in das Robot Operating System

Diese Bachelorarbeit stellt eine Lösung für die Portierung eines bestehenden Laufplaners für sechsbeinige Laufroboter in das Robot Operating System (ROS) dar, einem open source Meta-Betriebssystem für die Entwicklung von Robotern. Ferner ist der bestehende Laufplaner für den Laufroboter LAURON III erschaffen worden, während die Neuentwicklung für den Laufroboter Akrobat ausgeführt werden soll. Diese Arbeit stellt das Konzept und die Implementierung dafür bereit.

Transferring of a gait planner for a six-legged walking robot into the robot operating system

This bachelor thesis provides a solution for the migration of a gait planner of an existing six-legged walking-robot into the Robot Operating System (ROS). The ROS is an open source meta operating system used for the development of robots. The current walk planner is made for the LAURON III while the new planner should be made for the walking robot Akrobat. This bachelor thesis shows the concept and the implementation of this task.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel führt in das Thema der Portierung des Laufplaner in das Robot Operating System (ROS) und Gazebo ein. Neben der Motivation der Arbeit wird hier auch das Ziel der Arbeit definiert. Außerdem stellt dieses Kapitel den Aufbau der Arbeit dar.

1.1 Motivation

Die einfachste Art einen Laufroboter zu steuern, ist die Anwendung eines *statischen oder regulären Laufmusters* wie dem *wave gait*. Diese Laufmuster funktionieren auf ebenem Untergrund für beispielsweise sechsbeinige Laufroboter sehr gut und stabil. Anders sieht es allerdings aus, wenn das Gelände Hindernisse aufweist. Es ist in erster Linie sinnvoll zusätzlich zu dem regulären 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 bzw. *freie Laufmuster* 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 hilfreich, da 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 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 III* entwickelt. 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 neben einer Visualisierung auch eine Physik-Engine bereitstellt. Im weiteren Verlauf des Projekts 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 direkte und inverse Kinematik sowie die Grundlagen zur Laufplanung ein. Außerdem werden die benötigten Frameworks wie das ROS und Gazebo vorgestellt.

Kapitel 3 geht auf verwandete Arbeiten ein. Diese werden in Kapitel 4 genutzt, um eine Lösung für die Laufplanung mit dem Akrobat herzuleiten. Kapitel 5 stellt die Details der Implementierung dar.

Die Ergebnisse der Implementierung der Simulationsumgebung und des Laufplaners aus Kapitel 5 werden in Kapitel 6 dargestellt und bewertet.

Kapitel 7 fasst alle Ergebnisse zusammen und Kapitel 8 gibt einen Ausblick darüber, wie der Laufplaner in zukünftigen Projekten noch weiterentwickelt werden könnte.

Kapitel 2

Grundlagen

Dieses Kapitel stellt die benötigten Grundlagen für die Portierung des Laufplaners in ROS und Gazebo dar. Zunächst geht das Kapitel auf die relevanten Laufroboter, dann auf Kinematik und verschiedene Laufmuster ein. Zuletzt werden die benötigten Frameworks beschrieben.

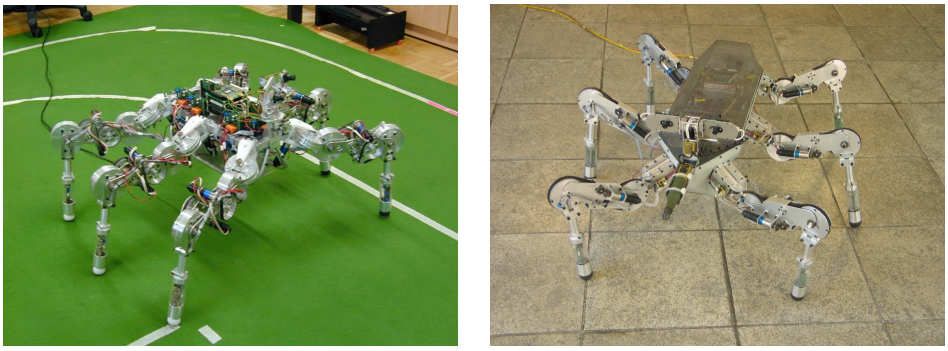
2.1 Laufroboter

Dieser Abschnitt stellt die für diese Arbeit relevanten Laufroboter vor. Dies ist zum einen der *Lauron III*, für den der vorliegende Laufplaner entwickelt wurde; zum anderen ist dies der *Akrobat*, für den der existierende Laufplaner migriert werden soll. Beide sind sich ziemlich ähnlich, jedoch sind die Unterschiede wichtig, um den Laufplaner auf den Akrobat zu migrieren.

2.1.1 Lauron

Dieser Unterabschnitt stellt den Lauron vor. Dieser wurde in verschiedener Literatur schon ausführlich dargestellt, welcher für diesen Abschnitt genutzt wird. [4]–[6], [24], [7]

Die erste Version des Lauron (Laufender Roboter Neuronal gesteuert) wurde 1994 am Forschungszentrum für Informatik (FZI) in Karlsruhe [25] entwickelt. Dabei wurden Teile des Roboters mit traditionellen Techniken, andere Teile wie die Steuerung der Gelenke, die Navigation oder die Interpretation von Sensorinformation



(a) Lauron III **(b)** Lauron IVb

Abbildung 2.1: Verschiedene Versionen des Lauron

mit neuronalen Netzwerken entwickelt. Die zweite Version des Lauron verbessert die Sensorausstattung sowie die Mechanik des Laufroboters. Des Weiteren wurde das Steuerungsprinzip, das auf neuronalen Netzen basierte, durch die hierarichische Modular controller architecture (MCA)-Architektur ausgetauscht. Für die dritte Version *Lauron III*, siehe Abbildung 2.1a, existiert der vorliegende und zu übertragende Laufplaner in Form der 3D-Bibliothek OpenInventor [2]. Daher wird nun näher auf diese Version des Laufroboters sowie auf das grundlegende Konzept des Lauron als Nachahmung der *Carausius morosus* (indische Stabheuschrecke) nach den Untersuchen von Holk Cruse eingegangen. [8] Eine weitere Version des Laufroboters ist der *Lauron IVb*, siehe Abbildung 2.1b. Beide Laufroboter befinden sich aktuell an der Hochschule Mannheim. Des Weiteren gibt es noch den *Lauron IVc*, der im Jahr 2005 fertiggestellt wurde. Die neuste Version des Laufroboters ist der *Lauron V*.

Der Lauron basiert auf dem Vorbild der indischen Stabheuschrecke (Carausius morosus), die sehr gut erforscht ist. Dies gilt sowohl für den mechanischen Aufbau als auch für die Abläufe der Bewegungen des Roboters. Der Körper ist in drei Teile geteilt.

- Kopf (Caput)
- Brust (Thorax)
- Hinterleib (Abdomen)

Die Brust ist wiederum in weitere drei Teile für jeweils ein Beinpaar unterteilt. Damit ergeben sich sechs Beine. Ein Bein besteht aus drei Segmenten:

- Hüfte (Coxa)

- Oberschenkel (Femur)
- Unterschenkel (Tibia)

Die Segmente sind durch die Gelenke Subcoxal α, Coxa-Trochanter β und Femur-Tibia γ verbunden. Der Fuß jedes Beines wird Tarsus genannt. Das erste Gelenk Subcoxal besitzt zwei Freiheitsgrade, die weiteren Gelenke besitzen einen Freiheitsgrad. Damit ist für die indische Stabheuschrecke die minimale erforderliche Zahl an Freiheitsgraden erreicht, damit der Fuß beliebig im dreidimensionalen Raum gesetzt werden kann.

Unter anderem ist noch wichtig, dass der Kopf der indischen Stabheuschrecke zwei lange Fühler besitzt. Diese könnten in einem Roboter beispielsweise als Laserscanner oder Kamera modelliert werden, welche sich einen Überblick über die Gegend verschaffen können.

Der Körper des *Lauron III* trägt die 7 Mikrocontroller (Infineon C167) und einen Hauptrechner, der auf Pentium basiert, die Akkumulatoren sowie den Kamerakopf. An beiden Seiten des Körpers sind jeweils drei Beine angebracht. Der Roboter wiegt 16 Kilogramm und hat eine maximale Zuladung von etwa 15 kg. Die Maximalgeschwindigkeit beträgt 0,5 $\mathrm{m\,s^{-1}}$.

2.1.2 Akrobat

Der folgende Unterabschnitt stellt den Laufroboter *Akrobat* vor und nutzt dazu die Arbeit von Wilen Askerow. [9]

Auch der *Akrobat* (Abbildung 2.2) ist der Form einer Stabheuschrecke nachempfunden und ähnelt in vielerlei Hinsicht dem Lauron. Der Roboterkörper ist ungefähr 56 Millimeter hoch, 102 Millimeter breit und hat eine Seitenlänge von 62 Millimeter. Ebenfalls besitzt dieser sechs Beine mit je drei Segmenten. Diese haben die Längen 72 Millimeter, 97 Millimeter und 163 Millimeter, welche für kinematische Berechnungen von großer Bedeutung sind. Jedes Bein wiegt ungefähr 0,8 Kilogramm.

Jedes der Gelenke hat exakt einen Freiheitsgrad, sodass das gesamte Bein drei Freiheitsgrade hat. Wie auch beim Lauron ermöglicht dies die beliebige Positionierung im dreidimensionalen Raum. Die Gelenke sind mit dem Servomotor Dynamixel RX64 ausgestattet. Außerdem hat jedes Gelenk einen definierten Arbeitsbereich, welcher nicht unter- oder überschritten werden darf.

- $\alpha = -50\,°$ bis $50\,°$
- $\beta = -106\,°$ bis $106\,°$
- $\gamma = -135\,°$ bis $135\,°$

Außerdem besitzt der Roboter am Kopf eine fest verbaute Kamera. Der Roboterkörper verfügt über genügend Freiraum für Batterien im mittleren Gehäuse. Im hinteren Gehäuse ist der Steuerrechner sowie die restliche Elektronik platziert.

Es handelt sich beim Rechner um den Einplatinencomputer Raspberry Pi (Modell B+) mit 512 MB Arbeitsspeicher und einem 700 MHz ARM 11 Prozessor. Der Raspberry Pi bietet vielfältige Anschlussmöglichkeiten:

- Wireless Local Area Network (WLAN)
- 4 Universal Serial Bus (USB)-Anschlüsse
- 1 Ethernet-Anschluss
- 1 High Definition Multimedia Interface (HDMI)-Anschluss, über den die Visualisierung mittels eines externen Bildschirms stattfindet
- Camera Serial Interface (CSI)-Schnittstelle für die Kamera
- Low Voltage Differential Signaling (LVDS)-Anschluss für die Ansteuerung von LCD-Displays

Abbildung 2.2: Der Akrobat vor dem C-Gebäude der Hochschule Mannheim

- GPIO (General Purpose Input Output)-Anschlüsse

Es besteht die Möglichkeit, das Gamepad F710 von Logitech per WLAN anzuschließen. Theoretisch sind auch andere Gamepads möglich, sofern diese im Quellcode konfiguriert wurden.

2.2 Kinematik

Um dem Roboter die richtigen Befehle zu senden, müssen als nächstes die Grundlagen für die Kinematik gesetzt werden. Ein Roboterbein entspricht jeweils einer kinematischen Kette, die aus mehreren starren Körpern besteht und die über Gelenke verbunden sind. Es handelt sich sowohl beim Lauron als auch beim Akrobat um Rotationsgelenke. Dieser Abschnitt beschreibt Methoden der Kinematik.

