UNIVERSITÄT DORTMUND

Planung und Implementierung eines evolutionären Ansatzes zur Steuerung eines zweibeinigen Roboters

Ralf Kosse

Algorithm Engineering Report
TR06-2-010
Dezember 2006
ISSN 1864-4503

Universität Dortmund
Fachbereich Informatik
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

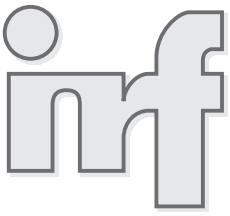
Diplomarbeit

Planung und Implementierung eines evolutionären Ansatzes zur Steuerung eines zweibeinigen Roboters

Ralf Kosse
r.kosse@gmx.de
September 2006

Gutachter:
Prof. Dr.-Ing. Uwe Schwiegelshohn
Prof. Dr. Günter Rudolph

Institut für Roboterforschung
Abteilung Informationstechnik
der Universität Dortmund



Lehrstuhl Informatik 11
Algorithm Engineering
der Universität Dortmund

Erklärung

Hiermit bestätige ich, die vorliegende Diplomarbeit selbständig und nur unter Zuhilfenahme der angegebenen Literatur verfasst zu haben.

Ich bin damit einverstanden, dass Exemplare dieser Arbeit in den Bibliotheken der Universität Dortmund ausgestellt werden.

Dortmund, 21. September 2006

Ralf Kosse

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit widmet sich der Entwicklung und Optimierung eines Laufmusters für einen humanoiden Roboter, der sich auf zwei Beinen in seiner Umwelt fortbewegen soll. Humanoide Roboter haben ihre Form in Anlehnung an das Aussehen und die physischen Eigenheiten des menschlichen Körpers erhalten. Sie können sich in der Umgebung des Menschen besser bewegen und verhalten als rollende Roboter, da der Mensch sein Umfeld in Bezug auf seine körperlichen Gegebenheiten angepasst hat. So sind z.B. Treppen für radgetriebene Roboter schwer zu überwinden; viele Werkzeuge des Menschen sind auf seine fünffingerige Hand ausgerichtet.

Um die Forschung im Bereich der humanoiden Roboter weiter zu fördern, wurde in der internationalen Weltmeisterschaft im Roboterfußball, dem Robocup [Rob06], eigens eine Liga für humanoide Roboter [Hum06] eingerichtet. Der Verfasser dieser Arbeit nahm mit dem Team BreDoBrothers [Bre06] am Robocup 2006 in Bremen teil, dabei trat der hier behandelte Kondo KHR-1 des Instituts für Roboterforschung der Universität Dortmund im Roboterfußball gegen andere humanoide Roboter internationaler Teams an.

1.1 Motivation

Praktisch alle vom Menschen konstruierten beweglichen Objekte an Land bewegen sich auf Rädern. Dagegen gibt es so etwas wie Räder in der Natur überhaupt nicht. Es ist einerseits schnell ersichtlich, dass wachsende Organismen keine Räder entwickeln können, da diese durch ihre Eigendrehung nicht über solche „Versorgungsleitungen“ wie Blutbahnen (bei Lebewesen) oder Kapillaren (bei Pflanzen) versorgt werden können. Aber ein noch wichtigerer Grund ist, dass Räder auf einen relativ ebenen und auch stabilen Untergrund angewiesen sind.

Radgetriebene Fahrzeuge können schlecht in felsigen Regionen fahren, bleiben leicht in weichem Untergrund stecken und können nur in aufwändigen Spezialausfertigungen auf Bäume klettern. Zudem können Beine eine höhere Traktion erreichen, auf bestimmten Untergründen (z.B. Sand oder Schnee) versagen Räder eher aufgrund der niedrigen Reibung. Dafür ist allerdings die Konstruktion eines Radantriebs extrem einfach, auch das Lenken der Bewegung ist sehr simpel. Zudem ist es für Roboter einfacher, sich in der Umwelt des Menschen zu bewegen, wenn sie einen ähnlichen Bewegungsapparat benutzen; bei Treppen wird dies sofort ersichtlich.

Dies führt nun zur Entwicklung von Laufrobotern, die sich durch ihren Antrieb besser in die Umgebung des Menschen integrieren können, als es radgetriebene Maschinen vermögen. Der Trend geht stärker in Richtung Service-Roboter, die dem Menschen einfache Aufgaben abnehmen. Dies wird in Zukunft von immer wichtigerer Bedeutung werden, so dass die Entwicklung von laufenden Robotern eine notwendige Aufgabe ist. So erklärt sich das Ziel dieser Arbeit, einem zweibeinigen Roboter ein möglichst schnelles Laufen zu ermöglichen. Die primäre Aufgabe des hier behandelten Roboters ist die Teilnahme am Robocup, dort ist eine

schnelle, aber auch stabile Laufbewegung von großem Vorteil. So wird hier nicht nur eine generelle Laufbewegung für einen hunmanoiden Roboter entwickelt, sondern diese auch für das spezielle Modell „Kondo KHR-1“ gezielt optimiert.

1.2 Aufgabenstellung

Ziel dieser Diplomarbeit ist es, dem Kondo KHR-1 ein eigenständiges zweibeiniges Laufen zu ermöglichen, um am Robocup [Rob06], der internationalen Weltmeisterschaft im Roboterfußball in der Liga für humanoide Roboter [Hum06], teilzunehmen. In dieser Diplomarbeit sollte ein möglichst dynamisches Laufen entwickelt werden, weil damit im Allgemeinen höhere Laufgeschwindigkeiten erreicht werden können.

Das eigenständige, autonome Laufen sollte über eine Modellierung einer parametrisierbaren Laufbewegung ermöglicht werden, die Parameter dieser Laufbewegung sollten mittels eines evolutionären Ansatzes vom Roboter selbst erlernt werden. Evolutionäre Ansätze (wie z.B. Genetisches Programmieren [BNKF02], Evolutionsstrategien [Sch95], Genetische Algorithmen [Gol89]) bieten sich für diese Art der Optimierung an, da es hier um eine nicht analytisch zu lösenden Problemstellung mit vielen Freiheitsgraden handelt. Diese Evolution des Laufens sollte gezielt für den Typ Kondo KHR-1 durchgeführt werden.

Um die Evolution einer Laufart zu ermöglichen, sollte eine passende Fitnessfunktion entwickelt werden, anhand derer die Güte des Laufmusters beurteilt werden konnte. Es sollte ein passender evolutionärer Ansatz ausgewählt, angepasst und implementiert werden, bestimmte Vorgaben (z.B. grundsätzliche Eigenschaften von Laufarten, wie Symmetrie) sollten als Modellannahmen in den Ansatz eingearbeitet werden. Die Software mit den oben genannten Aufgaben sollte dabei in einem Framework auf dem mit dem Roboter verbundenen PocketPC implementiert werden.

Um dem Roboter zu ermöglichen, eine Laufbewegung auf diese Weise zu erlernen, musste ein geeigneter Versuchsaufbau (wie etwa bei Nordin [WN02]) geschaffen werden, in dem der Roboter die verschiedenen Bewegungen validieren kann, ohne dabei beschädigt zu werden. Zudem sollte der Versuchsaufbau bei der Bewertung der Bewegungen helfen, indem mittels weiterer externer Sensoren (z.B. Kameras, Lichtschranken, u.ä.) zusätzliche Daten für die Lauf-Bewertungsfunktion des Roboters gesammelt werden. Schlussendlich sollte die Evolution des Laufens mit dem Roboter im Versuchsaufbau durchgeführt werden, wobei ein für diese Problemstellung angemessenes Optimierverfahren benutzt werden sollte.

1.3 Struktur der Arbeit

Nach der Einleitung und Übersicht in diesem Kapitel folgt in Kapitel 2 eine Vorstellung des hier benutzten Roboters vom Typ *Kondo KHR-1*. Es werden die Eigenschaften des Roboters sowie die daran für diese Arbeit durchgeführten Modifikationen beschrieben. Ferner werden zwei Software-Frameworks präsentiert, mit denen sich der Kondo KHR-1 ansteuern lässt, eine Auswahl eines dieser beiden Frameworks für diese Arbeit wird begründet.

In Kapitel 3 werden verschiedene laufende Roboter vorgestellt und erläutert, welche Metho-

den benutzt wurden, um deren Laufsteuerungen zu realisieren. Zudem werden Kriterien zur Beurteilung von Laufbewegungen aufgezeigt. In Kapitel 4 wird dann das für diese Arbeit entwickelte Laufmuster, die „WalkingEngine“, detailliert erläutert. Es werden die parametrisierbaren Trajektorien beschrieben, anhand deren die Roboter-Teile bewegt werden, zudem werden alle Parameter der WalkingEngine definiert. Die für die Ansteuerung des Roboters notwendigen Berechnungen werden nachvollzogen, ferner wird die Einbettung der WalkingEngine in das benutzte Framework erklärt.

Nach einer Übersicht über Optimierverfahren wird in Kapitel 5 die Auswahl der hier benutzten Verfahren für die Optimierung der WalkingEngine-Parameter begründet. Im Anschluss werden die Ergebnisse von Überprüfungen dieser Verfahren anhand von Testfunktionen präsentiert. In Kapitel 6 wird beschrieben, wie die Optimierung der Parameter praktisch durchgeführt wurde, es wird erläutert, welche Vorgehensweise bei der Optimierung sowohl im physikalischen Simulator als auch in einem Laufgestell mit den realen Roboter gewählt wurde.

Schließlich werden in Kapitel 7 die Ergebnisse der Optimierungen vorgestellt. Dabei werden zunächst die Messfehler sowohl für den benutzten physikalischen Simulator, als auch für das Laufgestell untersucht. Im Anschluss wird anhand von Simulations-Ergebnissen das Optimierverfahren ausgewählt, welches für die endgültige Optimierung der Laufparameter mit dem realen Roboter benutzt werden sollte. Nach einer Untersuchung der optimierten Laufparameter für verschiedene Laufrichtungen folgen die Ergebnisse der Parameter-Optimierung mit dem realen Roboter im Laufgestell.

Zum Schluss wird in Kapitel 8 eine Zusammenfassung dieser Arbeit gegeben, ferner werden noch offene Fragen aufgezeigt, sowie Anregungen für weitere Untersuchungen gegeben.

2 Der Roboter

In diesem Kapitel wird der in dieser Arbeit verwendete humanoide Roboter „Kondo KHR-1“ vorgestellt. Dabei wird zunächst die Hardware des Roboters erläutert, die Freiheitsgrade dieses Roboter-Modells sowie die Art der Ansteuerung des Roboters. Nach der Hardware werden die beiden für diesen Roboter verfügbaren Software-Frameworks vorgestellt und in Hinblick auf die Verwendbarkeit in dieser Arbeit verglichen.

2.1 Hardware

2.1.1 Der Roboter Kondo KHR-1

Der für diese Arbeit benutzte Roboter ist der in Abbildung 2.1(a) dargestellte KHR-1 von der japanischen Firma Kondo, von dem am Institut für Roboterforschung der Universität Dortmund zwei Exemplare zur Verfügung stehen. Dieser Roboter ist der erste kommerzielle humanoide Roboter, der als Bausatz geliefert wird. Der Kondo KHR-1 besitzt 17 Freiheitsgrade, allesamt rotatorische Gelenke durch die Verwendung von folgenden Servomotoren:

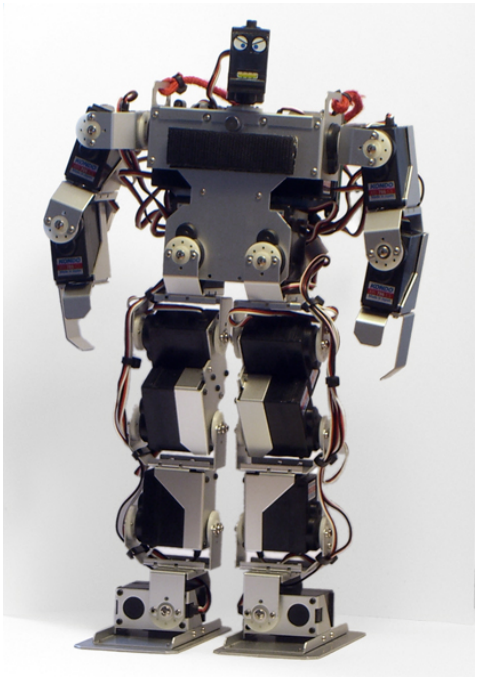
- einen Servo, um den Kopf zu drehen
- je 3 Servos in den Armen
- je 5 Servos in den Beinen

Eine Besonderheit stellt die Anordnung der Servos in den Beinen dar: es ist dem Roboter nicht möglich, die Beine zu drehen. Die Beinservos können die Füße des Roboters zwar entlang aller drei Raumachsen unterhalb des Oberkörpers bewegen; die Ausrichtung der Füße bleibt, bezogen auf den Oberkörper, jedoch stets gleich. So ist es z.B. nicht direkt durch die Ansteuerung eines dafür vorgesehenen Servos möglich, dass sich der Roboter dreht. Details werden in Abschnitt 4 erläutert.

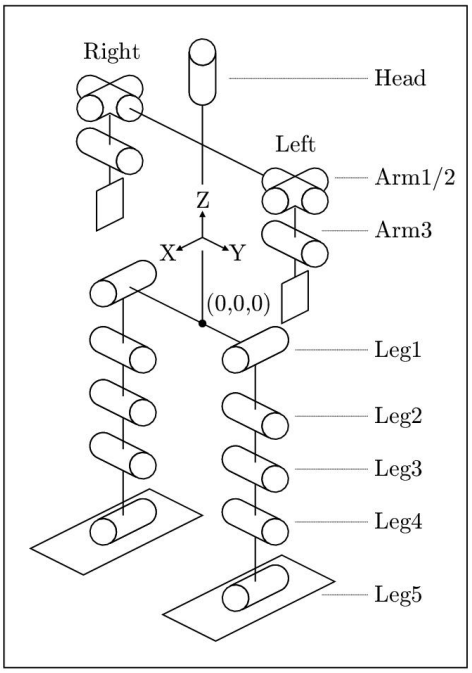
2.1.2 Modifikationen

Gegenüber dem handelsüblichen Kondo KHR-1 wurden für diese Arbeit Modifikationen an der Hardware vorgenommen. Die Ansteuerung der Roboter-Servos erfolgt über die von Kindler [Kin06] entwickelte Controller-Platine, die eine zuverlässige und robuste serielle Kommunikation zwischen Roboter und PC (auch PocketPCs sind möglich) ermöglicht und ferner einen Beschleunigungs-Sensor[1], sowie einen Rotations-Sensor (Gyroskop) trägt und ausliest.

[1] Der Beschleunigungs-Sensor misst die Beschleunigung in allen drei Raumachsen über eine kleine, bewegliche Masse im Sensor-Inneren und kann damit auch das Erdschwerefeld und somit die aktuelle Lage des Roboters erkennen.



(a) Der „Kondo KHR-1“ ohne den Pocket PC.



(b) Kinematische Struktur des Roboters mit eingezeichnetem Koordinatensystem und Achsenbezeichnungen.

Abbildung 2.1: Der Roboter Kondo *KHR-1*. Quelle: Kindler [Kin06].

Im Lieferzustand besitzt der Roboter Servos mit einem Getriebe aus vier Kunststoff-Zahnrädern. Die beiden Zahnräder mit dem geringsten Durchmesser haben dabei größte Torsionskraft auszuhalten, so dass die Räder schon nach einigen Stunden Betriebszeit oder bei Überbeanspruchung brechen. Der Hersteller bietet deshalb Austausch-Zahnräder aus Metall an, die zumindest diese beiden empfindlichsten Zahnräder ersetzen sollen. Für die beiden Kondo KHR-1 am IRF wurden diese Ersatzzahnräder für alle Servos mit Ausnahme des Kopfservos angeschafft, da sich nach einigen Versuchen mit der Ansteuerung des Roboters schon starke Verschleißerscheinungen zeigten. Durch die neuen Zahnräder konnte nicht nur die Lebensdauer der Servos stark verlängert werden, die Servos sind auch wesenstlich präziser; das „Spiel“ in der Bewegung ist deutlich geringer. Zudem sind zeitraubende Ausfälle des Roboters aufgrund beschädigter Servos wesentlich seltener.

Jedoch sinkt die Kraft der Servos mit den Metallzahnrädern deutlich. So können einige Bewegungen nicht mehr wie vorher ausgeführt werden, da die Servos nicht mehr die ursprünglichen Kräfte ausüben können. Dies betrifft vor allem die beiden obersten Servos in den Beinen des Roboters, welche für ein Auseinanderspreizen der Beine benutzt werden, da diese beim Laufen die gößten Kräfte aufnehmen. Da allerdings mit den Kunststoff-Zahnrädern eine zuverlässige Arbeit über längere Zeiträume nicht möglich ist, wird die Entwicklung und Optimierung der Laufbewegung mit den Metall-Zahnrädern durchgeführt. So hat das Optimierverfahren die Laufbewegung auf diese Art der Servo-Motoren anzupassen.

2.1.3 Ansteuerung des Roboters

Die in Abbildung 2.2 gezeigte Controller-Platine des Roboters ermöglicht es, diesen per serieller Schnittstelle über einen PC oder auch einen Pocket PC anzusteuern. Ferner sendet die Platine Daten von den auf ihr montierten Beschleunigungs- und Rotations-Sensoren an den angeschlossenen Rechner. Zudem lassen sich noch weitere Servo-Motoren und auch andere Sensoren an die Platine anschließen. Diese Platine ist auf dem Rücken des Roboters untergebracht und wird über ein serielles Kabel entweder mit einem stationären PC (z.B. zum Testen) oder mit einem PocketPC (an der Brust des Roboters befestigt zum autonomen Betrieb) verbunden.

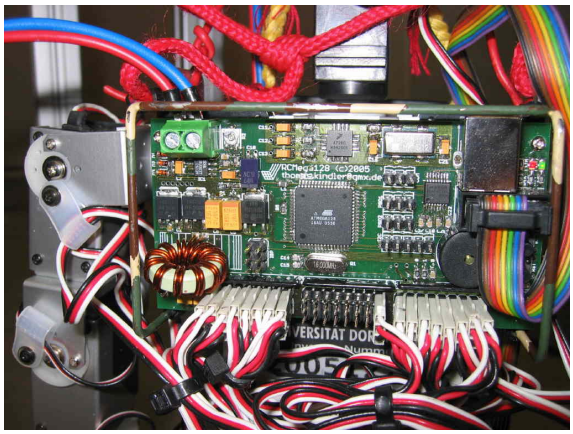


Abbildung 2.2: Die Controller-Platine auf dem Rücken des KHR-1. In der Mitte ist der Microcontroller zur Steuerung zu erkennen, unten sind die Anschlüsse für die Servo-Motoren, oben links der Anschluss für die Stromversorgung sowie rechts das Flachbandkabel für die serielle Ansteuerung.

Die Servo-Motoren des Kondo KHR-1 ermöglichen Drehungen von ca. 180°. Der gewünschte Winkel wird üblicherweise durch ein eigenes Protokoll über eine Datenleitung zum Servo übertragen und mittels eines an den Motor angebrachten Potentiometers eingelesen. Dazu enthalten die Servos neben dem Gleichstrom-Motor, dem Getriebe und dem Potentiometer auch eine Steuerungs-Elektronik. Durch Vergleich von Ist- und Soll-Position kann der Servo nun den angeforderten Winkel anfahren und auch halten. Beim KHR-1 kann nun die Soll-Position des Servos etwa alle 10ms neu ansteuern, was eine maximale Update-Rate der Servo-Winkel von 100 Hz ergibt. Praktische Tests von Kindler [Kin06] haben jedoch ergeben, dass eine Update-Rate von 50 Hz ausreicht[2], um flüssige Bewegungen zu relisieren, eine Erhöhung der Rate brachte keinen Vorteil.

Die Steuerung des Roboters erfolgt nun dergestalt, dass im zeitlichen Abstand von maximal 100 ms per serieller Schnittstelle die Soll-Werte für alle 17 Servomotoren an den Microcontroller der Controller-Platine gesandt werden. Die Kommunikation mit der Controllerplatine wird in einer Zwischenschicht des Software-Frameworks zur Robotersteuerung (siehe Abschnitt

[2] Allerdings erwarten die Servos mindestens alle 100 ms eine neue Soll-Position, was einer Frequnz von mindestens 10 Hz entspricht. Sinkt die Update-Rate unter diesen Wert, so schalten die Servos den Motor ab.

2.2) behandelt. Der Programmierer kann somit bequem über Schnittstellen die Servo-Winkel setzen, bzw. lesen oder die Sensordaten abfragen. Es muss jedoch sichergestellt werden, dass mindestens alls 100 ms die Servo-Werte gesetzt werden, wenn der Roboter seine aktuelle Position jeweils beibehalten soll. Ansonsten „sackt“ der Roboter zusammen, weil die Servos nach 100 ms ohne neue Werte abschalten. Sobald die nächsten Positionen gesendet werden, sind die Servos jedoch wieder aktiv.

2.1.4 Definition eines körpereigenen Koordinatensystems

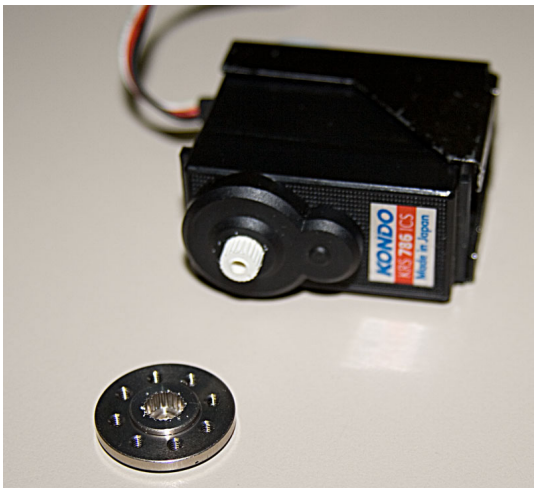
Um die Bewegungen des Roboters unmißverständlich ansteuern zu können, wird für diese Arbeit ein körpereigenes dreidimensionales Koordinatensystem für den Roboter definiert. Der Mittelpunkt dieses Koordinatensystems befindet sich dabei in der Mitte der „Hüfte“, im geometrischen Mittelpunkt zwischen den Achsen der obersten Bein-Servos. Die positive Z-Achse zeigt bei gerade stehendem Roboter senkrecht nach oben in Richtung Kopf, die positive X-Achse zeigt nach vorne (die Vorderseite des Roboters ist die Seite gegenüber der Controller-Platine). Die Positive Y-Achse zeigt in Bezug auf die Vorderseite des aufrecht stehenden Roboters nach links. In Abbildung 2.1(b) ist das Koordinatensystem eingezeichnet, die Pfeile deuten die positiven Achsenrichtungen an.

2.1.5 Kalibrierung der Servo-Motoren

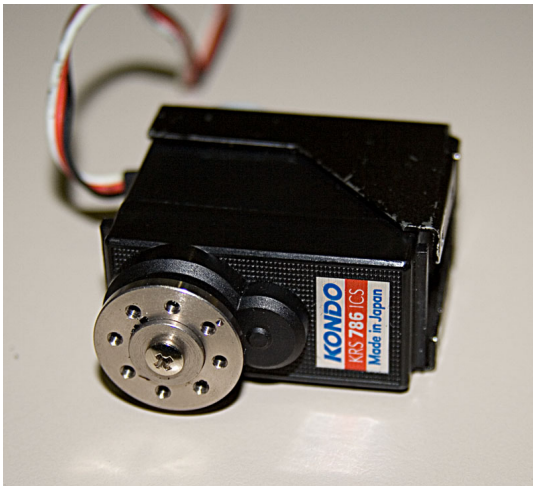
Servo-Motoren ermitteln ihre aktuelle Position über ein analoges Potentiometer, um die gemessene Potition mit der realen Position abzugleichen, muss jeder Servo kalibriert werden, um die Beziehung zwischen gemessener und tatsählicher Servo-Position zu ermitteln. Eine genaue Kalibrierung des Roboters ist von großer Bedeutung für ein gutes Funktionieren der Bewegungen, insbesondere des Laufens. Sind die Gelenke nicht genau kalibriert, kann es zu Unsymmetrien in den Bewegungen kommen. Dies führt dazu, dass ehemals gute Laufmuster schlecht oder im Extremfall gar nicht mehr funktionieren. Von daher ist eine korrekte Kalibrierung nach jeder Änderung an den Servos, wie Ausbau oder Austausch, unbedingt erforderlich.

Beim Zusammenbau des Roboters werden die nachfolgenden Gliedmaßen so an der Drehachse des Servos angebracht, dass der Servo sich in etwa in Mittelstellung befindet, wenn das Gelenk, in dem der Servo den Antrieb bildet, auch in Mittelstellung ist. Dies geht jedoch nur bis zu einer gewissen Genauigkeit, da die mit der Servo-Achse verbundene Antriebsscheibe nur 12 Aufnahme-Gewinde enthält. Bei doppelter Verschraubung entspricht dies 6 verschiedenen Positionen, jeweils um 60 Grad verdreht. Die Antriebsscheibe hat ein innenliegenden Zahnkranz, der auf dem Zahnrad der Servo-Antriebswelle aufgesetzt ist. Die Antriebsscheibe kann also jeweils nur um einen Zahn weitergedreht werden. Somit kann beim Zusammenbau keine vollkommene Genauigkeit hinsichtlich der Servo-Mittelstellung erfolgen. Ferner gibt es auch durch Fertigungstoleranzen Unterschiede in den Werten der Potentiometer. Die Abbildungen 2.3(a) und 2.3(b) zeigen einen der Kondo KHR1 Servos, mit abmontierter und mit angeschraubter Antriebsscheibe, in Abbildung 2.3(a) können das Zahnrad der Servo-Antriebswelle, sowie der inneliegende Zahnkranz der Antriebsscheibe erkannt werden.

Bei einer Kalibrierung wird die über den Widerstandswert des Potentiometers gemessene Position des Servos mit der tatsächlichen Position abgestimmt. Dies erfolgt dergestalt, dass die beiden Maximal-Ausschläge des Servos (linker/rechter Anschlag), sowie die Mittelstellung er-



(a) Der Servo mit abmontierter Antriebsscheibe. Hier ist der Kunststoff-Zahnkranz der Servo-Antriebswelle, sowie das Gegenstück in der Antriebsscheibe zu erkennen.



(b) Der Servo mit angeschraubter Antriebsscheibe, hier in der Variante aus Edelstahl.

Abbildung 2.3: Einer der Kondo KHR-1 Servos.

fasst und den Ist-Winkeln zugeordnet werden. Die so gemessenen Werte der Potentiometer werden für jeden Servo im Framework (siehe Abschnitt 2.2) gespeichert und für die Ansteuerung der Servos genutzt.

2.2 Software: verfügbare Frameworks

Für die Entwicklung eines Laufmusters und die anschließende Optimierung mit dem Roboter wird eine Software-Umgebung, ein *Framework* benötigt. Dieses Framework soll dem Entwickler helfen, den Code zu implementieren, zu debuggen und auf der Zielplattform zur Ausführung zu bringen. Es sollten für den hier benutzten Roboter z.B. Low-Level-Ansteuerungen, wie die Ansteuerung der Servo-Motoren und das Auslesen der Sensoren vom Framework erledigt werden, für häufig auftretende Aufgaben sollten Bibliotheken zur Verfügung stehen.