Mit Hilfe der direkten Kinematik lässt sich auf Grund der Gelenkausrichtungen berechnen, an welcher Position sich der sogenannte Endeffektor befindet. Der Endeffektor entspricht dem Fuß des Roboters. Über die inverse Kinematik kann bei einer gegebenen Fußposition berechnet werden, wie die Fußgelenke gestellt werden sollen. Die beschriebenen Methoden wurden bereits in weiterer Literatur erläutert. [10]–[13]

2.2.1 Denavit-Hartenberg-Verfahren

Denavit und Hartenberg entwickelten ein Verfahren zur Beschreibung von kinematischen Ketten. Durch dieses Verfahren können mit Hilfe von nur vier Parametern, den DH-Parametern, die gegenseitige Lage zweier Elemente beschrieben werden. Dies macht die Berechnung von kinematischen Ketten wesentlich einfacher.

Die vier Parameter (Abbildung 2.3a) können wie folgt berechnet werden:

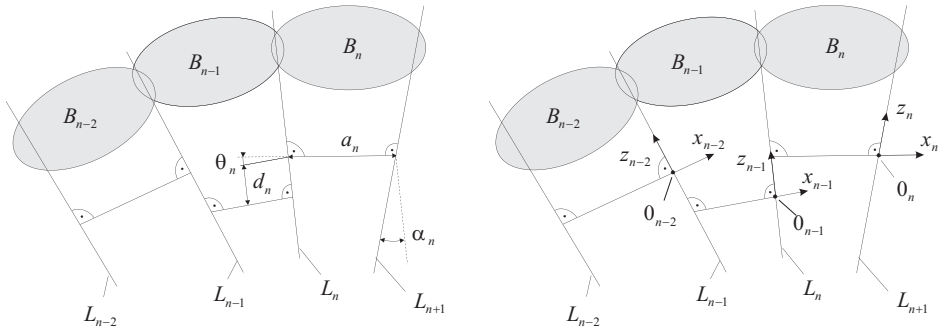
1. Entspricht der Länge der gemeinsamen Normale zwischen den Achsen L_n und L_{n+1}. Dies ist im Schaubild die Variable a_n.
2. Entspricht dem Versatz zwischen dieser und der vorherigen Normalen. Im Schaubild ist das die Variable d_n.
3. Entspricht dem Drehwinkel zwischen L_n und L_{n+1}. Im Schaubild ist das die Variable α_n.

4. Entspricht der Drehung um die Achse L_n. Im Schaubild ist dies die Variable θ_n.

Des Weiteren ist es eine Voraussetzung für das Verfahren, dass für jedes Gelenk ein eigenes Koordinatensystem definiert ist (Abbildung 2.3b), damit die Transformationsgleichungen aufgestellt werden können. Dabei ist es wichtig, dass eine Achse des Koordinatensystems auf der Gelenkachse und eine zweite auf der Normalen liegt. Damit sind nun alle Anforderungen erfüllt, um die direkte und die inverse Kinematik effizient zu berechnen.

2.2.2 Direkte Kinematik

Mit Hilfe der direkten Kinematik lässt sich nun berechnen, an welcher Position sich der sogenannte Endeffektor befindet (Abbildung 2.4). Gegeben sind hierbei die Winkel der Gelenke. Der Vektor $\vec{s} = (x, y, z, \alpha, \beta, \gamma)^T$ entspricht der Position und der Orientierung des Endeffektors. Der Vektor $\vec{q} = (q_1, ..., q_n)^T$, bei dem n die



(a) Parameter **(b)** Koordinatensysteme

Abbildung 2.3: Denavit-Hartenberg Verfahren

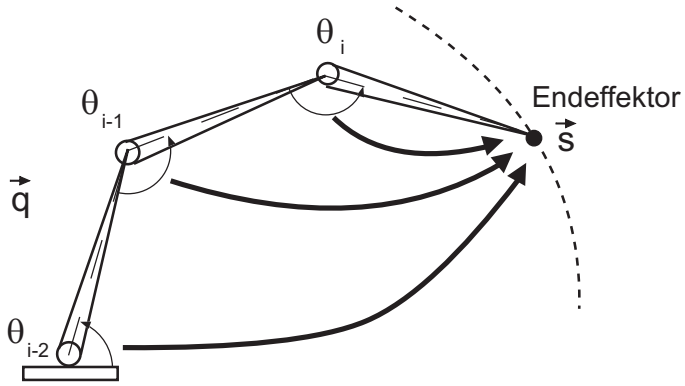


Abbildung 2.4: Direkte Kinematik

Anzahl der Gelenke ist, entspricht den Gelenkvariablen. Dann lässt sich die direkte Kinematik wie in Gleichung 2.1 ausdrücken.

$$\vec{s} = f(\vec{q}) \tag{2.1}$$

Mittels der DH-Parameter aus dem vorherigen Kapitel lässt sich nun eine homogene Transformationsmatrix vom Ausgangskoordinatensystem in das Koordinatensystem des Endeffektors abhängig der Gelenkvariablen, siehe Gleichung 2.2, erstellen.

$$^{n-1}_{n}T = \text{rot}(z_{n-1}, \theta_n) \cdot \text{trans}(0, 0, d_n) \cdot \text{trans}(a_n, 0, 0) \cdot \text{rot}(x_n, \alpha_n) \tag{2.2}$$

Der erste Parameter der Rotationsfunktion ROT bestimmt die Rotationsachse. Der zweite Parameter bestimmt den Winkel. Die Funktion TRANS entspricht einer Verschiebung im Raum. Die Gleichung setzt voraus, dass die Wahl der Koordinatensysteme wie im vorherigen Kapitel beschrieben, eingehalten wird.

Als nächstes ersetzt man die Transformationen und Rotationen in Gleichung 2.2 durch ihre homogenen Transformationenmatrizen und berechnet die gesamte Transformationsmatrix. Es ergibt sich das Ergebnis in Gleichung 2.3 und 2.4, welche den Übergang vom Gelenk $n-1$ in das Gelenk n mit Hilfe der DH-Paramter beschreibt.

$$^{n-1}_{n}T = \begin{pmatrix} \cos\theta_n & -\sin\theta_n & 0 & 0 \\ \sin\theta_n & \cos\theta_n & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & a_n \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_n & -\sin\alpha_n & 0 \\ 0 & \sin\alpha_n & \cos\alpha_n & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.3}$$

$$^{n-1}_{n}T = \begin{pmatrix} \cos\theta_n & -\sin\theta_n \cos\alpha_n & \sin\theta_n \sin\alpha_n & a_n \cos\theta_n \\ \sin\theta_n & \cos\theta_n \cos\alpha_n & -\cos\theta_n \sin\alpha_n & a_n \sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.4}$$

Da dies nur die Transformation zwischen einem und dem vorherigen Gelenk berechnet, muss nun die gesamte kinematische Kette in Gleichung 2.5 hergestellt werden. Dies erfolgt durch die Multiplikation aller Einzeltransformationen.

$$^{0}_{m}T = {}^{0}_{1}T \cdot {}^{1}_{2}T \cdot \ldots {}^{m-1}_{m}T \tag{2.5}$$

Mit dieser Matrix kann ein Punkt im Koordinatensystem des Endeffektors in das Anfangskoordinatensystem transformiert werden. Ebenfalls lässt sich durch eine Aufteilung in Teilmatrizen die Orientierung und Position des Endeffektors relativ zum Anfangskoordinatensystem berechnen. Dazu dient Gleichung 2.6.

$$ {}^{0}_{m}T = \begin{pmatrix} {}^{0}_{m}R & {}^{0}\vec{r} \\ 0 & 1 \end{pmatrix} \tag{2.6} $$

Dabei ist der Vektor ${}^{0}\vec{r}$ der gesuchte Vektor vom Koordinatensystem zum Endeffektor. Mit Hilfe der 3x3 Rotationsmatrix ${}^{0}_{m}R$, welche die Orientierung dieser Transformationen angibt, können die Winkel des Endeffektors berechnet werden. Dazu wird die Matrix noch aufgelöst. Durch das Auflösen des Gleichungssystems ergeben sich schlussendlich die folgenden Winkel in Gleichung 2.7:

$$ \begin{aligned} \alpha &= \arctan\left(\frac{\alpha_{21}}{\alpha_{11}}\right) \\ \beta &= \arctan\left(\frac{-\alpha_{31}}{\alpha_{11}\cos\gamma + \alpha_{21}\sin\gamma}\right) \\ \gamma &= \arctan\left(\frac{\alpha_{32}}{\alpha_{33}}\right) \end{aligned} \tag{2.7} $$

2.2.3 Inverse Kinematik

Mit Hilfe der inversen Kinematik lässt sich nun berechnen, wie die Gelenkswinkel ausgerichtet werden müssen, damit der Endeffektor an eine bestimmte Position

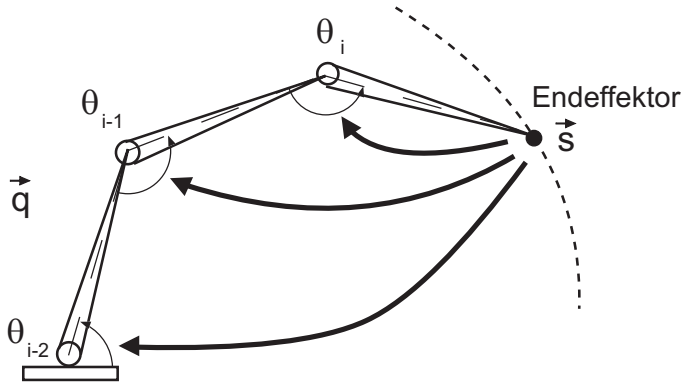


Abbildung 2.5: Inverse Kinematik

kommt (Abbildung 2.5). Dies ist gleichbedeutend mit der inversen Funktion der direkten Kinematik (Gleichung 2.8).

$$\vec{q} = f^{-1}(\vec{s}) \tag{2.8}$$

Dabei ist zu beachten, dass es in den meisten Fällen für eine Eingabe mehrere Lösungen geben kann. Eine berechnete Lösung kann aber auch ungültig sein, wenn beispielsweise der Arbeitsbereich eingeschränkt ist oder Hindernisse auf dem Weg zur gewünschten Position liegen. Ebenfalls gibt es für viele Positionen auch gar keine Lösung. Des Weiteren kann die kinematische Kette durch Singularitäten Freiheitsgrade verlieren. Dies geschieht, wenn die Drehung an unterschiedlichen Achsen dieselbe Bewegung ergibt, dadurch, dass diese aufeinander liegen oder parallel verlaufen.

Da bereits die Funktion der direkten Kinematik eine nichtlineare Funktion ist, ist auch die inverse Funktion eine nichtlineare Funktion. Das Problem gehört damit in die Kategorie der Lösung eines nichtlinearen Gleichungssystems. Dies lässt sich über eine analytisches Verfahren oder annähernd über ein numerisches Verfahren lösen.

Bei der analytischen Methode wird versucht, auf Basis der geometrischen Eigenschaften eine Formel herzuleiten. Dies wird umso schwerer, je mehr Freiheitsgrade die kinematische Kette hat. Der Vorteil liegt darin, dass die analytische Lösung ersten alle möglichen Gelenkstellungen für eine Position liefert und zweitens, dass sie wesentlich performanter ist. Die Lösungen sind exakt und Singularitäten können bereits beim Aufstellen der Gleichung festgestellt werden.