Für den Kondo-KHR1 stehen im IRF zwei Frameworks zur Verfügung. Dies sind einerseits *KondoControl* und das *BreDoBrothers-Framework*. Beide Frameworks haben unterschiedliche Schwerpunkte, so zeichnet sich KondoControl vor allem durch seine leichte Bedienbarkeit mittels grafischer Dialoge aus. Das BreDoBrothers-Framework dagegen ist konsolenbasiert, bietet aber neben seiner übersichtlichen modularen Struktur auch einen physikalischen Simulator. Im folgenden werden die Eigenschaften der beiden Frameworks vorgestellt und die Auswahl eines dieser beiden für diese Arbeit begründet.

2.2.1 KondoControl

Das Framework *KondoControl* [Kin06] stellt eine Umgebung für die im Fußballspiel beim Robocup benötigten Software-Module zur Verfügung. KondoControl wurde in der Programmiersprache C# für das *Microsoft .NET Framework* entwickelt; der Code für die Roboter-

Steuerung kann sowohl in C#, als auch in allen anderen vom Microsoft .NET Framework unterstützten Programmiersprachen (z.B. C++, Visual Basic, J#) implementiert werden. So steht eine breite Basis an Programmiersprachen zur Verfügung, vor allem C# zeichnet sich durch eine einfache und sichere Erstellung von Code aus.

Der KondoControl-Programmcode kann dank des .NET-Frameworks sowohl auf dem PC ausgeführt werden, wobei dann der Roboter per seriellem Kabel mit dem PC in Verbindung steht, als auch auf einem PocketPC, der am Roboter befestigt und mit ihm per seriellem Kabel verbunden ist. Kondo-Control kann die Bewegungen des Roboters optisch in einer 3D-Umgebung darstellen, wobei wahlweise sowohl die Soll-Positionen der Roboter-Gelenke, als auch deren Ist-Positionen angezeigt werden. Auch kann die Roboter-Steuerung rein in der Visualisierung dargestellt werden, ohne dass ein Roboter mit dem PC verbunden ist. Die Darstellung bezieht sich jedoch rein optisch auf die Gelenk-Positionen und berücksichtigt nicht physikalische Gegebenheiten wie z.B. Kollisionen von Roboterteilen, Schwerkraft, resultierende Sensor-Daten, usw.. Von daher kann die Optimierung der Laufbewegung mittles KondoControl nur mit dem realen Roboter durchgeführt werden, eine physikalische Simulation kann nicht durchgeführt werden.

KondoControl bietet weiter die Möglichkeit, mit verschiedenen einblendbaren Dialog-Fenstern verschiedene Aufgaben in Bezug auf den Roboter auszuführen. Hierzu gehören z.B. Lesen der Sensor-Daten, Kalibrierung der Servo-Motoren (siehe Abschnitt 2.1.5), Editieren von Bewegungsabläufen (z.B. Aufstehen nach einem Umfallen, Ball schießen, stehen, hinlegen), Updaten der Controller-Platinen-Firmware und vieles mehr.

2.2.2 BreDoBrothers-Framework

Das Framework des Teams „BreDoBrothers“ ist eine Weiterentwicklung des „German Team“-Frameworks [GTRLW+05], welches in der „Sony four legged league“ des Robocup für die Aibo-Roboterhunde benutzt wurde. In das BreDoBrothers-Framework wurde ferner ein physikalischer Simulator auf der Basis des Open Source-Projektes ODE („Open Dynamics Engine“ [ODE06]) integriert (siehe Abbildung 2.5). Damit kann der Roboter physikalisch simuliert werden, seine Bewegungen, die Interaktion mit der Umwelt sowie die Verarbeitung von Eingabedaten (über Sensoren) kann getestet werden, ohne dass ein echter Roboter benötigt wird.

Das Framework ist sehr gut strukturiert aufgebaut, alle elementaren Bestandteile, die für die Steuerung des Roboters benötigt werden, werden als Module, so genannte „Solutions“ implementiert und können sogar während der Laufzeit ausgetauscht werden. Diese Module kümmern sich z.B. um die einzelnen Gesichtspunkte der Bilderkennung (Tor-, Linien-, Mitspieler-, Ball-Erkenner), um die Bewegungen (Laufen, Ball schießen, Bewegung des Kopfes für die Bildaufnahme), das Verhalten der einzelnen Spieler, die Lokalisierung von Spielern und Bällen, Aufnahme und Verarbeitung von Sensor-Daten und weiteren. Eine schematische Übersicht über die Organisation und Verknüpfung der Module gibt Abbildung 2.6.

Das BreDoBrothers-Framework bietet die Möglichkeit, sowohl auf einem PC als auch auf einem Windows CE basierten PocketPC betrieben zu werden. Für beide Versionen werden dieselben Module benutzt, so dass es hier keine Konflikte gibt. In der PocketPC-Version kann

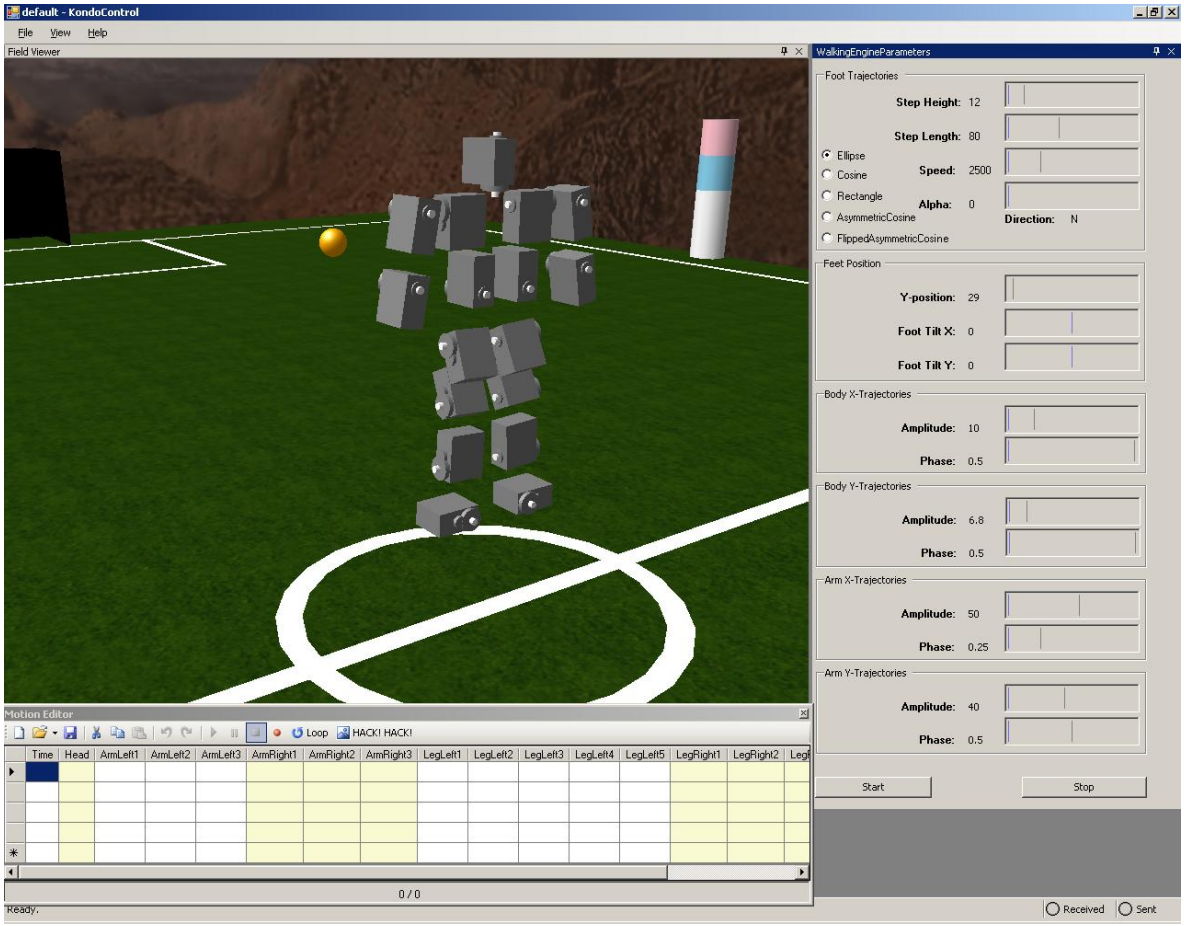


Abbildung 2.4: Screenshot des Frameworks *KondoControl*. Oben links sieht man das Fenster mit der grafischen Ausgabe der Roboter-Kinematik auf einem virtuellen Fußball-Spielfeld. Das Dialogfenster rechts ist für die Parametrisierung der in dieser Arbeit entwickelten Walking-Engine, das Fenster unten dient der Erstellung von Bewegungs-Abläufen (Motions).

der Roboter entweder autonom ein so genanntes Behavior ausführen, oder von der PC-Version des Frameworks per seriellem Kabel oder auch per WLAN ferngesteuert werden. Für das PC-Framework macht es von der Bedienung her keinen Unterschied, ob mit einem simulierten Roboter (womöglich auf einem simulierten Robocup-Spielfeld) oder mit einem realen Roboter (hier auch egal, ob per WLAN oder per Kabel angesteuert) gearbeitet wird; es kann leicht zwischen verschiedenen Ziel-Plattformen umgeschaltet werden. Wird der Roboter per PC-Version des Frameworks ferngesteuert, können alle Informationen, die dem PocketPC-Framework zur Verfügung stehen, von der PC-Version abgerufen werden, bis hin zu den aktuellen Kamera-Bildern und den Ausgaben der Bilderkennungsmodule. Bedient wird die PC-Version über eine Kommandozeile, mit der alle Aktionen aufgerufen werden können. Durch diese Austauschbarkeit der Plattformen und der Kommunikationswege ist es möglich, das Laufmuster für den Roboter sowohl am realen Roboter, als auch anhand seiner Simulation zu testen, ferner ist es möglich, den Roboter per Kabel oder per Funk anzusteuern.

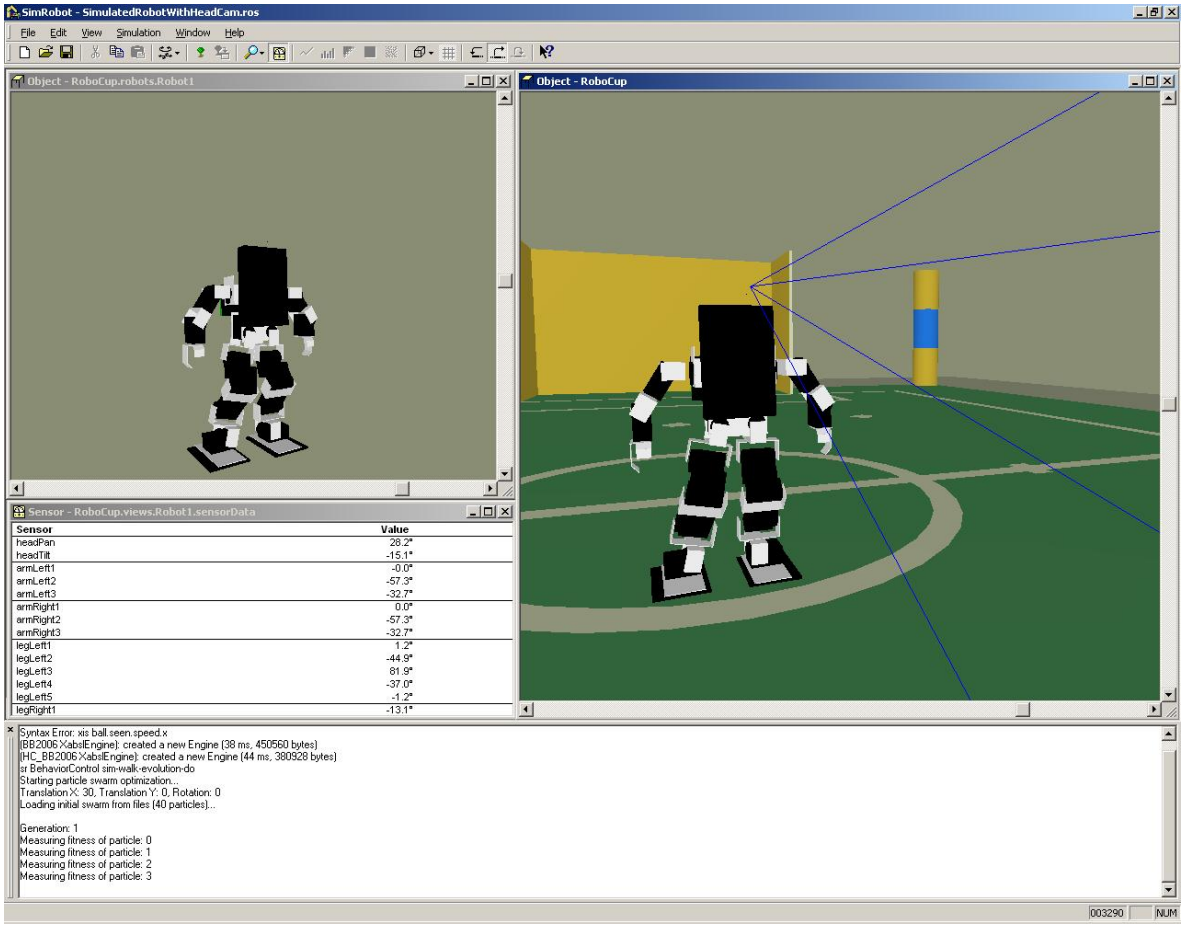


Abbildung 2.5: Screenshot des *BreDoBrothers*-Simulators. Links oben wird die Aktuelle Pose des Roboters grafisch angezeigt, rechts oben wird das simulierte Fußball-Spielfeld nebst Roboter dargestellt. Dabei sind auch Debug-Informationen, wie der Kamera-Blickwinkel eingezeichnet. In der Mitte links befindet sich das Ausgabe-Fenster mit den aktuellen Sensor-Daten und Gelenkwinkeln. Unten ist die Konsole zur Befehls-Eingabe und Debug-Ausgabe eingeblendet.