Die Software *ikfast* [14] findet Lösungen und erstellt Gleichungen für die inverse Kinematik einer kinematischen Kette. Dabei nutzt die Software im Gegensatz zu anderer Software statt der numerischen die analytische Methode.

Bei der numerischen Lösung wird mittels einer linearen Annäherung versucht die Lösung für f^{-1} zu finden. Dies lässt sich mit der Jacobi-Matrix berechnen. Dazu werden Teilfunktionen für x, y, z, α, β und γ benötigt, welche sich aus der Transformationsmatrix der direkten Kinematik ergeben. Die Teilfunktionen werden dann nach jeder Variablen partiell abgeleitet. Dies ergibt die $6\ x\ n$ Jacobi-Matrix. Sie beschreibt die Änderung der Position des Endeffektors bei Änderung der Gelenkstellungen.

Durch weitere Umstellung der Funktion ergibt sich eine Gleichung für die lineare Annäherung an die gewünschte Funktion. Diese Lösung wird dann mittels direkter Kinematik überprüft. Ist die Abweichung zu groß, wird die Jacobi-Matrix mit den zuvor berechneten Gelenkstellungen erneut berechnet. Dies ergibt die lineare Annäherung an die Lösung. Wichtig ist dabei, dass eine maximale Zahl an Iterationen für den Algorithmus angegeben ist, da es manchmal auch gar keine Lösung geben kann.

Im Gegensatz zur analytischen Methode liefert die numerische Lösung immer nur einen Wert. Ein weiteres Problem der numerischen Lösung sind Singularitäten, welche auftreten können, wenn die Jacobi-Matrix an Rang verliert und somit singulär wird. Die Folge ist, dass die Matrix nicht mehr invertierbar ist und damit keine Lösung mehr berechnet werden kann.

2.3 Laufplanung

Nun werden die Grundlagen für die Laufplanung dargestellt. Dazu werden verschiedene reguläre Laufmuster und das freie Laufmuster Random Sampling dargestellt. Die regulären Laufmuster wurden in zahlreicher Literatur bei vielen Insekten beobachtet und auf das Thema Robotik übertragen. [15], [16]

In der Robotik eignen sich diese, um auf ebenem Grund voranzuschreiten. Allerdings gibt es auch Untergründe, die stark uneben sind oder unüberwindbare Hindernisse aufweisen. Dann können freie Laufmuster wie Random Sampling zum Einsatz kommen.

2.3.1 Reguläre Laufmuster

Bei den regulären Laufmustern unterscheidet man zwischen drei häufig bei Insekten beobachteten Laufmustern, die aber alle der Gruppe *wave gait* angehören:

- slow wave gait
- ripple gait
- tripod gait

Zunächst einmal müssen zwei Begriffe definiert werden, welche für die Betrachtung der Laufmuster wichtig sind. Diese entsprechen den beiden Phasen, welche die freien Laufmuster benötigen, um nach vorne zu laufen.

- *power stroke*: In dieser Phase steht das Bein auf dem Boden und unterstützt die Körpermitte, sich nach vorne zu bewegen. Damit die Körpermitte sich nach vorne bewegen kann, wird das Bein in die entgegengesetzte Richtung gezogen, während es auf dem Boden bleibt.
- *return stroke*: Das Bein wird angehoben und nach vorne auf die nächste Position für die Ausführung der *power stroke* gesetzt.

Abbildung 2.6 zeigt nun für die drei Laufmuster zeitlich dargestellt, wann die *power stroke* und wann die *return stroke* ausgeführt wird.

Beim *slow wave gait* befindet sich immer ein Bein in der *return stroke*, während alle anderen sich in der *power stroke* befinden und den Roboterkörper stützen und nach vorne bewegen. Auch die Reihenfolge spielt eine Rolle. Zuerst werden alle Beine einer Seite umgesetzt, dann erst alle der anderen Seite.

Beim *ripple gait* werden um eine drittel Phase verschoben die Beine nacheinander umgesetzt. Daraus ergibt sich eine schnellere Bewegung als beim *slow wave gait*. Auch hier spielt die Reihenfolge eine Rolle. Es wird immer abwechselnd ein Bein

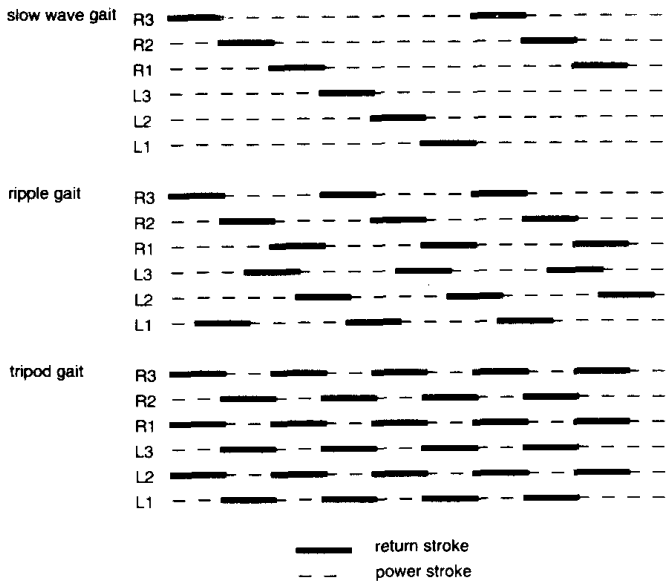


Abbildung 2.6: Reguläre Laufmuster von Insekten [15] [16]

von links und dann ein Bein von rechts umgesetzt, während die anderen Beine in der *power stroke* den Roboterkörper stützen und nach vorne bewegen.

Beim *tripod gait* sind gleichzeitig drei Beine in der Phase *power stroke*, während drei Beine in der Phase *return stroke* sind. Bei der Auswahl müssen zwei Beine der einen Seite und ein Bein der anderen Seite ausgewählt werden, damit der Roboter nicht umfällt.

Da der *tripod gait* die Phasen am schnellsten durchläuft, weil er immer die maximale Anzahl an möglichen Beinen gleichzeitig umsetzt oder stützen lässt, ist er in diesem Umfeld auch das schnellste der drei Laufmuster. Etwas langsamer ist der *ripple gait*, während der *slow wave gait* am langsamsten ist, da die Beine immer nur nacheinander umgesetzt werden.

2.3.2 Freie Laufmuster

Nun wird das freie Laufmuster *Random Sampling* sowie einige weitere relevante Algorithmen vorgestellt. [1]

Random Sampling

Beim Random Sampling werden zufällige gültige Lösungen generiert. 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 oder 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.

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

Tabelle 2.1: Wahrscheinlichkeit der Anzahl der Pfadsegmente für die geplante Strecke

Pfadsegmente	**Wahrscheinlichkeit** [%]
1	50,0
2	25,0
3	12,5
4	6,25
5	3,125

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 eine geringe Anzahl an Pfadsegmenten häufig sein sollen, da davon ausgangen wird, dass der direkte Weg der schnellste Weg ist. Höhere Anzahlen an Pfadsegmenten sollen weniger oft auftreten. Durch die Veränderung des Parameters p lassen sich die Wahrscheinlichkeiten dahingehend variieren, dass entweder eine höhere oder eine niedrigere Anzahl an Pfadsegmenten bevorzugt werden. Standardmäßig geht man von p=0,5 aus, welches die Wahrscheinlichkeitsverteilung in Tabelle 2.1 ergibt.

Festlegung der Wegpunkte Nachdem die Anzahl der Pfadsegmente definiert ist, müssen nun die Koordinaten der Wegpunkte 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 2.9 und Gleichung 2.10 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 2.7 zeigt die generierten Punkte und damit den Pfad des Roboters grafisch dargestellt.

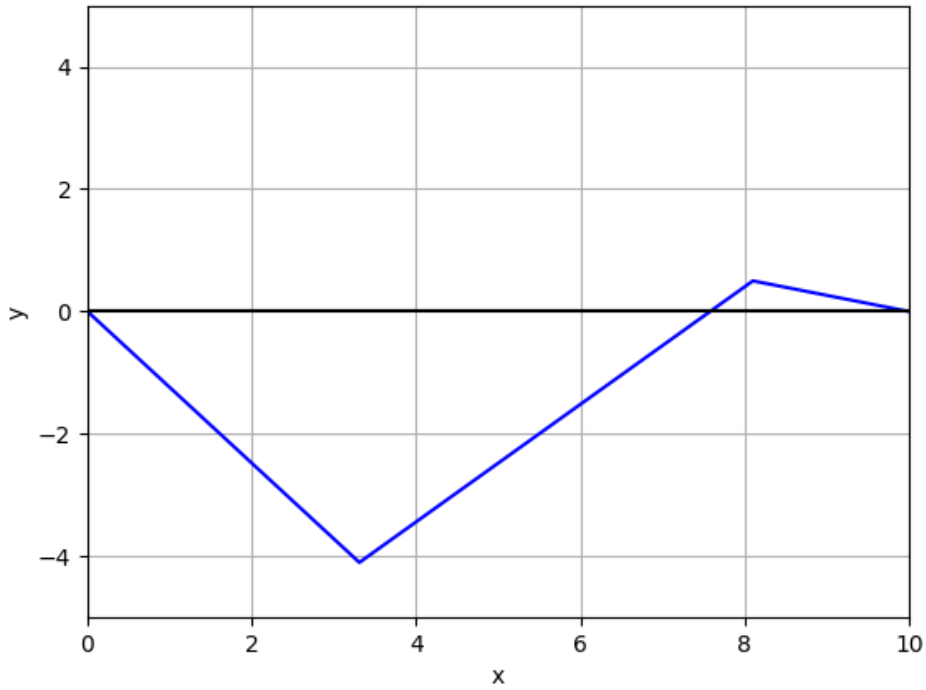


Abbildung 2.7: Zufällig generierte Wegpunkte

$$P_1 = \begin{pmatrix} x_m + x_r \\ y_m + y_r \end{pmatrix} = \begin{pmatrix} 5 - 2.31 \\ 0 - 4.1 \end{pmatrix} = \begin{pmatrix} 3.31 \\ -4.1 \end{pmatrix} \tag{2.9}$$

$$P_2 = \begin{pmatrix} x_m + x_r \\ y_m + y_r \end{pmatrix} = \begin{pmatrix} 5 + 3.1 \\ 0 + 0.5 \end{pmatrix} = \begin{pmatrix} 8.1 \\ 0.5 \end{pmatrix} \tag{2.10}$$

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.

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

Berechnung des zulässigen Bereichs Der zulässige Bereich des Mittelpunkts ist auf Grund von zwei Kriterien eingeschränkt. Zum einen durch die 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 Abbildung 2.8 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

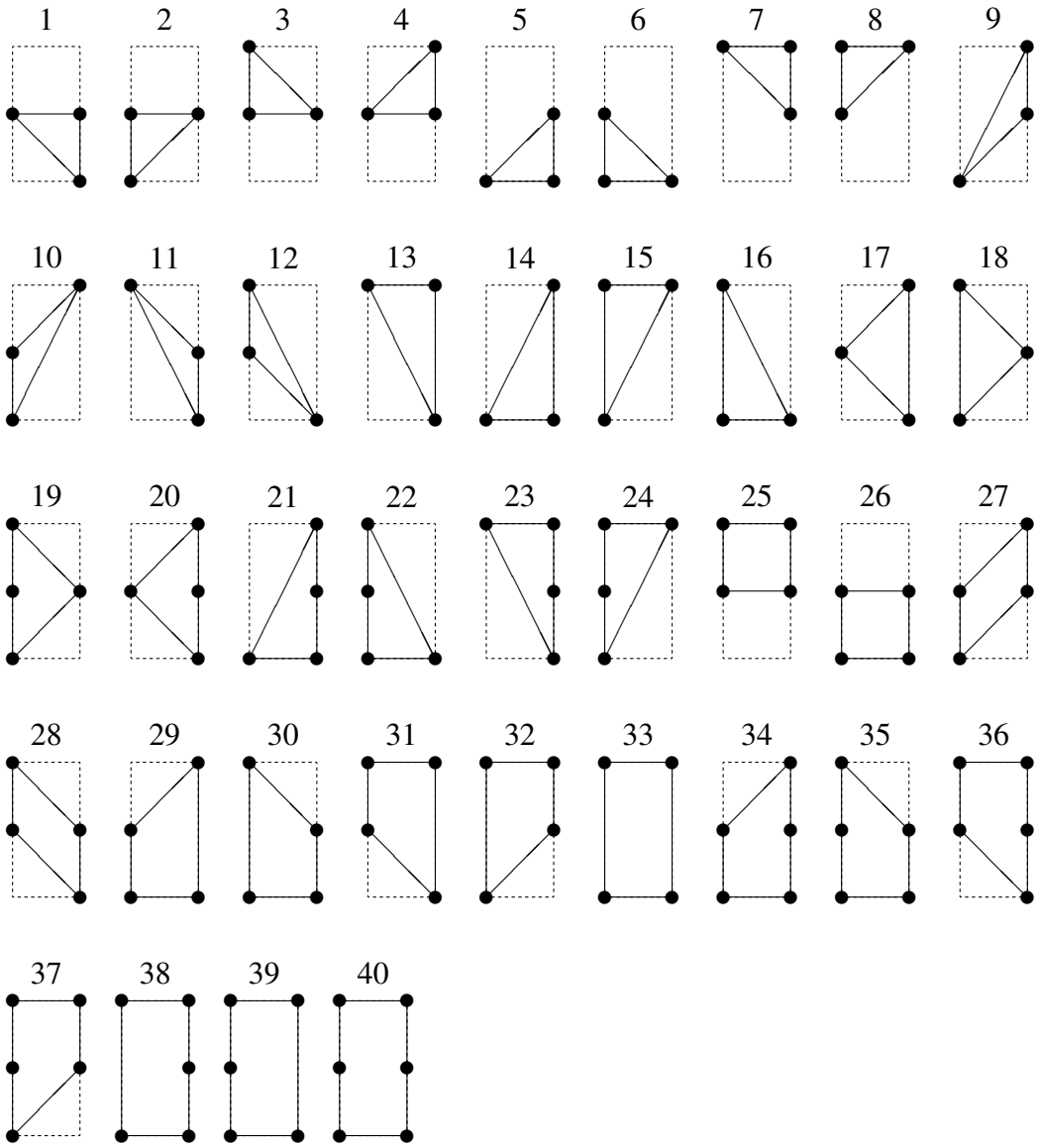


Abbildung 2.8: 40 zulässige Stützzustände [1]

Zeitdauer von 0 s angenommen wird. Damit wird der nächste Stützzustand mit dem aktuellen Stützzustand ausgeführt. Tabelle 2.2 gibt alle möglichen Stützzustände an.

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.

Tabelle 2.2: Transitionstabelle der Stützzustände [1]

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

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 wurden, 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.

Berechnungen des Mittelpunkts und der Bewegungsdauer Da nun jeder Übergang, eingeschlossen der Fußposition, der Fußkonfiguration und der zulässigen Bereich 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.

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.

Weitere Laufmuster

Neben dem Random Sampling existieren noch weitere freie Laufmuster, welche im folgenden dargestellt werden:

- *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.

2.4 Frameworks

Dieses Kapitel stellt die benötigten Frameworks für diese Arbeit dar. Das ist zum einen das ROS, welches die Basis der Entwicklung darstellt. Zum anderen ist das Gazebo, welches als Physik-Engine und 3D-Visualisierung genutzt wird. Des Weiteren wird auch noch das Vorgängersystem des Laufplaners, das OpenInventor-Framework beschrieben.

2.4.1 Robot Operating System

Für diese Arbeit ist der Aufbau und die Funktionsweise von ROS von großer Bedeutung, da der Laufplaner auf diese Plattform migriert werden soll. Diese Themen werden in zahlreicher Literatur behandelt. [17]–[19]

Das ROS ist ein Meta-Betriebssystem, welches primär auf der Linux Distribution Ubuntu und Debian verfügbar ist. Ebenfalls läuft ROS auch unter Windows Services für Linux. Es existieren auch experimentelle Versionen wie beispielsweise für Mac OS X. Das ROS besteht aus vielen wichtigen und nützlichen Komponenten. Diese werden nun Schritt für Schritt erklärt.

Packages und Nodes

Sogenannte Packages bilden die Grundlage einer jeden ROS-Entwicklung. In einem Paket werden alle nötigen Dateien gespeichert, die für das zu lösende Problem benötigt werden. Dies sind neben einem Manifest (package.xml), verschiedene Nodes, welche für die Berechnungen zuständig sind, Definitionen für Nachrichten und Services, damit die Nodes untereinander kommunizieren können sowie benötigte CMake-Konfigurationen, damit das Projekt erfolgreich kompiliert werden kann. Nodes können in C++ oder Python programmiert werden. Theoretisch sind auch andere Sprachen denkbar, allerdings haben sich bisher nur diese beiden Varianten bewährt.

Kommunikation zwischen Nodes

Die Kommunikation zwischen Nodes kann über Topics oder Services erfolgen.

Topics folgen dem Publisher-Subscriber-Prinzip. Das bedeutet, dass ein Node Nachrichten veröffentlicht, während ein anderer diesen Nachrichtenkanal verfolgt. Es können auch mehrere diesen Nachrichtenkanal verfolgen. Das Nachrichtenformat ist das msg-Format. Solche Formate werden nach Konvention im msg-Ordner abgespeichert.

Services folgen dem Client-Server-Prinzip. Hierbei handelt es sich um eine einmalige Abfrage einer Node. Der Client fragt eine Berechnung einmalig beim Server an. Das Nachrichtenformat ist das srv-Format. Auch diese werden nach Konvention im msg-Ordner gespeichert.

Jede Nachricht beinhaltet einen sogenannten Header, der aus einer Identifikationsnummer, einem Zeitstempel sowie einer Frame-ID besteht.

ROS-Master

Vor jeder Ausführung einer Node in ROS muss der sogenannte ROS-Master gestartet werden. Dieser ist die zentrale Einheit im System, welche für jegliche Lookups sowie Namensregistrierungen der Nodes zuständig ist. Ohne den ROS-Master ist keine Kommunikation zwischen Nodes möglich. Da ROS ein verteiltes System ist, erfolgt die Kommunikation netzwerktransparent. Dem entsprechend werden die Nachrichten automatisch auch über das Netzwerk geleitet. Die Konfiguration erfolgt über Umgebungsvariablen im System, welche die IP-Adresse der Systeme beinhalten.

Launch-Files

Nodes können manuell über die Kommandozeile gestartet werden. Dies wird bei einer wachsenden Zahl an Nodes und Konfigurationen immer unübersichtlicher, weshalb es sogenannte Launch-Files gibt. Diese bündeln gewisse Funktionalität, die ausführt werden sollen. Listing 2.1 zeigt ein Beispiel dafür. Mit dem include-Befehl können weitere Launch-Files eingefügt werden. Auch dies trägt dazu bei, eine bessere Modularität zu erreichen. Mit dem node-Befehl wird eine Node aus einem Package gestartet. Launch-Files bieten viele weitere Möglichkeiten wie das Setzen von Parametern, welche innerhalb der Ausführung von Nodes genutzt werden können. Außerdem können Aufrufe von Nodes weiter konfiguriert werden. Bei-

spielsweise kann mittels des output-Befehls gesteuert werden, wo die Konsolenausgaben ausgegeben werden. Der respawn-Befehl lässt eine Node neustarten, wenn sie abgestürzt ist. Der required-Befehl legt fest, dass nach Absturz diese Node die komplette Anwendung ebenfalls gestoppt werden soll.

Bag-Files

Gerade in der Roboterentwicklung ist es oftmals nicht sinnvoll, Abläufe während der Entwicklung wiederholt auf dem Roboter ausführen zu müssen. Während in der Softwareentwicklung dies normalerweise kein Problem darstellt, kann der Roboter durch eine falsche Bewegung kaputt gehen oder auf lange Sicht abgenutzt werden. Daher bietet das ROS die Möglichkeit alle Daten, die von Topics gesendet werden mittels eines sogenannten Bags aufzunehmen und erneut abzuspielen. Dies lässt sich auch sinnvoll mit einer Simulation kombinieren. Bag-Aufnahmen oder Wiedergaben können sowohl über die Kommandozeile als auch in Launch-Files ausgeführt werden.

Kommandozeilen-Tool

Nahezu alle wichtigen ROS-Dienste lassen sich über die Kommandozeile steuern. Jedes vorgestellte Konzept hat ein eigenes Interface, welches für eine unkomplizierte Interaktion sorgt:

- Nodes: rosrun
- Topics: rostopic
- Services: rosservice
- Launch-Files: roslaunch

```
1 <launch>
2   <include file="$(find hexapod)/launch/model.launch"/>
3
4   <node name="akrobat" pkg="hexapod" type="akrobat" output="screen" respawn="false"
        required="true"></node>
5 </launch>
```

Listing 2.1: Beispiel eines Launch-Files

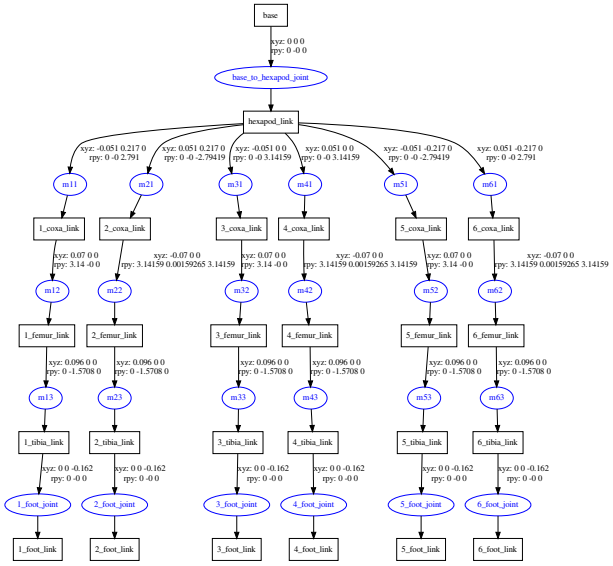


Abbildung 2.9: Darstellung des Transformationsbaums des Akrobats

- Bags: rosbag

Des Weiteren gibt es für einige Standardbefehle aus Linux Wrapper-Funktionen wie ‚ls‘und ‚cd‘, welche speziell für ROS entwickelt wurden. Diese sind ‚rosls‘und ‚roscd‘, welche den ROS-Ordner auflisten oder direkt in ein ROS-Package wechseln.