Der modulare Aufbau des BredoBrothers-Framework wird in Abbildung 2.6 dargestellt. Die Module gehören den Schichten *Perception*, *Object Modeling*, *Behavior Control* und *Motion Control* an. Diese Schichten entsprechen unterschiedlichen Abstraktionsebenen des Systems. Auf der *Perception*-Ebene werden die Sensor-Rohdaten bezüglich Gelenk-Positionen, Kamera-Bildern, Beschleunigungs-Sensoren, usw. vorverarbeitet. Daraus ergeben sich solche Daten wie Positionen der Extremitäten, Blickwinkel der Kamera, usw.

In der *Object Modeling*-Schicht werden die Perception-Daten anhand des Weltmodells weiterverarbeitet. Es wird das *World State* berechnet, daraus ergeben sich z.B. die eigene Position auf dem Spielfeld, die Position des Balles, des Tores sowie anderer Spieler. Anhand des gewonnenen Weltbildes kann nun in der *Behavior Control*-Schicht das Verhalten des Roboters definiert werden. Es wird dort entschieden, ob zum Ball gegangen wird, ob der Ball gespielt wird, ob die Position auf dem Spielfeld verändert werden soll, uvm. Schließlich werden in

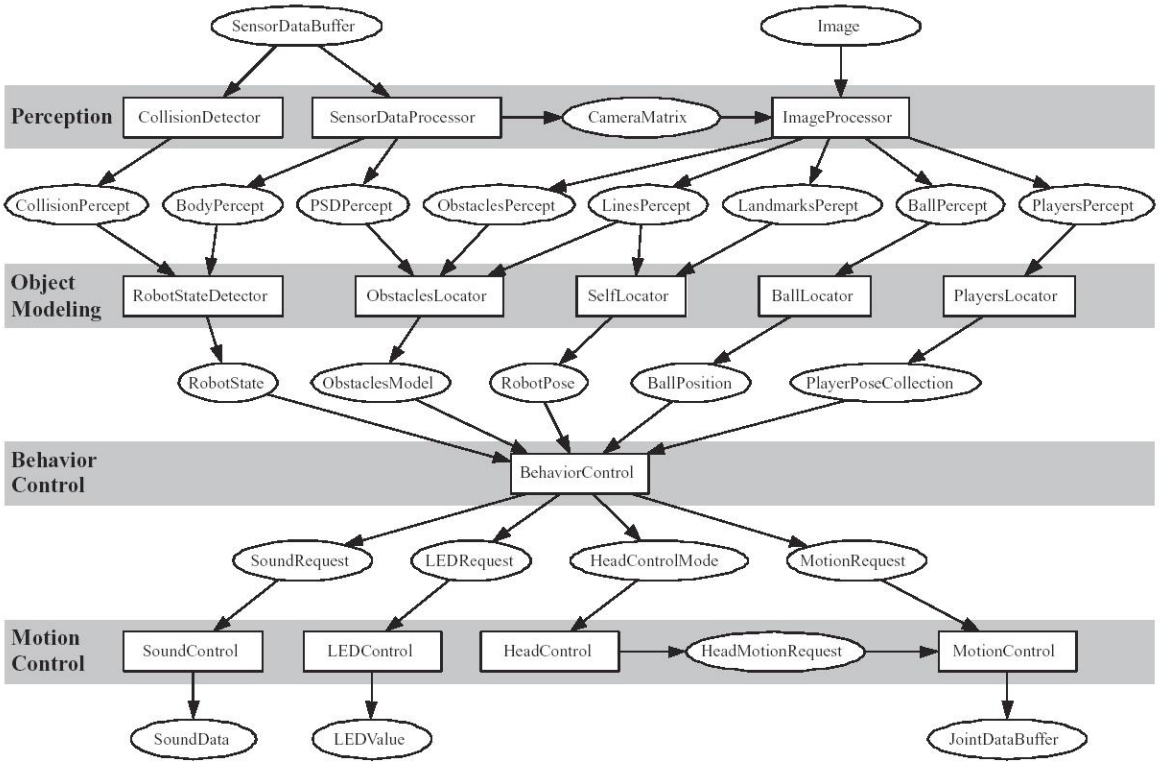


Abbildung 2.6: Übersicht über die Module des *BreDoBrothers*-Frameworks (Quelle: German Team [GTRLW+05]). Hier sind die verschiedenen konzeptionellen Verarbeitungs-Schichten, sowie den darin enthaltenen Modulen dargestellt.

der *Motion Control*-Schicht die Aktoren des Roboters angesteuert, um zu laufen, Bälle zu schießen, die Kamera zu bewegen, usw..

2.2.3 Auswahl eines Frameworks

Durch die Möglichkeit des BreDoBrothers-Framework, eine physikalische Simulation durchzuführen, aufgrund des modularen Aufbaus und der daraus resultierenden klaren Trennung einzelner Funktionsteile und aufgrund der Möglichkeit, das Verhalten des Roboters mittels Behaviors einfach beschreiben zu können, wurde für diese Arbeit dieses Framework ausgewählt. Das im Rahmen dieser Arbeit entwickelte Laufmuster (siehe Abschnitt 4) wurde zunächst aufgrund der einfacheren Entwicklung mit C# als „Proof of Concept“ für KondoControl implementiert und getestet. Allerdings musste unter anderem das Timing der Bewegungen sowie die Einbettung in das Framework vom Programmierer selbst durchgeführt werden, da KondoControl keine definierten Schnittstellen für diese Aufgaben bietet. Später wurde dann die WalkingEngine im performanteren C++ als Modul für das BreDoBrothers-Framework umgesetzt.

Die WalkingEngine im BreDoBrothers-Framework gehört der MotionControl-Schicht an, das Modul „DortmundWalkingEngine“ kann nun die Daten der Perception-Schicht vorverarbeitet entgegennehmen, eine direkte Ansteuerung der Sensoren muss nicht mehr erfolgen. Der

Aufruf der WalkingEngine wird von der MotionControl-Schicht durchgeführt, so muss sich der Entwickler nicht mehr um das Timing des Moduls kümmern. Ferner können auch die Servo-Motoren über eine High-Level-Schnittstelle angesprochen werden, was die Entwicklung sehr erleichtert. Die benutzten Optimierverfahren wurden als Teil der BehaviorControl-Schicht implementiert, was die Implementierung weiter vereinfachte, da die modulare Trennung es ermöglichte, die WalkingEngine von dem Optimierverfahren zu entkoppeln. Versuche mit den Optimier-Verfahren wurden schließlich mit dem Simulator des Framworks durchgeführt.

2.3 Implementierte Software-Module

Hier werden die im Rahmen dieser Arbeit für das BreDoBrothers-Framework entwickelten Module und Tools aufgezeigt. Es handelt sich dabei um das Modul „DortmundWalkingEngine“, welches die parametrisierbare Laufbewegung des Roboters erzeugt. Ferner wurden Verhaltens-Module erstellt, welche benutzt werden, um die Optimierung der WalkingEngine-Parameter durchzuführen. Außerdem wurde eine Bibliothek für die Berechnung inverser Kinematik geschaffen, sowie Tools, welche der Analyse der Optimierungs-Läufen dienen.

2.3.1 Das Modul „WalkingEngine“

Das in dieser Arbeit entwickelte Laufen wurde als ein Modul des Typs „WalkingEngine“ implementiert. Die WalkingEngine wird vom Framework alls 20ms aufgerufen und soll nur tätig werden, wenn ein „walk request“, eine Lauf-Anforderung vorliegt. Über „requests“ erfolgt ein Nachrichten-Versand an die jeweils beteiligten Module im Framework, worauf diese dann aktiv werden. Ein walk request betrifft die jeweils aktivierte WalkingEngine (von der immer nur eine aktiv sein kann), welche mit dem walk request mitgeteilt bekommt, mit welcher Geschwindigkeit (in mm/sec) in welche Richtung gelaufen werden soll. Die Laufrichtung beinhaltet ein Wertepaar für die translatorische Laufrichtung (x- und y-Koordinate) in Bezug auf die durch den Roboter-Körper festgelegten Raumachsen, sowie einen Wert für die rotatorische Bewegung (in Grad/sec). Die WalkingEngine sorgt dann selbst für den Bewegungsablauf des Laufens und stellt die Bewegungen so ein, dass die geforderten Geschwindigkeiten und Richtungen eingehalten werden. Sie beschleunigt den Roboter, bzw. bremst ihn bei Änderungen der walk request-Geschwindigkeiten ab und bestimmt den Zeitpunkt, zu dem das Laufen unterbrochen werden darf, ohne dass der Roboter umfällt (meist geregelt durch einen Kontakt beider Füße mit dem Boden). Eine Beschreibung der in dieser Arbeit konkret implementierten Moduls „DortmundWalkingEngine“ folgt in Abschnitt 4.6.

2.3.2 Die Verhaltens-Module

Um das Verhalten des Roboters im Fußballspiel definieren zu können, gibt es die Verhaltens-Module, genannt „Behavior“. In ihnen wird festgelegt, was der Roboter in Bezug auf welche Ereignisse zu tun hat. Logisch gesehen entspricht dies einer „finite state machine“. Diese Behaviors legen z.B. das Verhalten eines Torwartes oder eines Stürmers fest. Aber auch alle anderen Arten von Aktionen, die mit dem Roboter möglich sind, können hierdurch realisiert werden. Das Behavior bekommt über eine wohldefinierte Schnittstelle Zugang zu den anderen Modulen. So erhält das Behavior alle benötigten Daten, sowohl in Rohform von den Sensoren, als auch in verarbeiteter Form von den Cognition-Modulen (wie z.B. Position des Balles, Position des Spielers auf dem Feld, aktuelle Geschwindigkeit, usw.). Ferner kann das Behavior

den Roboter über die Motion-Module zu Aktionen bewegen (Laufen, Ball schießen, Aufstehen, etc.).

Um die Optimierung der Parameter der in dieser Arbeit entwickelten WalkingEngine unter Benutzung des BreDoBrothers Framework leicht zu ermöglichen, werden die Optimieralgorithmen sowie alle weiteren benötigten Programm-Konstrukte (Ansteuerung des Roboters, Fitnessmessung, Aufstehen nach einem Umfallen, etc.) als ein Behavior implementiert. So ist es leicht möglich, die Algorithmen sowohl im Simulator zu testen und zu untersuchen, als auch in Verbindung mit dem realen Roboter zu benutzen. Die Behavior-Modul sind entkoppelt von der benutzten WalkingEngine, so kann die Funktion der WalkingEngine vom Behavior über definierte Befehle angesprochen werden, ohne sich um deren konkrete Implementierung kümmern zu müssen.

3 Laufende Roboter

Dieses Kapitel gibt zunächst einen kurzen Überblick über das Laufen von Robotern und zeigt einige Arbeiten auf, die sich dieses Problems angenommen haben. Hier soll keineswegs eine vollständige Kollektion angestrebt, sondern lediglich ein Einblick in die verschiedenen Ansätze gegeben werden. Ferner wereden Kriterien zur Beurteilung von Laufbewegungen aufgezeigt und für diese Arbeit ausgewählt.

3.1 Roboter mit Beinen

Hier wird eine kurze Übersicht über die bisherigen Arbeiten zum Laufen von Robotern gegeben. Während die ersten Entwicklungen von Laufmustern für Roboter aufgrund der Einfachheit meist für Roboter mit sechs oder mehr Beinen galten, wurden in der Folgezeit vermehrt Laufmuster für Vierbeiner und für Zweibeiner entwickelt.

3.1.1 Einbeiner

Am *MIT Leg Laboratory* wurden mehrere einbeinige Roboter entwickelt, welche sich dadurch auszeichnen, dass sie sich hüpfend fortbewegen. Nach ersten Modellen, die an einem Arm befestigt waren, und sich so nur auf einer Linie bewegen konnten, stellte Raibert 1986 [Rai86] einen einbeinigen Roboter vor, der sich frei auf ebenem Boden hüpfend forbewegen konnte. Dieser bis zu $4,8$ Meilen pro Stunde schnelle „3D One-Leg Hopper“ ist in Abbildung 3.1 zu sehen. Jedoch ist offensichtlich, dass der praktische Nutzen von hüpfenden Robotern eher gering ist. Der Sinn dieser Entwicklungen liegt in der Erforschung der diesen Robotern eigenen Dynamik; die Ansteuerung der Einbeiner des MIT erfolgte über die physikalische Modellierung der Bewegungen und deren mathematische Lösung.

3.1.2 Zweibeiner

Nachdem bereits im Jahr 1969 durch Ichiro Kato von der Waesda Universität in Japan mit dem „WAP-1“ ein zweibeiniger Roboter entwickelt wurde, wurde zum Thema zweibeiniges Laufen besonders in den letzten zehn Jahren viel veröffentlicht. Nicht zuletzt der Robocup hat zu dieser Entwicklung beigetragen, da humanoide und somit zweibeinige Roboter seit zwei Jahren an dieser Fußballmeisterschaft teilnehmen. So ist vor allem das Team *Nimbro* [Nim06] sehr aktiv in diesem Bereich; es wurden vom Team Nimbro bisher acht Laufroboter benutzt, davon fünf selbstentwickelte und -gebaute. Abbildung 3.2(a) zeigt den bislang größten Roboter „Robotino“ des Nimbro-Teams, welcher auch beim Robocup 2006 in der TeenSize Klasse teilnahm. Die Laufsteuerungen des Freiburger Teams basieren auf einem zentralen Laufmustergenerator [BMS06]. Über diesen werden dann die Beingelenke periodisch ausgelenkt. Das Freiburger Team hat auch Evolutionäre Optimierverfahren zur Laufoptimierung eingesetzt, so wurde ein Genetischer Algorithmus eingesetzt, um die Laufgeschwindigkeit eines Kondo

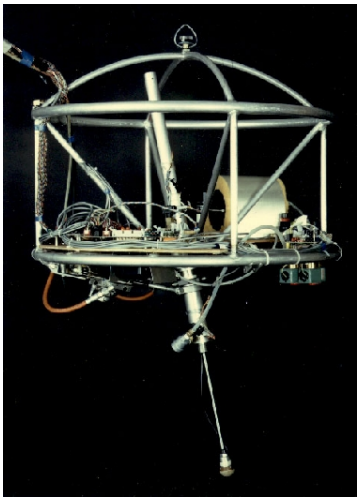
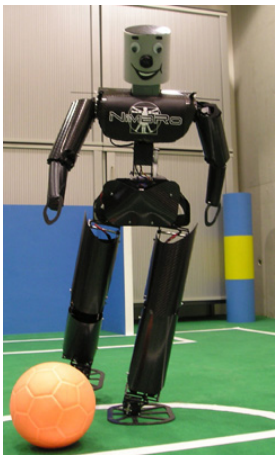


Abbildung 3.1: Der „3D One-Leg Hopper“ des MIT.