Robotermodell und TF-Framework

Robotermodelle können in einem URDF angegeben werden. Dieses Format beinhaltet jedes Segment und jedes Gelenk sowie deren Zusammenspiel. Dabei wird für jedes Segment die Visualisierung, die Massenträgheit sowie ein Kollissionsmodell angegeben. Die Visualisierung und die Kollission erfolgt beispielsweise über das Format STereoLithographie (STL), aber auch, falls möglich, über einfache geometrische Objekte. Die Gelenke beinhalten den Verweis auf das Eltern- und auf das Kindelement sowie einen Arbeitsbereich. Eine Möglichkeit das URDF-Format zu verbessern bieten XML Macros (Xacro). Xacro zielt vor allem darauf ab das Format lesbarer und kürzer zu gestalten. Über Tools wie den rviz, können Robotermodelle visualisiert werden. Mittels des Kommandozeilen-Tools urdf_to_graphiz lässt sich der Transformationsbaum eines Robotermodell visualisieren. Dies wird in Abbildung 2.9 am Beispiel des Akrobat dargestellt.

2.4.2 Gazebo

Die Entwicklung des Frameworks Gazebo von Dr. Andrew Howard und seinem Studenten Nate Koenig beginnt im Jahr 2002 an der University of Southern California. Seit dem wird das Projekt ständig weiterentwickelt. Gazebo ist eine Robotersimulation mit einer Physik-Engine und hoher Kompatibilität zu ROS. Der folgende Abschnitt soll basiernd auf Abbildung 2.10 einen Überblick über die Architektur des Frameworks geben, das für den Laufplaner eingesetzt wird. [26] [20]

Physik-Engine

Als Physik-Engine wird standardmäßig die Open Dynamics Engine (ODE) genutzt. Es sind auch weitere Physik-Engines möglich, da diese austauschbar sind. Dies liegt daran, dass sie nicht fest in der Softwarearchitektur verdrahtet sind. Alternativ zum Standard können demnach Bullet, Simbody oder DART genutzt werden.

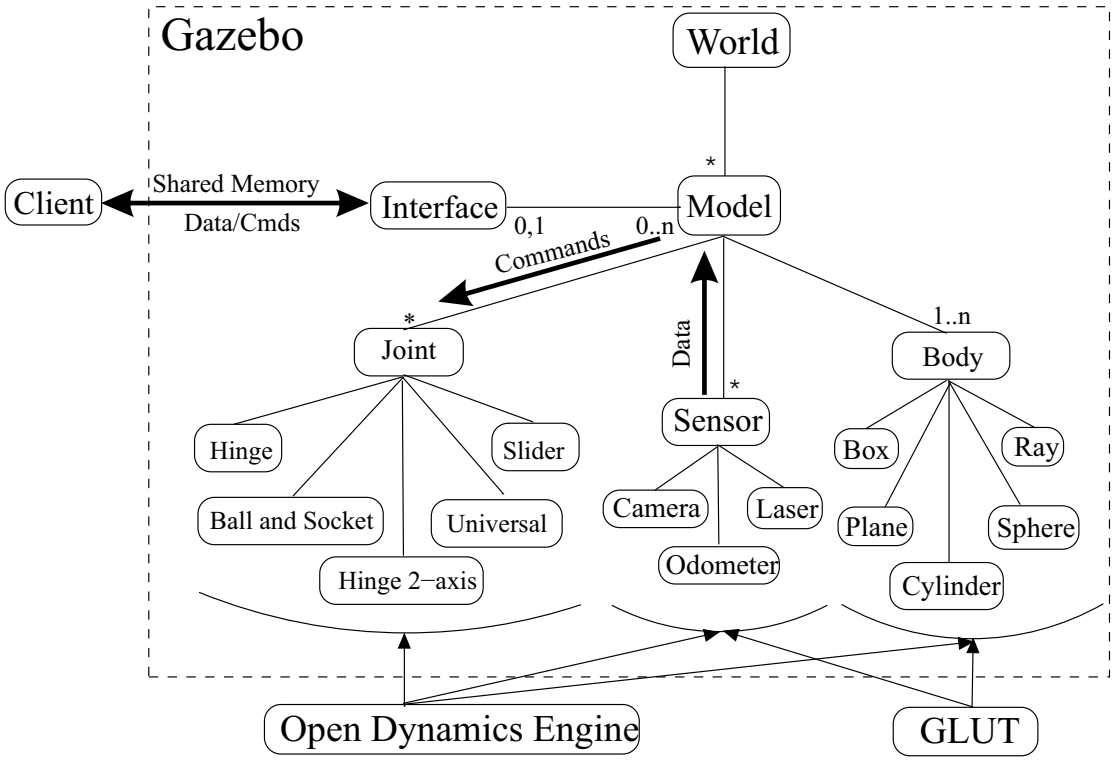


Abbildung 2.10: Architektur des Gazebo-Frameworks [20]

Visualisierung

Für die reine Visualisierung nutzt Gazebo standardmäßig OpenGL sowie GLUT (OpenGL Utility Toolkit). OpenGL ist plattformunabhängig, hochskalierbar, stabil und wird stetig weiterentwickelt. Außerdem sind viele Funktionalitäten in OpenGL auf Hardware-Ebene in der Grafikkarte entwickelt, so dass die CPU entlastet ist und andere Aufgaben schneller leisten kann.

Welt

Die Gazebo-Welt ist eine Sammlung aus verschiedenen Komponenten:

- *Modelle*: Modelle sind Objekte mit einer physikalischen Definition.
- *Körper*: Körper sind die grundlegenden geometrischen Bauteile beispielsweise des Roboters.
- *Gelenke*: Gelenke verbinden die Körper miteinander und formen damit die möglichen kinematischen Bewegungen.
- *Sensoren*: Ein Sensor in Gazebo ist ein abstraktes Gerät ohne physikalische Definition. Gazebo bietet aktuell einen Odometriesensor, einen Nähesensor sowie eine Kamera an.

Erstellung von Modellen

Aktuell müssen Modelle per Hand erstellt werden. Dies erfolgt in der Regel über das sdf-Format. Kommt ROS zum Einsatz, kann mittels der Node urdf_spawner auch das urdf-Format verwendet werden.

Abschließend ist noch wichtig, dass Gazebo durch Plugins erweiterbar ist und eine umfangreiche Schnittstelle bietet, die in einem ROS-Node mit C++ oder Python aufgerufen werden kann. Des Weiteren lässt Gazebo auch ein Rendering der Simulation auf einem Remote-Server zu, so dass verteilte Anwendungen möglich werden. Dies ist in vielen Fällen aufgrund begrenzter Rechenleistung von Vorteil.

2.4.3 Open Inventor

TODO

Ich denke, dass hier auch der Open Inventor hinein sollte. Stammt aus den 90ern von SGI. Dateiendung .iv Das wurde nach VRML weiterentwickelt und daraus ging schließlich X3D hervor. Entstanden sind daraus auch Szenengraphensysteme, etwa Open Scene Graph. Den Open Inventor gibt es heute noch. Der wird beispielsweise zur medizinischen Visualisierung benutzt (ggf. Zitat von Bernhard Preim, ich frage dazu auch gern bei Barbara Waldkirch oder Sandy Engelhardt nach. Bernhard kenne ich übrigens auch persönlich ...) Eine Alternativ-Implementierung ist Coin3D, verwendet beispielsweise bei OpenRave. Dazu gibt es eine Diss von Rosen Diankov, Aus der ecke kommt auch IKfast. Bitte nachschauen, von wem IKfast tatsächlich ist.

Kapitel 3

Verwandte Arbeiten

Dieses Kapitel stellt weitere Arbeiten dar, die sich mit dem Thema der Laufplanung von sechsbeinigen Laufrobotern mittels Random Sampling auseinandergesetzt haben.

3.1 André Herms

Andrè Herms [1] vergleicht zunächst die folgenden Algorithmen zur Laufplanung:

- Random Sampling
- Greedy Verfahren
- Branch and Bound
- Lokale Suche
- Tabu-Suche
- Simulated Annealing
- Genetische Algorithmen

Dabei nutzt Herms die folgenden Kriterien:

- Parallelisierbarkeit
- Speicherbedarf
- Anytime-Fähigkeit
- Allgemeine Anwendbarkeit

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.

Es 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 bewertet das Random Sampling anhand der genannten Kriterien. Auch hier wird nochmals deutlich, dass das Random Sampling positiv abschneidet.

Daher entscheidet die Arbeit von André Herms, dass das Random Sampling für den Laufplaner genutzt werden soll, da es zwischen allen Algorithmen am besten abschneidet.

Der von André Herms daraufhin entwickelte Laufplaner für den *Lauron III* basiert auf der 3D-Bibliothek OpenInventor [2]. Dieses Toolkit basiert wiederum auf OpenGL und ist in C++ implementiert. Des Weiteren ist es objektorientiert implementiert. 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*.

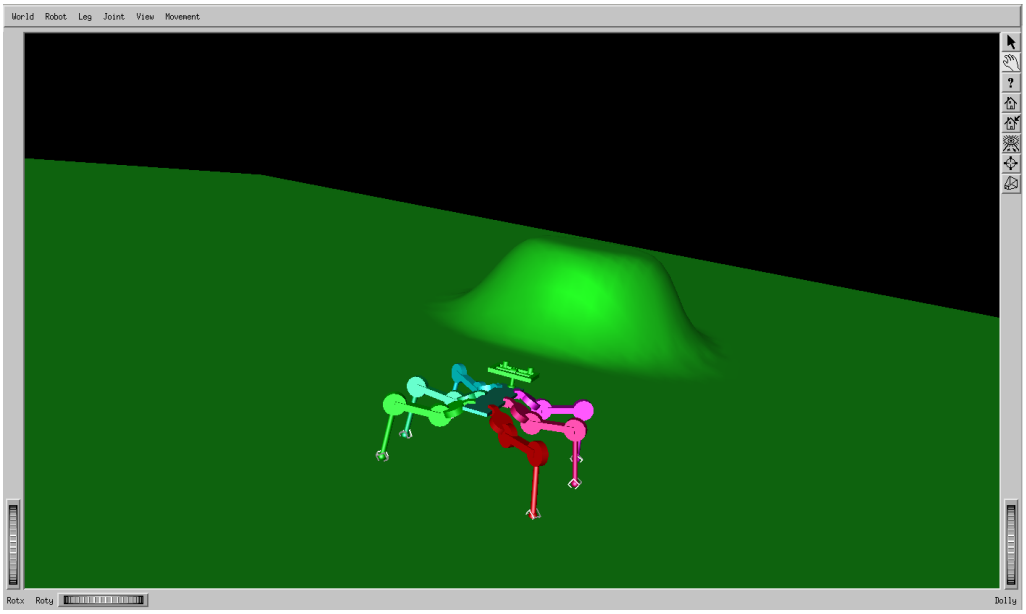


Abbildung 3.1: Screenshot aus der OpenInventor-Umgebung

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.

Eine Konfigurationsdatei existiert bereits in dieser Umgebung, die es ermöglichen soll, auch andere Robotermodelle zu testen. 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 einer Bitlogik gespeichert. Dies ist sehr nützlich um Speicherplatz zu sparen, macht auf der anderen Seite das Finden von Fehlern allerdings schwieriger.

3.2 Uli Ruffler

Uli Ruffler [3] hat den Laufplaner auf eine inkrementelle Arbeitsweise angepasst. Des Weiteren hat er die Basis für die Stereobildverarbeitung gelegt. Der Laufplaner kann nun während des Laufens neue Lösungen generieren, allerdings werden diese Lösungen noch nicht verwendet.

Kapitel 4

Konzeption der Lösung

Dieses Kapitel nutzt die in Kapitel 3 dargestellten verwandten Arbeiten, um ein Konzept für die Portierung des Laufplaners für den Akrobat in ROS und Gazebo formal darzustellen. Als Basis dafür werden die Arbeiten von André Herms [1] und Uli Ruffler [3] herangezogen. Für die Darstellung einer Lösung sind zwei Aspekte von Bedeutung. Dies ist zum einen die Simulationsumgebung. Zum anderen ist dies der Laufplanungsalgorithmus. Beide Aspekte können getrennt voneinander betrachtet werden.

4.1 Simulationsumgebung

Damit die Simulationsumgebung möglichst nahe an der Software des Laufroboters Akrobat sein soll, soll in beiden Fällen das ROS genutzt werden. Um die reale Umgebung zu simulieren, benötigt die Lösung eine Physik-Engine, die bei André Herms [1] Simulationsumgebung nur rudimentär vorhanden ist. Bei der Nutzung von Gazebo wird standardmäßig die Open Dynamics Engine (ODE) mitgeliefert, welche auch hierbei zum Einsatz kommen soll.

Zusätzlich soll es möglich sein, verschiedene Umgebungen laden zu können, damit die Flexibilität beim Testen der Laufalgorithmen erhalten bleibt. Dies ist mit Hilfe von verschiedenen Gazebo-Welten zu lösen. Im ersten Schritt soll dazu eine leere Welt genutzt werden.

Um außerdem ein möglichst reales Szenario zu simulieren, muss ein präzises Robotermodell im Format URDF für das ROS und Gazebo zum Einsatz kommen, welches das Aussehen, das Kollissionsmodell sowie die Massenträgheit genau be-

schreibt. Der bestehende Laufplaner besitzt aktuell noch das Modell des *Lauron III*, welches in der neuen Lösung keine Anwendung mehr findet.

Des Weiteren sollen die Gelenkbewegungen nicht mehr über OpenInventor [2] ausgeführt werden. Dazu soll ein ROS-spezifisches Paket mit dem Namen *ros_control* verwendet werden. Durch dieses werden Motoren an den Gelenken simuliert, welche über ROS-Topics angesteuert werden können.

4.2 Laufplanung

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 und Uli Ruffler eine xml-Schnittstelle, welche Bewegungen repräsentiert. Die hier dargestellte Schnittstelle soll eine Abwandlung der vorherigen Schnittstellen sein, da diese nicht die Dauer von Bewegungen speichert, sondern lediglich für jeden Schritt die Fußpositionen relativ zur definierten Ausgangsposition. Wie im Vorgängermodell sollen Bewegungen generiert und auch eingelesen werden können.

Der vorherige Laufplaner hat kinematische Berechnungen manuell in OpenInventor durchgeführt. Das neue System soll das ROS-Framework TF verwenden, um möglichst einfach Koordinatentransformationen zu berechnen. Die inverse Kinematik wird weiterhin über die analytische Methode berechnet. Allerdings werden dabei die Objektfunktionen von TF verwendet. Weitere geometrische Berechnungen werden ebenfalls über das TF-Framework berechnet.

Des Weiteren muss der Algorithmus auf den Laufroboter Akrobat angepasst werden, da der vorherige Algorithmus auf dem *Lauron III* basiert und die Maße andere sind.

Kapitel 5

Implementierung in ROS und Gazebo

Dieses Kapitel stellt eine Implementierung für das in Kapitel 4 dargestellte Konzept der Entwicklung einer Simulationsumgebung und der Portierung des Algorithmus Random Sampling zur Laufplanung für den Akrobat dar.

5.1 Vorgehensweise und Aufbau des Pakets

Da eine komplette Kopie des vorherigen Algorithmus zur Laufplanung 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 des Pakets
    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 des Laufplaners erst zum Schluss zu beginnen und mit dem Aufsetzen der Simulation zu beginnen. Der Grund dafür ist, dass es damit während der Portierung 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 5.1 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 [27] sowie das Hopper-Paket [28] angelehnt. Beide Pakete nutzen wichtige Konzepte für das Robotermodelle und die Gelenksteuerung, die auch für dieses Projekt wichtig sind.

5.2 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.

5.2.1 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 5.2 zeigt stellt dies sowie die Koordinatensystemen des Roboters dar.

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:

```
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 5.1: Dateibaum des ROS-Pakets

- Aufsetzen des Kollisionsmodells durch das Attribut <collission>
- Aufsetzen der Massenschwerpunkte und der Trägheitsmomente 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 [21] runterzurechnen, so dass sie später besser von Gazebo verarbeitet werden können, da sie optimiert sind.

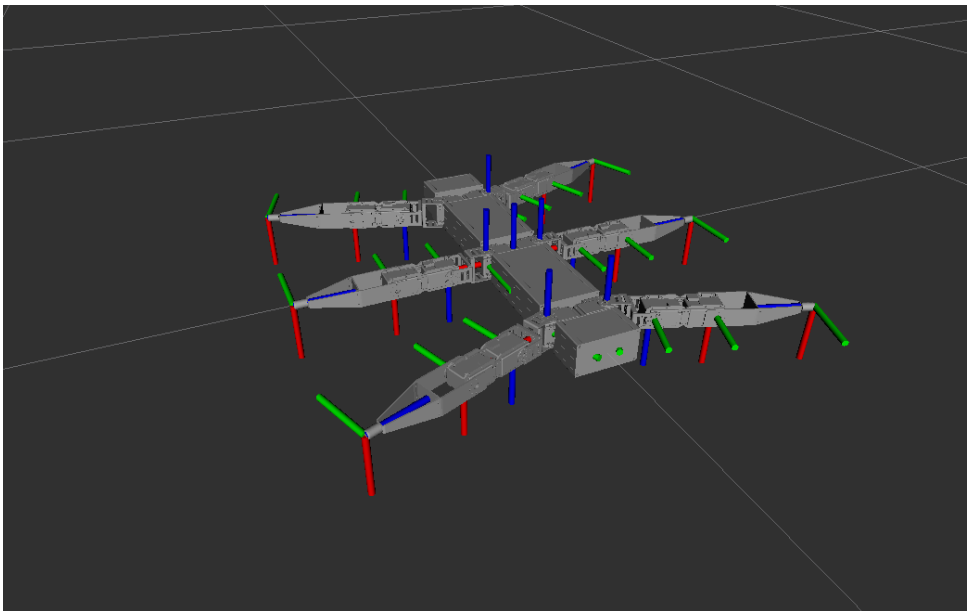


Abbildung 5.2: Darstellung des Akrobats im rviz

Aufsetzen der Massenschwerpunkte und der Trägheitsmomente

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 [29]. Abbildung 5.3 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

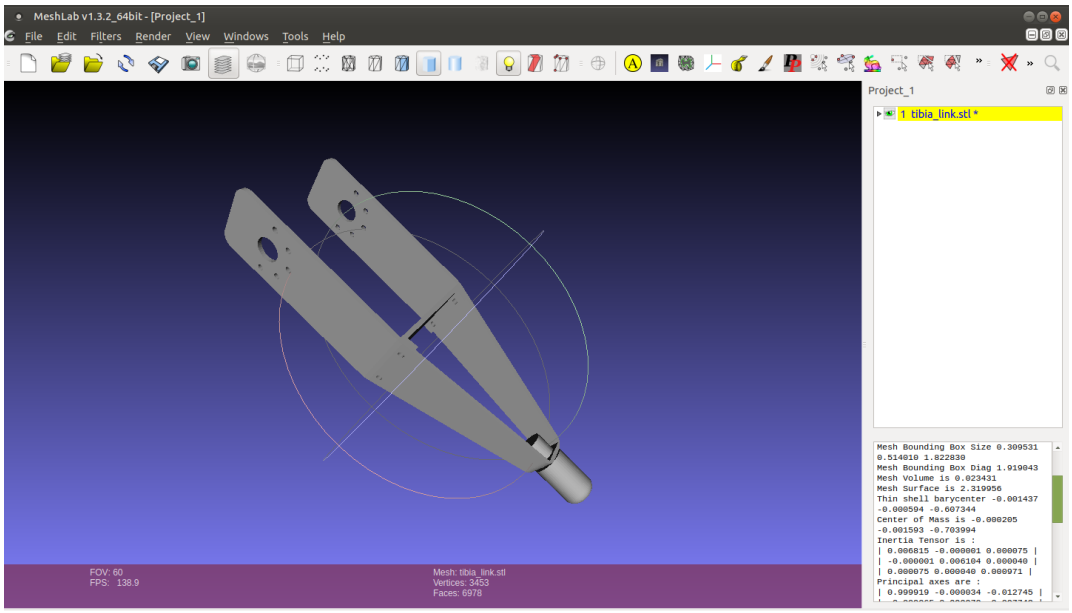


Abbildung 5.3: Auslesen der Trägheitsmomente mit MeshLab

5.2.2 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 5.4 zu sehen ist.

5.2.3 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

Damit ist das Aufsetzen des Roboters sowie der Simulationsumgebung vollständig. Der Roboter wird nun in Gazebo angezeigt, wie in Abbildung 5.5a zu sehen ist.

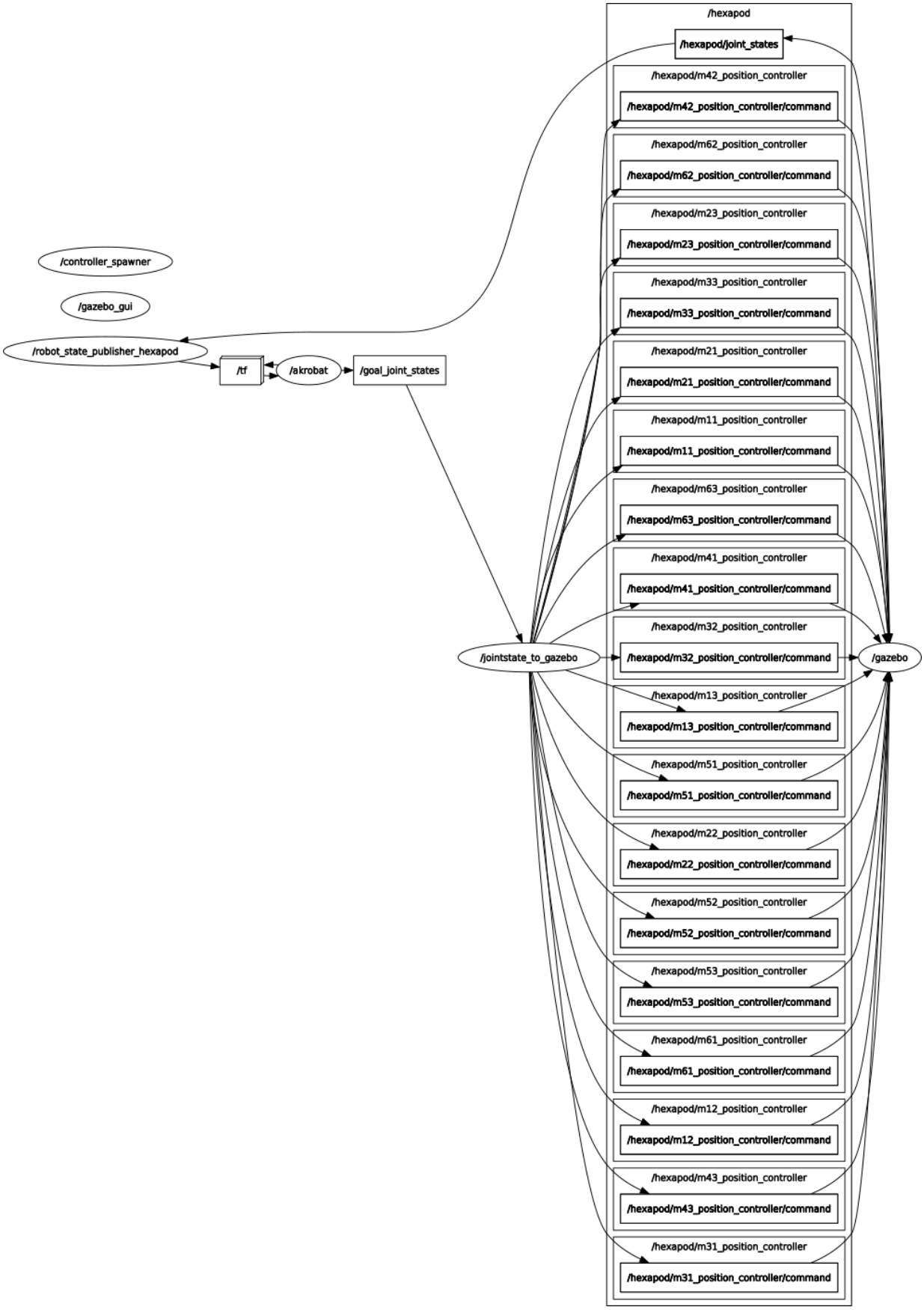


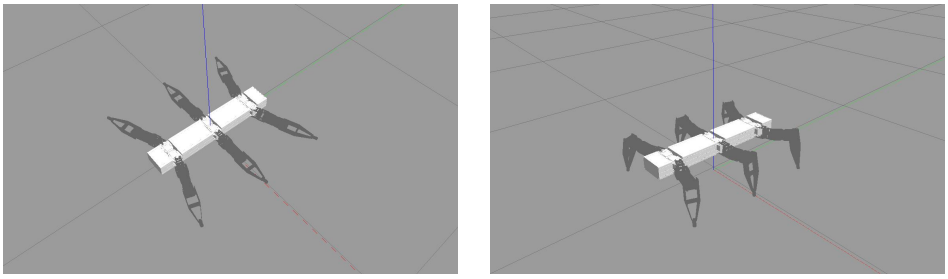
Abbildung 5.4: ROS-Topics der Simulation

5.3 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 5.5b 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 erlaubten Winkel befinden. Ist das nicht der Fall, wird eine Warnung ausgegeben.



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

Abbildung 5.5: Aufstehen des Akrobat in Gazebo