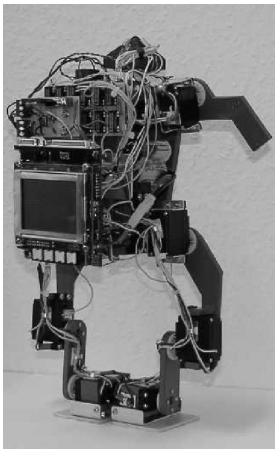
KHR-1 zu optimieren. Damit wurden Geschwindigkeiten von bis zu 14 cm/s erreicht.

Das Team *Darmstadt Dribblers* benutzt sechs verschiedene zweibeinige Roboter, darunter „Bruno“ (Abbildung 3.2(b)), einen 55 cm großen Humanoiden, welcher mit 40 cm/s der bislang schnellste zweibeinige Roboter des Robocup ist. Bei dem Darmstädter Roboter werden für mehrere Körperteile Trajektorien vorgegeben und für diese dann per inverser Kinematik die erforderlichen Gelenkwinkel errechnet. Die Stabilität des Laufens, der Energieverbrauch, sowie die Geschwindigkeit wurden mit verschiedenen Verfahren optimiert [FKK+05].

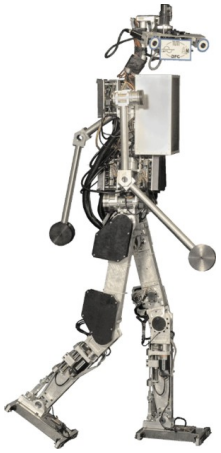


(a) Der Freiburger „Robotino“.

(b) Der Darmstädter „Bruno“.



(c) Der Dortmunder „Zorc“.



(d) Der Münchener „Johnnie“.

Abbildung 3.2: Die Humanoiden Roboter „Robotino“, „Bruno“, „Zorc“ und „Johnnie“ des Freiburger „Nimbro“-Teams, der „Darmstadt Dribblers“, des Lehrstuhls „Systemanalyse“ der Universität Dortmund und der Technischen Universität München.

An der Technischen Universität München wurde der seinerzeit weltweit schnellste humanoide

Roboter „Johnnie“ (siehe Abbildung 3.2(d)) entwickelt, welcher bei einer Größe von 180 cm und einem Gewicht von 49 kg eine Laufgeschwindigkeit von 2,4 km/h erreichen kann. Dabei erfolgt die Laufsteuerung durch vorgegebene Trajektorien für die verschiedenen Phasen der Laufbewegung anhand von Vorgaben für dynamische Randbedingungen des Systems. Um Störungen des Gleichgewichtes zu vermeiden, werden die Trajektorien zur Laufzeit über die Berechnung eines dynamischen Modells der Maschine adaptiert. So können auch unebene Untergründe belaufen werden, es handelt sich nicht um ein rein statisches Laufmuster, sondern um eine hochgradig dynamisch angepasste Laufbewegung.

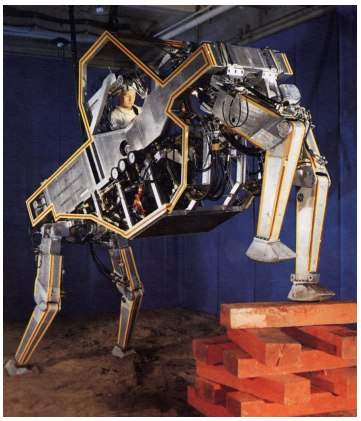
Einen neuronalen Controller nutzten Wischmann et al. [WP04] vom Fraunhofer Institut für autonome intelligente Systeme in Sankt Augustin. Ihr Ziel war es, ein zweibeiniges Laufen zu entwickeln, welches energieminimal ist und es dem Roboter ermöglicht, auf einem schrägen Untergrund sich alleine durch die Kraft der Gravitation zu bewegen. Dabei handelt es sich um ein so genanntes „passives Laufen“. Der neuronale Controller wurde mittels eines Evolutionären Algorithmus trainiert, weshalb Wischmann und Pasemann auf die mathematische Lösung der Gleichungen für die Ansteuerung verzichten konnten. Die Entwicklung des passiven Laufmusters erfolgte in einem Simulator, basierend auf der „Open Dynamics Engine“ [ODE06], auf der auch der BreDoBrothers-Simulator (Abschnitt 2.2.2) basiert.

Ziegler hat in seiner Dissertation [Zie03] einen anderen Weg als die bisher beschriebenen benutzt: er setzte nicht die Vorgabe der Trajektorien für Körperteile ein, sondern er benutzte *Genetische Programmierung* [Koz92], um ein Programm zur Steuerung der zum Laufen benötigten Bewegungen zu entwickeln. Dies wurde für verschiedenste Arten von Robotern benutzt, welche größtenteils simuliert wurden. Jedoch wurde Genetische Programmierung auch für den humanoiden Roboter „Zorc“ (Abbildung 3.2(c)) erfolgreich benutzt. Wolff und Nordin [WN02] [WN03] verfolgten den gleichen Ansatz, auch zum Teil in Zusammenarbeit mit Ziegler. Dabei benutzten sie den Roboter „Elvina“, der im Aufbau dem „Zorc“ sehr ähnlich ist.

Die heutige Robotik ist allerdings aufgrund der technischen Schwierigkeiten noch weit davon entfernt, das von der Natur entwickelte Laufen vergleichbar nachzubilden. Dies liegt nicht nur an der komplexen und schwierig zu steuernden Bewegung und der schwer beherrschbaren Dynamik, die dem Laufen eigen ist, sondern auch in hohem Maße an der momentan verfügbaren Hardware-Technik. Die meisten Roboter, so auch der Kondo KHR-1, die bisher gebaut wurden, benutzen Motoren, die nur starre und kaum flexible Bewegungen ermöglichen. Das Laufen von Tieren und Menschen jedoch geschieht mit dehnbaren Muskeln, welche durch ihre federartige Funktion weiche Bewegungen und die Rückgewinnung von Schwungenergie erst ermöglichen. Die Entwicklung von künstlichen Muskeln, die diesen Nachteil von Motoren nicht aufweisen, ist momentan noch im Anfangsstatium, so ist der Roboter *Lara* [Lar06] der Technischen Universität Darmstadt bisher der weltweit einzige Roboter mit künstlichen Muskeln. Für die 130 cm große Lara wird das Memory-Metall „NiTinol“ verwendet, dessen Drähte beim Anlegen einer Spannung eine vorher eingeprägte Form annehmen. Dadurch können die Draht-Bündel, die als Muskeln verwendet werden, kontrahieren und den Roboter auf diese Weise bewegen.

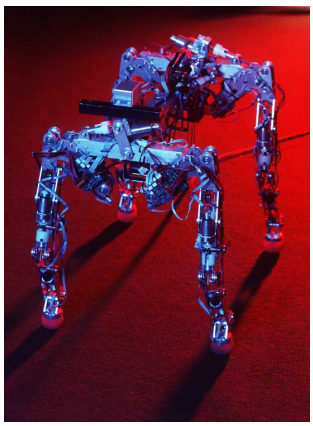
3.1.3 Vierbeiner

Bereits 1968 wurde ein vierbeiniger Laufroboter entwickelt, allerdings noch nicht autonom, sondern vom Menschen gesteuert. Ralph Mosher [LM68] entwickelte bei General Electric den „WalkingTruck“, welcher in Abbildung 3.3(a) zu sehen ist. Dieser hydraulisch angetriebene Roboter war über 3 m hoch, wog über 1,3 Tonnen und konnte eine Geschwindigkeit von bis zu 5 Meilen pro Stunde erreichen. Gesteuert wurde er von einem Menschen in seinem Inneren, der dazu beide Hände und beide Füße benötigte. Allerdings war die Steuerung sehr kompliziert, der Roboter war mit seiner Größe und seinem Gewicht eher ein Forschungsobjekt.



(a) Der „Walking Truck“ von General Electric aus dem Jahr 1968.

(b) Der Roboterhund „Aibo“ (hier der aktuellste Txp „ESR-7“) von Sony.



(c) Der „Bisam“ der Universitäten Duisburg und Jena.

Abbildung 3.3: Die vierbeinigen Roboter „Walking Truck“, „Aibo“ und „Bisam“.

Sehr viele Veröffentlichungen über vierbeiniges Laufen behandeln den in Abbildung 3.3(b) gezeigten Roboterhund „Aibo“ von Sony. Da dieser Roboter in der „Four Legged League“ beim Robocup seit 1998 eingesetzt wird und Geschwindigkeit ein entscheidender Faktor für Erfolg im Roboterfußball ist, beschäftigten sich Forscher aus der ganzen Welt mit dieser Roboter-Plattform. Das German Team [GTRLW+05] modellierte die Laufbewegungen des Aibo anhand von Trajektorien, welche über Parameter sehr detailliert einstellbar waren. Diese Parameter wurden dann mit Evolutionären Algorithmen optimiert. Momentan erreichen die Aibos mit diesem Ansatz Geschwindigkeiten von über 50 cm/s. Weitere Details sind in [HNF07] zu finden.

An den Universitäten Duisburg und Jena wurde die vierbeinige „säugetierartige Laufmaschine“ „Bisam“ (siehe Abbildung 3.3(c)) entwickelt. Die Laufsteuerung erfolgte nach dem Vorbild der Biologie: es wurden kleine Funktionseinheiten namens „Reflex“ erbaut, die in hierarchischen Struktutren miteinander vernetzt wurden. Diese „Reflexe“ wurden aus so genannten „Radial Basis Function“-Netzwerken, sowie aus Fuzzy-Controllern zusammengebaut. Diese wurden dann durch Ansätze des Reinforcement Learning optimiert. Es wurde also der Fokus auf die Nachbildung der Natur gelegt, es sollte eine möglichst gute Kopie des Laufapparates von Säugetieren geschaffen werden.

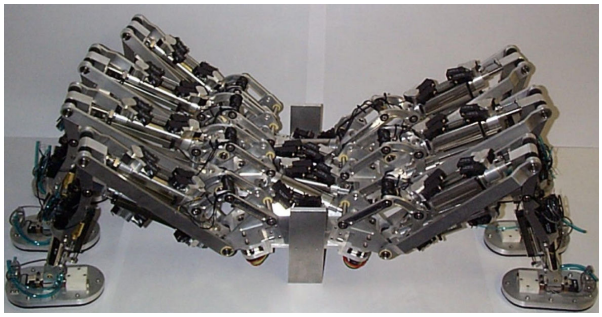
Die bisher beeindruckendste Arbeit im Bereich des vierbeinigen Laufens hat die amerika-

nische Firma „Boston Dynamics“ 2006 mit dem Roboterhund „Big Dog“ geleistet. Es handelt sich dabei um einen 1 m langen, 0,7 m hohen und 75 kg schweren Vierbeiner, der entwickelt wurde, Lasten bis zu 50 kg mit einer Geschwindigkeit von bis zu 3,3 Meilen pro Stunde zu tragen. Der Roboter wird über einen Verbrennungsmotor mit Energie versorgt und per Hydraulik bewegt. Ein eingebauter Computer steuert den Roboter, wobei eine solch dynamische Steuerung erreicht wird, dass selbst heftige Tritte gegen den Roboterkörper den „Big Dog“ nicht umstürzen lassen. Der Roboter wurde für das US-Amerikanische Militär entwickelt und kann sich selbst in unebenem Gelände, auf glatten und auf sehr weichen Untergründen, noch sicher bewegen. Details zur Art der Laufsteuerung wurden von Boston Dynamics bislang nicht bekanntgegeben.

3.1.4 Sechs- und Achtbeiner

Die Entwicklung von Laufbewegungen für Roboter mit sechs oder mehr Beinen ist aus einem Grund wesentlich leichter als für Roboter mit weniger Beinen: wenn mindestens sechs Beine vorhanden sind, kann der Roboter abwechselnd auf einer Hälfte der Beine stabil stehen, während die anderen Beine nach vorne bewegt werden. Da ein Roboter auf minimal drei Beinen stabil stehen kann, steht ein Sechs- oder Mehrbeiner bei abwechselndem Anheben der Hälfte seiner Beine immer auf mindestens drei Beinen, und ist nicht in der Gefahr, umzukippen. So handelt es sich bei dieser Art des Laufens stets um statische Stabilität (siehe Abschnitt 3.2.1).

(a) Der Sechsbeiner „Tarry 2“ der Universität Duisburg.



(b) Der Achtbeiner „Robobug IV“ der Universität Southampton.

Abbildung 3.4: Die sechs- und achtbeinigen Roboter.

Der sechsbeinige Roboter „Tarry“ (dargestelllt in Abbildung 3.4(a)) des Fachbereiches Mechanik der Universität Duisburg hat bei den Abmessungen 50 x 50 x 20 cm und einem Gewicht von knapp 3 kg eine maximale Nutzlast von 2900 g. Der Roboter wird mittels Neuronaler Netze angesteuert [FGLK98], es wird ein statisch stabiles Laufen erreicht. Die Beinbewegung erfolgt dergestalt, dass abwechselnd das jeweils mittlere Bein der einen und das vordere sowie hintere Bein der anderen Seite angehoben und nach vorne gesetzt wird. So steht der Roboter jederzeit auf den verbleibenden drei Beinen. Das Neuronale Netz wird trainiert durch eine Trainingsphase und ist zum Teil fähig, den Roboter unter geänderten Bedingungen, wie z.B. anderen Bodenverhälnissen, anzusteuern.