```
<?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 5.1: Aufbau der Bewegungsdatei

5.4 Generieren und Einlesen von Bewegungen als xml-Datei

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 5.1 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 [30] 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.

5.5 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 Tat-

sache, dass die xml-Datei keine Bewegungsdauern speichert, muss eine 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.

$$f(x) = -0,5 \cdot \cos(x) + 0,5 \tag{5.1}$$

$$f(y) = -0,5 \cdot \cos(y) + 0,5 \tag{5.2}$$

$$f(z) = -0,5 \cdot \cos(z) + 0,5 \tag{5.3}$$

Der in Gleichung 5.1, 5.2 sowie 5.3 verwendete Cosinus ist skaliert und vertikal verschoben, so 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. Die 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 Zwischenpositionen 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 5.2 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 5.2: 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.

5.6 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.

5.7 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.

5.7.1 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.

5.7.2 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 weitere definierte Angaben verändert werden:

- Angepeilte Zielposition des Roboters
- Schrittgröße als potentielle Position für das Absetzen des Fußes
- Abschnittslänge für die spiralförmige Suche beim Absetzen des Fußes
- Minimal gültiger Stability-Margin
- Maximale Anzahl von Anhebungen oder Absetzungen von Füßen

5.7.3 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: Addition, Subtraktion, Multiplikation, Division
- Länge der Vektors mittels *length*
- Normalisierung von Vektoren mittels *normalise*
- Distanz zweier Punkte mittels *distance*

5.7.4 Veränderung der Zufallszahlengenerierung

Für die Generierung von Zufallszahlen steigt das System auf den Pseudozufallszahlengenerator Mersenne-Twister [22] um. Außerdem wird beispielsweise die geome-