Der Achtbeiner „Robobug IV“ (zu sehen in Abbildung 3.4(b)) der Universität Southampton [LGC01] wird angesteuert über eine feste parametrisierbare Beinbewegung. Die Parametrisierungen beschreiben die Phasenlage bezüglich der anderen Beine, sowie das Verhältnis zwischen den Zeiten für die Bodenberührungs- und die Schwingphase für jedes einzelne Bein. Zur Optimierung der Laufgeschwindigkeit und der Laufstabilität wurde ein Genetischer Algorithmus benutzt.

3.2 Beurteilung einer Laufbewegung

Zur Beurteilung eines Laufmusters können verschiedene Kriterien benutzt werden. So werden unter anderem *statische* und *dynamische* Stabilität verwendet, es kann ferner der *Zero Moment Point* (ZMP) betrachtet werden. Die Laufgeschwindigkeit ist ein weiterer entscheidender Faktor, ebenso die „Glattheit“ des Gangs, also die Größe der Erschütterungen und die Harmonie der Bewegungsfolge. Im Folgenden werden diese Beurteilungsmaße erläutert und deren Relevanz für diese Arbeit beleuchtet.

3.2.1 Dynamische vs. Statische Stabilität

Beim Laufen wird zwischen zwei Arten von Stabilität unterschieden: *statische Stabilität* und *dynamische Stabilität*. Statische Stabilität herrscht, *„wenn sich die Projektion des Roboter-Schwerpunktes über seinem Aufstandspolygon befindet. Das Aufstandspolygon wird gebildet durch die konvexe Hülle der Fußpunkte der Beine“* [Zie03]. Bei Zweibeinern mit punktförmig anzunehmenden Füßen ist also statische Stabilität nicht möglich, da ja mit zwei Punkten kein Polygon aufgespannt werden kann. Beim Kondo KHR-1 existiert jedoch sehr wohl ein Polygon, da die Füße rechteckig sind. Befindet sich also die Projektion des Schwerpunktes jederzeit über dem Aufstandspolygon, welches von den Fußplatten des Kondo KHR-1 und dem Raum dazwischen gebildet wird, so ist der Roboter statisch stabil, er kann nicht umkippen. Dies führt jedoch zu einem allgemein langsamen und schwerfälligen Laufen, weshalb hier nicht gefordert wurde, dass das Laufen für den Kondo KHR-1 nun unbedingt statisch stabil sein soll.

Ziegler erläutert zur dynamischen Stabilität, dass beim dynamischen Gehen die Projektion des Schwerpunktes zeitweise außerhalb des Aufstandspolygons liegt. Der Roboter wird dann beginnen zu kippen, was beim dynamischen Laufen durch eine Vorwärtsbewegung eines Beines abgefangen werden muss. Um diese dynamische Balance zu sichern, kann z.B. wie bei Albert [Alb01] der so genannte *Zero Moment Point* (ZMP) berechnet werden. Der ZMP ist der Punkt am Untergrund, an welchem sich die Gravitation und die durch die Trägheit erzeugten Momente die Waage halten. Wenn die Laufsteuerung sicherstellt, dass der ZMP jederzeit innerhalb des Aufstandspolygons liegt, so ist der Roboter dynamisch stabil. Da der ZMP jedoch nur schwer zu berechnen ist, wurde dieser aufwändige Weg hier nicht gewählt.

Bei dem in dieser Arbeit entwickelten Laufmuster wird nicht von vornherein ein nur stabiles oder nur dynamisches Laufen angestrebt. Vielmehr wird ein Laufmuster entwickelt, welches dem menschlichen Gang nachempfunden ist. Durch Optimierung dieses parametrisierbaren Musters soll versucht werden, die Laufgeschwindigkeit möglichst weit zu erhöhen. Die Problematik des dynamischen Laufens, dass die Laufbewegung nicht inmitten eines Schrittes gestoppt werden kann, soll hier dadurch umgangen werden, dass der Roboter zum Laufen langsam aus dem Stillstand mittels Schrittlängen-Vergrößerung beschleunigt und zum Stehenbleiben auch

wieder auf diese Weise abbremst. Ferner soll die Laufbewegung nur unterbrochen werden, wenn sich beide Füße des Roboters auf dem Boden befinden.

3.2.2 Laufgeschwindigkeit

Wie bereits in Abschnitt 1.2 beschrieben, sollte in dieser Arbeit ein möglichst schnelles Laufen für den Kondo KHR-1 entwickelt werden. Von daher liegt bei der Entwicklung auch der Fokus auf genau dieser Forderung. So wird für die Beurteilung der Laufparameter (in Abschnitt 6) ausschließlich die Laufgeschwindigkeit benutzt. Dabei ist offensichtlich, dass dies nicht zwangsweise der beste Weg ist, das Laufmuster zu beurteilen. Jedoch wird davon ausgegangen, dass dieses Kriterium genügt, da in Verbindung mit der Laufgeschwindigkeit betrachtet werden soll, ob der Roboter durch Benutzung der einzelnen Laufparameter-Sätze umfällt. So hat ein instabiles Laufen gar nicht erst die Chance, als gut beurteilt zu werden; bei der Geschwindigkeitsmessung werden Parametersätze, die zu Stürzen führen, behandelt, als wäre ihre Geschwindigkeit gleich Null. So soll auf diesem Wege das Problem der Stabilität automatisch gelöst werden.

3.2.3 Glattheit der Laufbewegung

Neben der Geschwindigkeit ist eine weitere sinnvolle Beurteilung die „Glattheit“ des Laufens, wobei sich dieser Begriff schwer fassen lässt. Für das menschliche Auge ist es leicht, einen „glatten“ Gang von einem „unruhigen“ zu unterscheiden, ohne sich über die exakte Definition dieser Begriffe im Klaren zu sein. Diese Definition wird hier gar nicht erst versucht, es soll bloß festgehalten werden, dass ein Laufmuster, welches zu geringen Seitenbeschleunigungen und zu einer geringen Erschütterung des Roboter-Körpers führt, grundsätzlich von Vorteil ist. So wird bei geringen Querbeschleunigungen weniger Energie verbraucht, bei geringeren Erschütterungen werden die Gelenke und Motoren weniger belastet und es kann auch generell von einem stabileren Laufen ausgegangen werden, da dort der Roboter weniger von seiner aufrechten Position abweicht, also seltener in die Gefahr kommt, umzufallen.

Die Glattheit wurde in dieser Arbeit nicht als Kriterium zur Laufbeurteilung benutzt, da das Haupt-Augenmerk bei der Laufgeschwindigkeit lag, und erst einmal die grundsätzliche Optimierung der Laufparameter mit diesem Qualitätsmerkmal durchgeführt werden sollte. Jedoch wäre für weitere Arbeiten eine Kombination aus Geschwindigkeit und Glattheit interessant zu untersuchen. Beim Kondo-KHR-1 könnten als Maß für die Glattheit der Laufbewegung z.B. die Messwerte der Beschleunigungs-Sensoren benutzt werden. So wäre es denkbar, die Laufgeschwindigkeit zu maximieren und gleichzeitig die Summe der Beträge der Beschleunigungs-Messwerte zu minimieren. Dies könnte über eine geeignete Kombination in einer Fitnessfunktion, oder auch über mehrkriterielle Optimierverfahren geschehen.

4 Die WalkingEngine für den Kondo KHR-1

In diesem Kapitel wird das in dieser Arbeit entwickelte Laufen beschrieben, das durch die so genannte *WalkingEngine*, ein Software-Modul des entsprechend benutzten Frameworks (siehe Abschnitt 2.2), gesteuert wird. Die WalkingEngine bewegt die Servo-Motoren des Roboters dergestalt, dass diese sich gemäß den Vorgaben der Steuerungs-Instanz im Framework bewegen. Dabei lehnt sich die Ansteuerung der Gelenke stark an das menschliche Laufen an. Durch die genaue Beschreibung der Einzelbewegungen und deren Parametrisierung wird ein Einblick in den komplexen Raum der einzustellenden Parameter dieser Laufbewegung gewonnen.

4.1 Trajektorien der Roboter-Gliedmaßen

Die Laufbewegung des Kondo KHR-1 entsteht, indem Teile des Körpers entlang festgelegter Bahnen, so genannter *Trajektorien* bewegt werden. Die Körperteile, die von der hier vorgestellten WalkingEngine angesteuert werden, sind:

- die Füße
- die Arme
- der Oberkörper

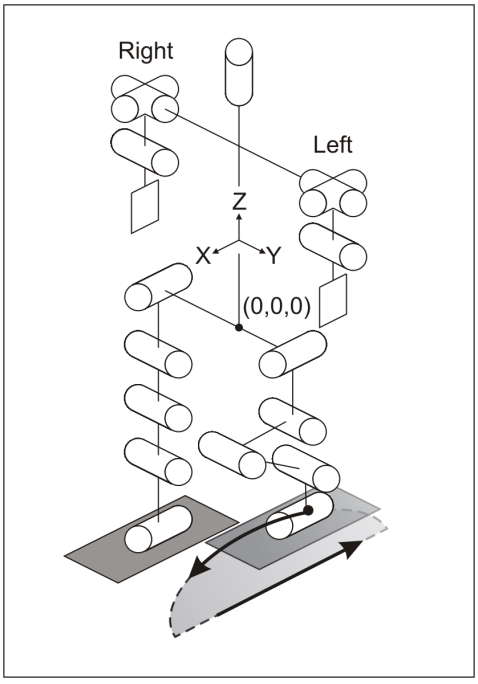
Die Füße werden entlang einer für zwei Dimensionen definierten Trajektorie bewegt, die Arme bewegen sich über jeweils ein eigenes Gelenk entlang eindimensionaler Bahnen. Ferner wird noch der Oberkörper eindimensional bewegt. Diese Bewegungen entsprechen im Groben der Laufbewegung des Menschen, die Benutzung von Trajektorien erfolgt, um klar definierte und leicht handhabbare Bewegungs-Vorgaben zu erhalten. Düffert [Düf04] zeigte, dass die Verwendung von Trajektorien als Beschreibung für Laufmuster zu sehr guten Ergebnissen führt.

Alle in der WalkingEngine benutzten Trajektorien werden in der Zeit, die für einen Vollschritt festgelegt wurde (Parameter *StepDuration*, siehe Abschnitt 4.5), einmal komplett abgefahren. Dabei gibt die Variable t mit $0 \leq t < StepDuration$ an, an welcher Position im Schritt sich die WalkingEngine gerade befindet.

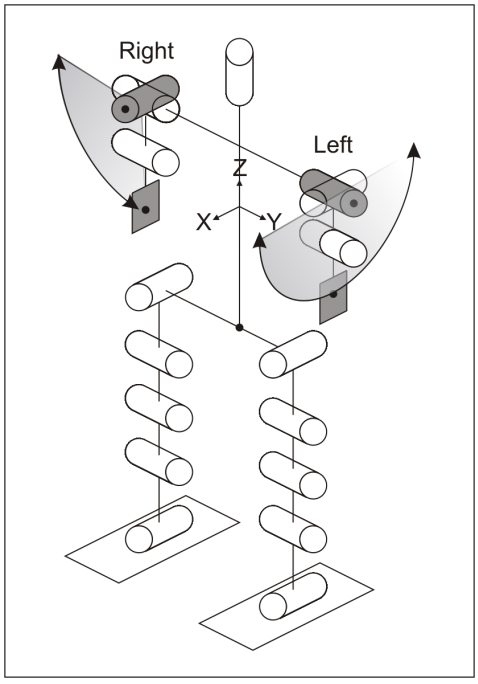
4.1.1 Füße

Alle in der Natur vorkommenden Laufmuster sind so gestaltet, dass für jedes Bein zwischen einer Phase der Bodenberührung und einer Phase, während derer das Bein durch ein Anheben in Laufrichtung vorgesetzt wird, unterschieden werden kann. Durch die Phase des Voransetzens wird das Bein, in Bezug auf den Körper, in die Laufrichtung bewegt und wieder abgesenkt. Während des Bodenkontaktes bewegt sich das Bein dann entgegen der Laufrichtung, um den Körper in die Laufrichtung zu drücken. Dabei ist die Bewegung des Fußes auf dem Boden

als Gerade aufzufassen, wenn man davon ausgeht, dass der Oberkörper parallel zum Boden bewegt werden soll, was im folgenden für die Laufbewegung des Roboters gelten soll. Die Bewegung des Fußes in der Luft ist jedoch fast beliebig wählbar. Auch die Bewegung der Gelenke ist in unterschiedlicher Weise möglich, so kommen in der Natur z.B. Kniegelenke vor, die nicht wie beim Menschen nach vorne einknicken, sondern nach hinten[1].



(a) Die Fuß-Trajektorie für einen Fuß, hier die Halb-Ellipse.



(b) Die Arm-Trajektorien in der X-Z- und in der Y-Z-Ebene.

Abbildung 4.1: Schematische Darstellung der Fuß- und Arm-Trajektorien.

Die Bewegung des einzelnen Fußes kann somit als Trajektorie im dreidimensionalen Raum aufgefasst werden. Dabei reicht es allerdings aus, sich auf eine zweidimensionale Form der Trajektorie, also eine zweidimensionale Ebene im dreidimensionalen Raum, zu beschränken. Die Trajektorie wird aufgespannt durch die Achsen aus der Laufrichtung und der Richtung entgegen der Schwerkraft. Die dritte Dimension würde keinen weiteren Vorteil für diese Art des Laufens bringen, weil die Füße bei einer Bewegung außerhalb der beschriebenen Achsenbereiche nur unnötige Wegstrecken zurücklegen würden, ohne z.B. die Schrittlänge in Laufrichtung oder die Schritthöhe zu vergrößern. Eine solche Trajektorie hat eine geschlossene Bahn darzustellen, die pro Schritt vom Roboterfuß einmal komplett abgefahren wird. Die Unterseite der Trajektorie soll dabei wegen des hier betrachteten ebenen Untergrundes eine Gerade sein, um den Fuß beim Vorandrücken des Körpers stets in Kontakt mit dem Boden zu halten, bzw. um den Oberkörper stets parallel zum Boden zu führen. Zudem werden die Fußsohlen stets parallel zum Boden ausgerichtet, so dass der Fuß beim Aufsetzen den Boden komplett berührt und

[1] Bei Vierbeinern sind meist beide Varianten anzutreffen: bei den Vorderbeinen knicken die Kniegelenke nach vorne ein, bei den Hinterbeinen nach hinten. Bei Zweibeinern kommen auch beide Varianten vor: beim Menschen und bei Primaten knicken die Knie nach vorne ein, bei Vögeln dagegen ist es umgekehrt.

während der Phase des Voransetzens nicht in die Gefahr kommt, durch einen abgewinkelten Fuß mit einer dessen Kanten den Boden zu berühren. Weiterhin muss bei der hier benutzten Ansteuerung der Trajektorien sichergestellt sein, dass die Zeit für das Abfahren des oberen Trajektorien-Abschnittes genauso lang ist, wie die Zeit für die Verfolgung des unteren, geradenförmigen Trajektorien-Segmentes. Dies ist notwendig, da sonst die Kopplung der beiden Trajektorien nicht mehr synchron wäre. Die Beine würden sich in dem Fall nicht abwechselnd bewegen, der Phasenversatz wäre ständig verschieden, was eine ungleichmäßige Schrittfolge bedeuten würde.

Daher werden die Füße nun von der WalkingEngine so an der Trajektorie entlanggeführt (siehe Abbildung 4.1(a)), dass stets der eine Fuß auf dem Boden steht und sich auf dem unteren Trajektorien-Segment nach hinten bewegt, während der andere Fuß ohne Bodenkontakt eine Bewegung in Laufrichtung ausführt. Dabei sind die beiden Trajektorien der Füße hinsichtlich ihres Timings um die Hälfte der Schrittdauer gegeneinander phasenverschoben, so dass der eine Fuß immer den unteren Teil seiner Trajektorie abfährt, während der andere Fuß den oberen Teil seiner Trajektorie verfolgt. Die entgegengesetzten Endpunkte der beiden Trajektorien werden von beiden Füßen gleichzeitig erreicht, dann befinden sich beide Füße für einen kurzen Moment gleichzeitig auf dem Boden. Bei der Entwicklung der WalkingEngine für den Kondo-KHR1 wurden verschiedene Arten von Trajektorien untersucht, dies waren *Halbkreis*, *Rechteck*, *Halb-Ellipse*, *Sinus* und *Spezial-Sinus*.

Um das Entlangfahren des Fußes an der Trajektorie umzusetzen, müssen die dafür benötigten Gelenkwinkel berechnet werden. Dies geschieht mittels *inverser Kinematik* (siehe Abschnitt 4.2), einem Verfahren, welches aus Soll-Positionen im Raum die dafür benötigten Gelenkwinkel errechnet. Da es für mehr als drei Gelenke nicht zwangsweise eine eindeutige Lösung dafür gibt (so können verschiedene Gelenkwinkel zur selben Raumposition des Endes der kinematischen Kette[2], hier des Fußes, führen), werden hier Annahmen getroffen, die zu eindeutigen Lösungen führen. Details hierzu sind im Abschnitt 4.2 zu finden.

4.1.1.1 Halb-Kreis

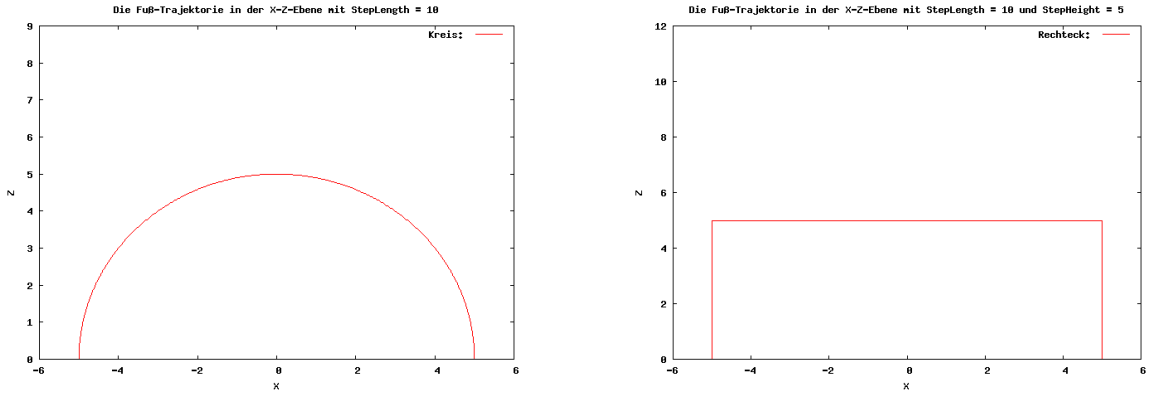
Die Halbkreis-Trajektorie (für eine beispielhafte Skizze siehe Abbildung 4.2(a)) entspricht folgender Vektor-Formel:

$$f(\vec{t}) = \begin{pmatrix} x(t) \\ z(t) \end{pmatrix} = \begin{cases} \begin{pmatrix} -\cos(2\pi t) \cdot \frac{StepLength}{2} \\ \sin(2\pi t) \cdot \frac{StepLength}{2} \end{pmatrix} & \text{für } 0 \leq t \leq 0.5 \\ \begin{pmatrix} \frac{StepLength}{2} - 2StepLength(t - 0.5) \\ 0 \end{pmatrix} & \text{für } 0.5 < t < 1 \end{cases}$$

Dabei ist, bedingt durch die Kreisform, die Schrittlänge ($StepLength$) immer gleich der doppelten Schritthöhe ($StepHeight$). Dies sorgt zwar für ein gewünschtes lotrechtes Aufsetzen der Füße auf dem Boden, führt allerdings bei langen Schritten zu einem starken Anheben der Füße. Dies bedingt eine starke Instabilität des Roboters, da die angehobenen Beine den Schwerpunkt weit nach oben verlagern und ein instabiles Laufen erzeugen. Dagegen würde eine geringe Schritthöhe auch zu einer geringen Schrittweite führen, was dann bei gleicher

[2] Für eine Definition der *Kinematischen Kette* siehe Abschnitt 4.2.1.

Schrittdauer in einer langsameren Laufgeschwindigkeit resultieren würde.



(a) Die Halbkreis-Fußtrajektorie.

(b) Die Rechteck-Fußtrajektorie.

Abbildung 4.2: Die Fußtrajektorien HALBKREIS und RECHTECK für Schrittweite 10 und Schritthöhe 5 in der X-Z-Ebene.

4.1.1.2 Rechteck

Die Rechteck-Form bietet zwar auch ein senkrechtes Absetzen der Füße, hat jedoch den gravierenden Nachteil, dass sich die Füße sehr ruckartig bewegen, was an den rechtwinkligen Ecken der Trajektorie liegt. In diesen Ecken werden die Füße in Sekundenbruchteilen grob in ihrer Richtung umgelenkt, was sich in Erschütterungen des Roboter-Körpers niederschlägt.

Die Rechteck-Trajektorie (für eine beispielhafte Skizze siehe Abbildung 4.2(b)) ist definiert durch:

$$f(\vec{t}) = \begin{pmatrix} x(t) \\ z(t) \end{pmatrix} = \begin{cases} LiftUp & \text{für } 0 \leq t \leq EndTimeLift \\ AirPhase & \text{für } EndTimeLift < t \leq EndTimeForeward \\ RaiseDown & \text{für } EndTimeForeward < t \leq 0.5 \\ GroundPhase & \text{für } 0.5 < t < 1 \end{cases}$$

Dabei sind:

$$
\begin{aligned}
LiftUp &= \begin{pmatrix} -\frac{StepLength}{2} \\ 2t \cdot TrajectoryLength \end{pmatrix} \\
AirPhase &= \begin{pmatrix} ((2t - StartTimeForeward) \cdot TrajectoryLength) - \frac{StepLength}{2} \\ StepHeight \end{pmatrix} \\
RaiseDown &= \begin{pmatrix} \frac{StepLength}{2} \\ 2t \cdot TrajectoryLength \end{pmatrix} \\
GroundPhase &= \begin{pmatrix} \frac{StepLength}{2} - 2StepLength(t - 0.5) \\ 0 \end{pmatrix}
\end{aligned}
$$

mit:

$$
\begin{aligned}
EndTimeLift &= StepHeight/TrajectoryLength \\
StartTimeForeward &= EndTimeLift \\
EndTimeForeward &= EndTimeLift + (StepLength/TrajectoryLength) \\
StartTimeLower &= EndTimeForeward
\end{aligned}
$$

4.1.1.3 Halb-Ellipse

Die Halbellipse (siehe Abbildung 4.3(a)) ist definiert durch:

$$
f(\vec{t}) = \begin{pmatrix} x(t) \\ z(t) \end{pmatrix} = \begin{cases} \begin{pmatrix} -\cos(2\pi t) \cdot \frac{StepLength}{2} \\ \sin(2\pi t) \cdot StepHeight \end{pmatrix} & \text{für } 0 \leq t \leq 0.5 \\ \begin{pmatrix} \frac{StepLength}{2} - 2StepLength(t - 0.5) \\ 0 \end{pmatrix} & \text{für } 0.5 < t < 1 \end{cases}
$$

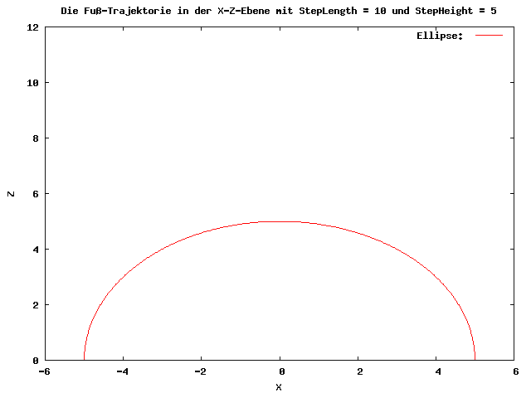
Diese Trajektorie ist als die empfehlenswerteste anzusehen, da die Schritthöhe entkoppelt von der Schrittweite ist und die Füße lotrecht auf den Boden aufsetzen. Ferner werden durch die stetige und abgerundete Bahn sanfte Bewegungen erzeugt.

4.1.1.4 Sinus

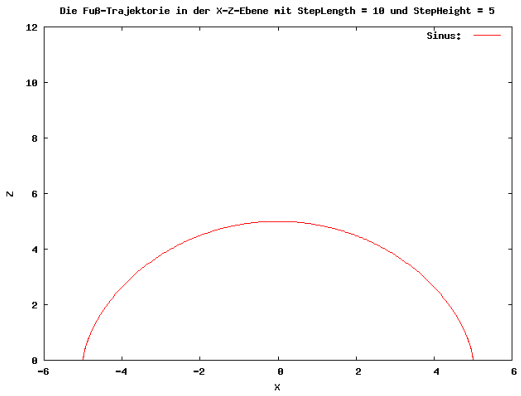
Die Sinus-Trajektorie (siehe Abbildung 4.3(b)) nach Albert [Alb01] ist definiert durch:

$$
f(\vec{t}) = \begin{pmatrix} x(t) \\ z(t) \end{pmatrix} = \begin{cases} \begin{pmatrix} \frac{StepLength}{2}\left(2 \cdot ((2t) - 1) - \frac{\sin(\pi(1+2\cdot(2t-1)))}{\pi}\right) \\ \sin(2\pi t) \cdot \frac{StepLength}{2} \end{pmatrix} & \text{für } 0 \leq t \leq 0.5 \\ \begin{pmatrix} \frac{StepLength}{2} - 2StepLength(t - 0.5) \\ 0 \end{pmatrix} & \text{für } 0.5 < t < 1 \end{cases}
$$

Sie ähnelt sehr stark der Halb-Ellipse, hat jedoch den Nachteil, dass sich die Füße beim Aufsetzen auf den Boden immer noch leicht in die Laufrichtung bewegen, was zu einem Stolpern auf dem Teppichboden führen kann.



(a) Die Halbellipsen-Fußtrajektorie.



(b) Die Sinus-Fußtrajektorie.

Abbildung 4.3: Die Fußtrajektorien HALBELLIPSE und SINUS für Schrittweite 10 und Schritthöhe 5 in der X-Z-Ebene.

4.1.1.5 Spezial-Sinus

Die Spezial-Sinus-Trajektorie (siehe Abbildung 4.4) stammt von Meredith et al. [MM04] und wurde für die Laufmuster-Erzeugung in animierten Trickfilmen entwickelt. Sie ist folgendermaßen definiert:

$$f(\vec{t}) = \begin{pmatrix} x(t) \\ z(t) \end{pmatrix} = \begin{cases} \begin{pmatrix} \pi t \cdot StepLength - \frac{StepLength}{2} \\ \begin{cases} \frac{StepHeight}{2}(1 - \cos(6\pi t)) & \text{für } t < \frac{1}{6} \\ \frac{StepHeight}{2}(1 - \cos(\frac{\pi(1+6t)}{2})) & \text{sonst} \end{cases} \end{pmatrix} & \text{für } 0 \leq t \leq 0.5 \\ \begin{pmatrix} \frac{StepLength}{2} - 2StepLength(t - 0.5) \\ 0 \end{pmatrix} & \text{für } 0.5 < t < 1 \end{cases}$$

Zwar wirkt diese Form der Beinbewegung in Animationen sehr natürlich, jedoch setzen auch hier die Füße nicht senkrecht auf den Boden auf. Zudem wird der Roboter-Körper ungleichmäßig ins Pendeln gebracht, da die beiden Trajektorien-Teile (Anheben und Absenken des Fußes) unterschiedlich schnell abgefahren werden.