trische 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 5.3.

```
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 5.3: Geometrische Verteilung mittels C++

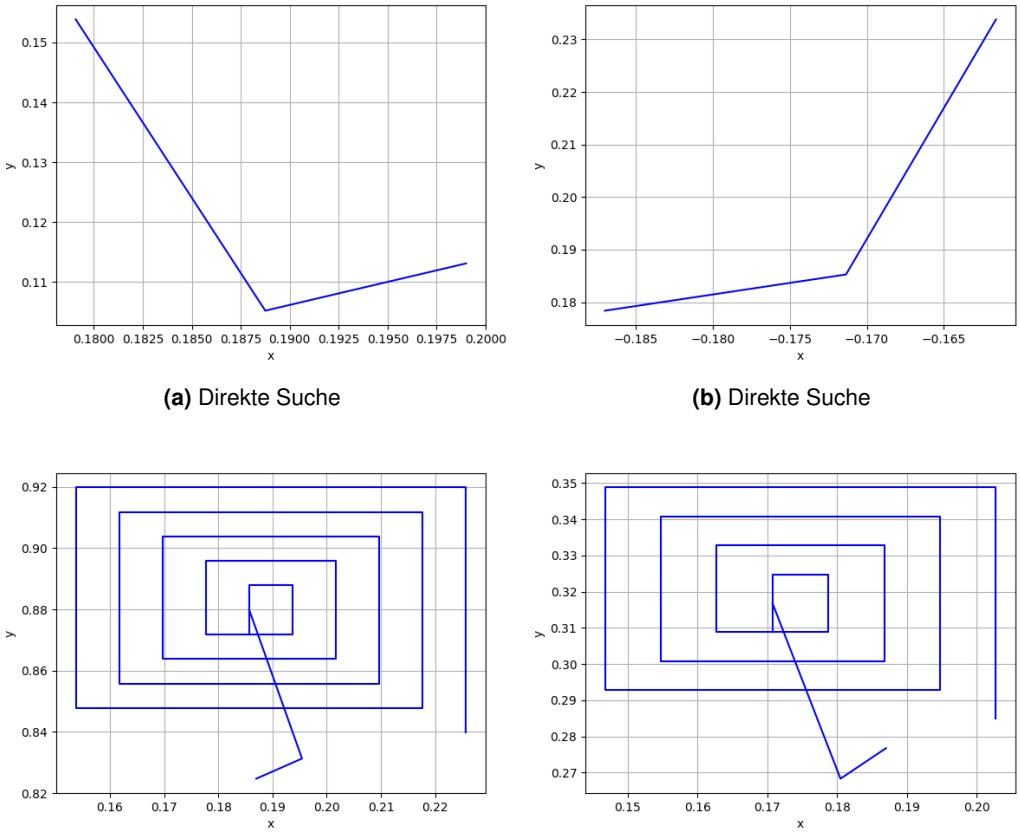
Kapitel 6

Testen der Ergebnisse

Dieses Kapitel testet die Implementierung des nach ROS und Gazebo migrierten Laufplaners. Zum einen können die Ergebnisse durch die Visualisierung in Gazebo überprüft werden. Eine weitere Möglichkeit der Auswertung, welche sich auch zur Fehleranalyse eignet, ist die Nutzung eines Frameworks, um mathematische Darstellungen anzufertigen. Dazu nutzt diese Arbeit das auf Python basierende Framework matplotlib [23]. Dieses wird mit Hilfe einem Header für C++ eingebunden, welcher die Befehle im Hintergrund in Python ausführt. Damit lassen sich unter anderem Bewegungen der Füße und des Roboterkörpers in einem Graphen darstellen.

Das Random Sampling erfolgt wie in Kapitel 3 beschrieben. Dabei gibt es einige Schritte, die bei der Implementierung fehlerhaft sein können, da sie auf verschiedenen Robotern mit anderen Werten ablaufen müssen.

Die erste Herausforderung ist das Absetzen eines Fußes. Während das Anheben eines Fußes in der Regel immer möglich ist, kann es beim Absetzen zu Problemen kommen. Das Ziel ist es, den Fuß in eine neue gültige und zufällige Position zu bringen. Diese sollte nach Möglichkeit in Richtung Ziel liegen, damit der Roboter voranschreiten kann. Bei der Generierung der Position wird ausgehend von einer Startposition ein Vektor addiert, der leicht in Richtung Ziel verschoben ist. Danach wird ein weiterer zufälliger Vektor addiert. Dann wird versucht, den Fuß an dieser Stelle abzusetzen. Ist diese Position nicht gültig, wird mit Hilfe einer Spiralsuche eine bessere Position gesucht. Die Implementierung erforderte im Gegensatz zum Lauron andere Werte für Vektorlänge und den Abständen der Spiralsuche, um genau solche Positionen zu finden, die den Roboter besser zum Ziel bringen können.



(a) Direkte Suche

(b) Direkte Suche

(c) Spiralsuche

(d) Spiralsuche

Abbildung 6.1: Auswertung der Positionsfindung beim Absetzen eines Fußes

Sind diese Werte angepasst, ergeben sich wie in Abbildung 6.1a und Abbildung 6.1b in den meisten Fällen bereits gültige Fußpositionen, ohne die Spiralsuche anwenden zu müssen, vorausgesetzt das Weltmodell ist flach. Selten kommt es dazu, dass eine Spiralsuche durchgeführt werden muss. Beim Ablauf der Sprialsuche wird wie in Abbildung 6.1c und Abbildung 6.1d in der Regel immer ein gültiges Ergebnis gefunden. Nur sehr selten kommt es vor, dass ein Fuß keine Absetzposition findet. Ist dies der Fall, soll der Algorithmus die generierte Lösung verwerfen.

Eine weitere Herausforderung ist es, ob der Algorithmus es insgesamt schafft, gültige Lösungen zur anvisierten Zielposition zu generieren. In folgendem Beispiel versucht der Laufplaner für den Akrobat gültige Lösungen für die Bewegung von der Startposition $S(0,0)$ zur Zielposition $Z(0,1)$ zu generieren. Dies entspricht der Bewegungen von einem Meter. Dazu wird eine vollständig flache Welt genutzt. Abbildung 6.2 zeigt drei nacheinander ausgeführte Generierungen von Lösungen für dieses Problem. Ferner zeigt es die einzelnen Fußbewegungen sowie die möglichen

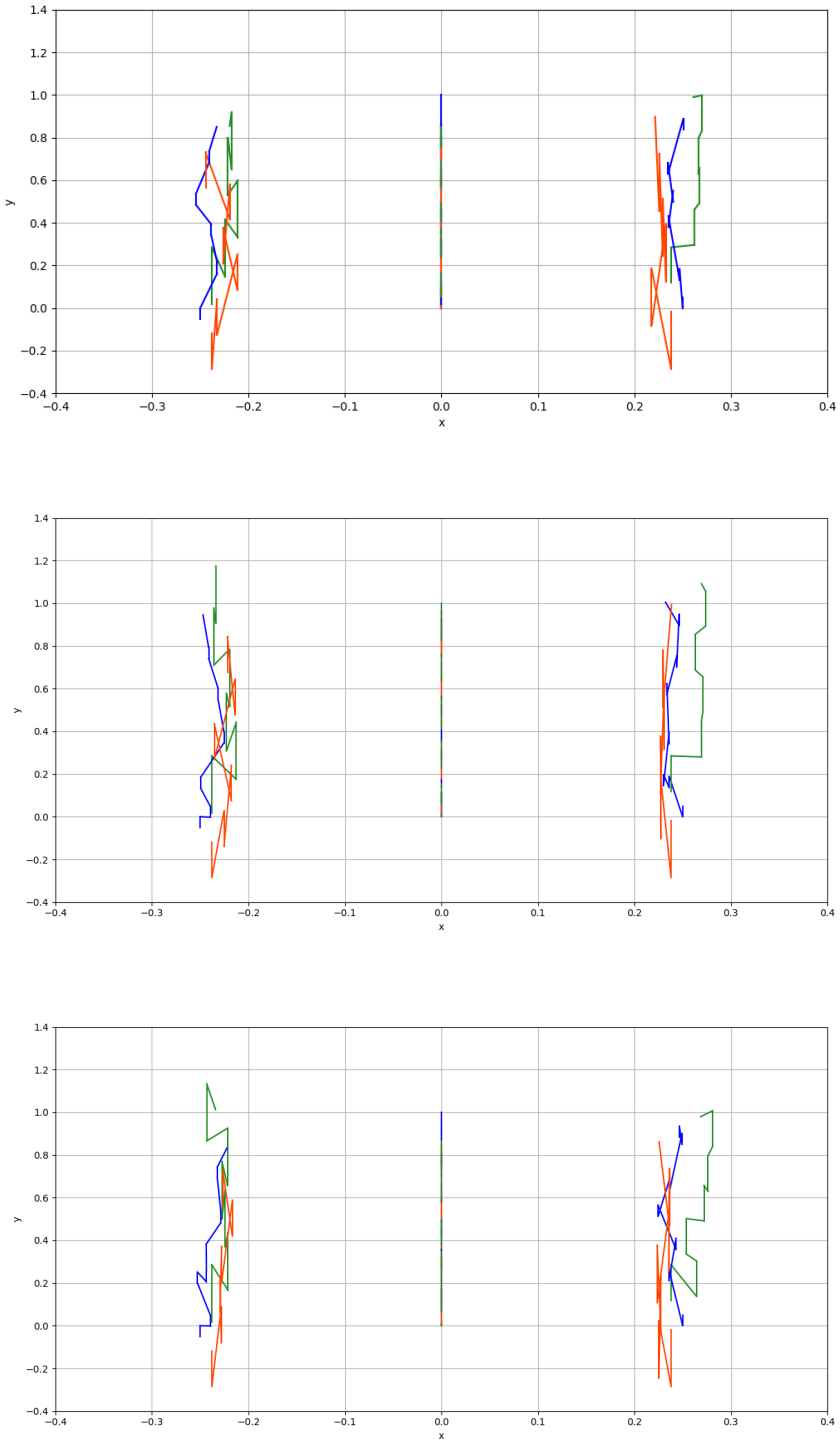


Abbildung 6.2: Auswertung von Bewegungen von Körper und Füßen

Bereiche, in denen sich die Körpermitte bewegen darf. Das Resultat ist in allen drei Fällen erfolgreich. Dass dies funktioniert, bestätigt ebenfalls, dass die Auswahl der nächsten Fußänderung, also die Entscheidung welcher Fuß angehoben oder abgesetzt wird, mittels der Glücksradauswahl geeignete Entscheidungen trifft, die nicht dafür sorgen, dass die Bewegungen sich gegenseitig aufheben.

Kapitel 7

Zusammenfassung

Das Ziel dieser Arbeit war es den Laufplaner für den Lauron III von André Herms, welcher über die OpenInventor-Simulation entwickelt wurde, für den Akrobat auf ROS und Gazebo bereitzustellen.

Um dieses Ziel zu erreichen, mussten in Kapitel 2 die Grundlagen gelegt werden. Das Kapitel beschäftigt sich zunächst mit den beiden relevanten Laufrobotern Lauron und Akrobat, um die Unterschiede, welche für eine Portierung wichtig sind, herauszuarbeiten. Danach wurden die Grundlagen für die direkte und inverse Kinematik gelegt, welche für die Fußsteuerung des Laufroboters benötigt werden. Danach geht das Kapitel auf verschiedene Arten der Laufplanung ein. Zuletzt führt das Kapitel noch in die Zielsysteme Robot Operating System (ROS) und Gazebo ein.

Danach werden die verwandte Arbeiten von André Herms [1] und Uli Ruffler [3] in Kapitel 3 dargestellt. Diese werden als Basis genutzt, um den neuen Laufplaner für den Akrobat in ROS und Gazebo zu entwickeln. Kapitel 4 stellt ein Konzept zur Entwicklung einer solchen Software dar. Kapitel 5 stellt die mögliche Implementierung dieses Konzepts dar.

Zum Abschluss werden die generierten Bewegungen des Random Samplings, insbesondere die Fuß- und Körperbewegungen, in Kapitel 6 grafisch mit Hilfe der Bibliothek matplotlib ausgewertet.

Kapitel 8

Ausblick

Nachdem der Laufroboter in der Simulation getestet wurde, sollte dieser auch in einer realen Umgebung getestet werden, da sich dort in der Regeln weitere Herausforderungen deutlich machen, da die Simulation von idealen Bedingungen ausgeht. Dies ist zwar schon insofern gegeben, dass Gazebo eine Physik-Engine bereitstellt, trotzdem sollte der Laufplaner auf weitere Unterschiede zur realen Umgebung untersucht werden.

Des Weiteren ist ein nächster Schritt den portierten Laufplaner auch auf weiteren Geländeformen wie beispielsweise auf unebenen Gelände zu testen. Auch bietet es sich an zu untersuchen, wie der Algorithmus sich bei unüberwindbaren Hindernissen verhält. Durch die Streckenplanung kann der Roboter einen Weg um das Hindernis planen.

Aktuell werden die Fußbewegungen mittels einer trigonometrischen Funktion interpoliert. Dies kann auch durch eine Spline-Interpolation wie bei Jörg Fellmann [13] umgesetzt werden. Diese Form der Interpolation bietet einige Vorteile und lässt die Bewegungen realer aussehen.

Außerdem könnte der Roboter in der Simulation noch durch einen Sensor am Kopf ausgestattet werden, welcher die Umgebung verarbeitet und Hindernisse an den Algorithmus weitergibt. Auch könnten über die Höheninformationen die Füße richtig auf dem Boden platziert werden, da der Algorithmus aktuell noch nicht weiß, wo sich dieser befindet.

Abkürzungsverzeichnis

ROS Robot Operating System

URDF Unified Robot Description Format

STL STereoLithographie

Xacro XML Macros

ODE Open Dynamics Engine

MCA Modular controller architecture

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literatur

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

[2] J. Wernicke, „The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2“, *Open Inventor Architecture Group, Addison-Wesley*, 1994.

[3] U. Ruffler, „Laufplanung basierend auf realitätsnahen Umgebungsdaten für einen sechsbeinigen Laufroboter“, Diplomarbeit, Hochschule Mannheim, 2006.

[4] B. Gaßmann, „Erweiterung einer modularen Laufmaschinensteuerung für unstrukturiertes Gelände“, Diplomarbeit, Universität Karlsruhe, 2000.

[5] R. Troilo, „Sensorgesteuerte Bewegung und Bewegungsplanung für ein Roboterbein“, Diplomarbeit, Hochschule Mannheim, 2007.

[6] K. Berns, S. Cordes und W. Ilg, „Adaptive, neural control architecture for the walking machine LAURON“, in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'94)*, IEEE, Bd. 2, 1994, S. 1172–1177.

[7] M. Ziegenmeyer, A. Rönnau, T. Kerscher, J. M. Zöllner und R. Dillmann, „Die sechsbeinige Laufmaschine LAURON IVc“, in *Autonome Mobile Systeme 2009*, Springer, 2009, S. 225–232.

[8] H. Cruse, „The function of the legs in the free walking stick insect, Carausius morosus“, *Journal of Comparative Physiology*, Jg. 112, Nr. 2, S. 235–262, 1976.

[9] W. Askerow, „Konstruktion, Aufbau und Inbetriebnahme einer sechsbeinigen Laufmaschine unter Verwendung inverser Kinematik“, Bachelorarbeit, Hochschule Mannheim, 2014.

[10] J. Denavit und R. S. Hartenberg, „A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices“, in *ASME Journal of Applied Mechanics*, Bd. 22, 1955, S. 215–221.

[11] D. W. Wloka, *Roboter Systeme 1*, 1. Aufl. Springer, 1992.

[12] T. Ihme, „Steuerung von sechsbeinigen Laufrobotern unter dem Aspekt technischer Anwendungen“, Diss., Otto-von-Guericke-Universität Magdeburg, 2002.

[13] J. Fellmann, „Entwicklung eines statisch stabilen Laufalgorithmus für einen zweibeinigen Laufroboter“, Diplomarbeit, Hochschule Mannheim, 2007.

[14] R. Diankov, „Automated Construction of Robotic Manipulation Programs“, Diss., Carnegie Mellon University, Pittsburgh, PA, Sep. 2010.

[15] C. Ferrell, „A comparison of three insect-inspired locomotion controllers“, *Robotics and autonomous systems*, Jg. 16, Nr. 2-4, S. 135–159, 1995.

[16] D. M. Wilson, „Insect walking“, *Annual review of entomology*, Jg. 11, Nr. 1, S. 103–122, 1966.

[17] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler und A. Y. Ng, „ROS: an open-source Robot Operating System“, Jg. 3, Nr. 3.2, S. 5, 2009.

[18] A. Martinez, *Learning ROS for Robotics Programming*. Packt Publishing Ltd., 2013, Bd. 2.

[19] J. M. O'Kane, *A Gentle Introduction to ROS*. Department of Computer Science und Engineering, 2013, Bd. 2.1.2.

[20] N. Koenig und A. Howard, „Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator“, in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, Sep. 2004, S. 2149–2154.

[21] 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.

[22] 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.

[23] P. Barrett, J. Hunter, J. T. Miller, J.-C. Hsu und P. Greenfield, „matplotlib–A Portable Python Plotting Package“, in *Astronomical data analysis software and systems XIV*, Bd. 347, 2005, S. 91.

Web-Dokumente

[24] *Modular Controller Architecture MCA2*, (Letzter Zugriff am 14.07.2020). Adresse: http://mca2.sf.net.

[25] *FZI Forschungszentrum Informatik, Abteilung: Interaktive Diagnose- und Servicesysteme*, (Letzter Zugriff am 20.06.2020). Adresse: https://www.fzi.de/.

[26] *Gazebo - Robot simulation made easy*, (Letzter Zugriff am 21.06.2020). Adresse: http://gazebosim.org.

[27] *Akrobat - Control and visualization of a six-legged walking robot based on ROS.* (Letzter Zugriff am 10.06.2020). Adresse: https://github.com/informatik-mannheim/akrobat.

[28] *Hopper Project for Gazebo using ROS Developement Studio*, (Letzter Zugriff am 10.06.2020). Adresse: https://bitbucket.org/theconstructcore/hopper.

[29] *Inertial parameters of triangle meshes*, (Letzter Zugriff am 11.06.2020). Adresse: http://gazebosim.org/tutorials?tut=inertia&cat=build_robot.

[30] *Pugixml - Light-weight, simple and fast XML parser for C++ with XPath support*, (Letzter Zugriff am 11.06.2020). Adresse: https://pugixml.org.