4.1.1.6 Auswahl einer Fuß-Trajektorie

Es stellte sich bei Experimenten mit den Trajektorien heraus, dass die Halbellipse (siehe Abbildung 4.3(a)) am Vorteilhaftesten ist. Eine Rechteck-Form sorgt für zu ruckartige Bewegungen und starke Erschütterungen des Roboterkörpers, ein Halbkreis (Abbildung 4.2(a)) sorgt zwar für glatte Bewegungen, jedoch ist die Schritthöhe immer gleich der halben Schrittlänge. Die Sinus- und Spezial-Sinus-Trajektorien (Abbildungen 4.3(b) und 4.4) erzeugen zwar auch sehr glatte und auch natürlich aussehende Bewegungen, allerdings verursachen sie ein Stolpern des Roboters, weil die Füße kurz vor dem Aufsetzen auf dem Boden noch leicht in Laufrichtung bewegt werden. Deshalb wurde für die endgültige WalkingEngine die Halb-Ellipse als Fuß-Trajektorie ausgewählt und ausschließlich weiter verwendet. Bei dieser Art der Bewegung sind

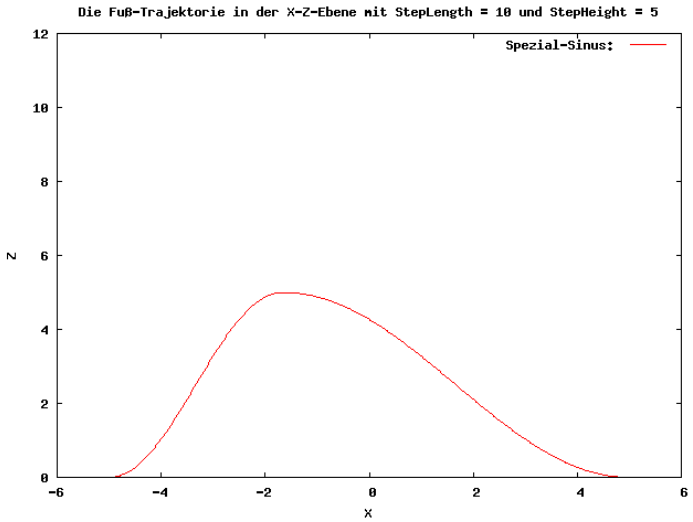


Abbildung 4.4: Die Spezial-Sinus-Fußtrajektorie für Schrittweite 10 und Schritthöhe 5 in der X-Z-Ebene.

Schritthöhe und Schrittlänge entkoppelt, die Füße setzen senkrecht auf den Boden auf, zudem handelt es sich um eine glatte Bewegung, bei welcher der Roboter-Körper relativ ruhig bleibt.

4.1.2 Arme

Die Arme werden, anders als die Füße, mit einer Trajektorie angesteuert, welche nur eindimensional auf ein Gelenk einwirkt und damit den Arm im Raum bewegt. Diese Trajektorie betrifft also nur ein einziges rotatorisches Gelenk, daher kann hier auf die Berechnung inverser Kinematik (siehe Abschnitt 4.2) verzichtet werden. Bei dem hier vorgestellten Laufmuster werden die Arme sowohl in der X-Z-Ebene als auch in der Y-Z-Ebene bewegt, einen Eindruck davon vermittelt Abbildung 4.1(b). Die Geschwindigkeit und die Phasenlage der Armbewegungen sind dabei fest an die Fußbewegungen gekoppelt. Beide Armbewegungen werden als Sinus-Schwingung durchgeführt, die Zeit für ein Vor- und wieder Zurückschwingen der Arme entspricht genau der Zeit für einen Schritt beider Beine. Es wird für diese Kopplung derselbe Parameter $t \in [0, 1]$ benutzt, wie in allen anderen Trajektorien auch.

Es wurde neben der Sinus-Bewegung für die beiden Armtrajektorien auch eine lineare Ansteuerung getestet, bei der der Winkel ohne die Umrechnung in einen Sinus direkt linear entlang des Parameters t angefahren wurde. Die Ergebnisse waren jedoch sehr unbefriedigend, da sich ein ähnliches Verhalten wie bei der Rechteck-Trajektorie (siehe Abschnitt 4.1.1.2) für die Fußbewegungen einstellte. Die Bewegungen sind bei linearer Ansteuerung im Wendepunkt, an dem der Arm wieder zurückbewegt wird, sehr ruckartig, so dass der gesamte Roboter-Körper davon erschüttert wird. Beim Sinus dagegen wird die Bewegung vor dem Umkehrpunkt abgebremst, die gesamte Bewegung ist sanfter und wirkt „natürlicher“. Es wird dadurch ein allgemein stabileres Laufverhalten gewährleistet. Zudem bleiben die Arme länger in den Bereichen der Wendepunkte, was dafür sorgt, dass die Arme länger die Funktion als Ausgleichsgewichte für die Beine erfüllen können.

Die Aufgabe der Armbewegung besteht darin, einen Gewichts-Ausgleich für das jeweils angehobene Bein zu erzeugen. Wenn z.B. das rechte Bein angehoben wird, verlagert sich der Schwerpunkt des Roboters nicht nur nach oben, sondern auch nach rechts. Dadurch, dass der linke Arm nun angehoben wird, wird der Schwerpunkt zurück nach links geführt. Dies wird weiterhin durch die Oberkörper-Bewegung (siehe Abschnitt 4.1.3) unterstützt.

4.1.2.1 X-Z-Ebene

Die Sinus-Trajektorie für die Arm-Bewegung in der X-Z-Ebene ist definiert als:

$$\omega_{Arm_x}(t) = \frac{\pi}{2} ArmAmplitudeX \cdot \sin\left(2\pi\left(t + ArmPhaseX\right)\right)$$

mit Phasenversatz $ArmPhaseX = 0.25$ konstant für diese Arm-Bewegung. Das bedeutet, dass der linke Arm nach vorne schwingt, wenn sich das rechte Bein nach vorne bewegt[3]. Mit diesem Winkel $\omega_{Arm_x}(t)$ wird das Arm-Gelenk 1 direkt angesteuert. Der Phasenversatz $ArmPhaseX$ ist hier fest, damit bei einem Schritt mit dem rechten Bein nach vorne der linke Arm entsprechend nach vorne ausgelenkt wird. Zwischen den beiden Arm-Gelenken gibt indes nochmals einen Phasen-Versatz von 0.5, so dass sich die Arme stets genau entgegengesetzt bewegen.

4.1.2.2 Y-Z-Ebene

Da die Bewegung der Arme in der Y-Z-Ebene durch den Körper des Roboters nach innen beschränkt ist, sieht die zugehörige Trajektorie etwas anders aus:

$$\omega_{Arm_y}(t) = \frac{\pi}{4} ArmAmplitudeY \cdot \Big(\sin\big(2\pi(t + ArmPhaseY)\big) + 1\Big)$$

mit Phasenversatz $ArmPhaseY = 0.5$ konstant für diese Arm-Bewegung. Diese Form der Trajektorie wird als Winkel $\omega_{Arm_y}(t)$ direkt auf das jeweilige Arm-Gelenk 2 addiert und stellt sicher, dass der Arm nicht gegen den Körper gelenkt wird. Ferner wird der Arm nur bis zur Waagerechten, und nicht etwa über die Schulter-Höhe, angehoben. Es wird also entgegen der Armbewegung in der X-Z-Ebene nicht ein maximal möglicher Halbkreis[4], sonderen lediglich maximal ein Viertelkreis beschrieben, in dessen Anfangspunkt der Arm senkrecht nach unten zeigt und, je nach Wert von $ArmAmplitudeY \in [0, 1]$, bis zur waagerechten Ausrichtung angehoben wird. Auch bei dieser Bewegung gibt es zwischen den beiden Arm-Gelenken nochmals einen Phasen-Versatz von 0.5, so dass sich die Arme stets entgegengesetzt bewegen.

4.1.3 Oberkörper

Um die Schwerpunktverlagerung des Roboters durch das Anheben eines Fußes noch weiter zu kompensieren, als dies alleine durch die Armbewegungen (Abschnitt 4.1.2) möglich ist, wird zudem der Oberkörper in der Y-, aber auch in X-Richtung bewegt. Einen Eindruck von den beiden Bewegungen vermitteln die Abbildungen 4.5(a) und 4.5(b).

[3] Wenn sich beide Beine bei der X-Koordinate 0 befinden (d.h. ein Bein steht auf dem Boden in der Mitte der unteren Geraden der Trajektorie und das andere Bein befindet sich in der Mitte der oberen Trajektorien-Hälfte), dann weist der Arm senkrecht nach unten.

[4] Wie groß der Kreisabschnitt bei der Trajektorie in der X-Z-Ebene ist, hängt vom Parameter $ArmAmplitudeX \in [0, 1]$ ab, der bei Wert 1 einen Halbkreis erzeugt und bei Wert 0 gar keine Bewegung. Für den Wert $0,5$ würde beispielsweise ein Viertelkreis erzeugt, in dessen Mittelpunkt der Arm senkrecht nach unten „hängen“ würde.

4.1.3.1 In Y-Richtung

Die Bewegung des Roboter-Oberkörpers in die Y-Richtung ist die wichtigere der beiden Oberkörper-Auslenkungen, daher wird sie hier zuerst erläutert. Wenn ein Bein entlang der Fußtrajektorie angehoben wird, verlagert sich der Roboter-Schwerpunkt stark in diese Richtung. Dies wird zwar zum Teil durch die Armtrajektorien kompensiert, jedoch ist zudem eine Seitwärtsbewegug des Oberkörpers vonnöten. Dies ist auch beim Gang von Menschen zu beobachten. Die Ansteuerung der Bewegung erfolgt ähnlich der Armtrajektorien, auch hier existiert eine direkte Kopplung an die Beinbewegungen. Bei jedem Vollschritt beider Beine wird der Oberkörper einmal nach rechts und nach links ausgelenkt (siehe Abbildung 4.5(a)). Dazu werden die Beingelenke 1 und 5 Sinus-förmig mit angemessener Amplitude ausgelenkt. Eine lineare Ansteuerung wird aus denselben Gründen wie bei den Armbewegungen (Abschnitt 4.1.2) nicht verwendet. So ist die entsprechende Trajektorie definiert als:

$$\omega_{Body_y}(t) = \frac{\pi}{2} BodyAmplitudeY \cdot \sin(2\pi(t + BodyPhaseY))$$

Hier wird der Phasenversatz $BodyPhaseY = 0$ konstant gesetzt, was bedeutet, dass der Oberkörper sich zur maximalen Amplitude nach rechts lehnt, wenn sich das linke Bein in der Mitte der oberen Beintrajektorien-Hälfte befindet. Der Oberkörper ist jeweils in Mittelstellung (also komplett senkrecht ohne Auslenkung), wenn sich beide Beine auf dem Boden befinden. Der Wert von $\omega_{Body_y}(t)$ wird auf die Werte der beiden Beingelenke 1 addiert; die Beingelenke 5 werden dagegen mit $-\omega_{Body_y}(t)$ additiv verändert. Es wird hier addiert auf den Winkel, der durch die inverse Kinematik als Folge der Soll-Positionen der Fußtrajektorie berechnet wurde (Abschnitt 4.2.2). Dies bewirkt letztendlich, dass sich der Oberkörper parallel zum Boden auf dem Ausschnitt eines Kreises bewegt; der Kreismittelpunkt liegt in der Mitte zwischen den beiden Beingelenken Nummer 5.

Diese Pendel-Bewegung des Körpers in der Y-Richtung ist schon für ein Geradeauslaufen wichtig, erhält jedoch noch mehr Bedeutung für diagonales und vor allem für seitwärtiges Laufen. Wenn der Roboter seitwärts läuft[5], so ist es notwendig, dass sich der Roboter noch viel weiter zur Seite lehnt, damit die Füße überhaupt zur geforderten Schrittweite ausholen können; ansonsten würde der Roboter stürzen.

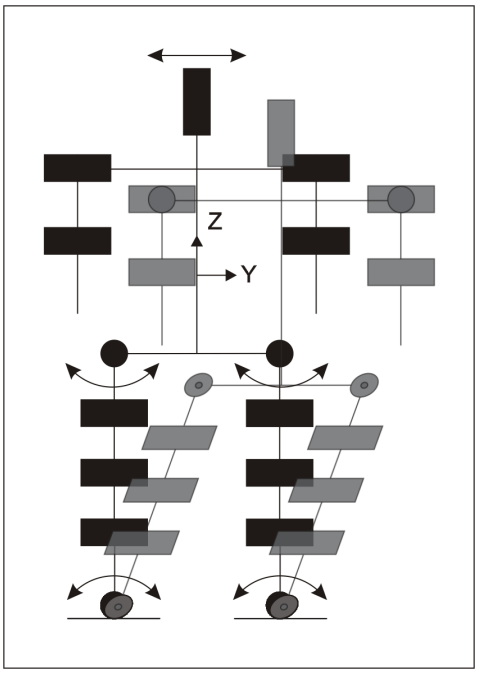
4.1.3.2 In X-Richtung

Die Bewegung des Oberkörpers in der X-Richtung wurde nur der Vollständigkeit halber eingeführt, dabei wird der Oberkörper in etwa so bewegt, wie es bei laufenden Vögeln zu beobachten ist: mit jedem einzelnen Schritt schwingt der Körper leicht nach vorne, um den Körper während des Schrittes leicht nach vorne kippen zu lassen. Da diese Bewegung für jeden Einzelschritt durchzuführen ist, wird sie im Vergleich mit den anderen Trajektorien (bei den Armen und bei der Oberkörper-Bewegung in Y-Richtung) mit der doppelten Frequenz betrieben. Ferner wird die Bewegung nur in die Laufrichtung ausgeführt, wie in Abbildung 4.5(b) für ein Geradeauslaufen zu erkennen ist. Sie ist somit definiert als:

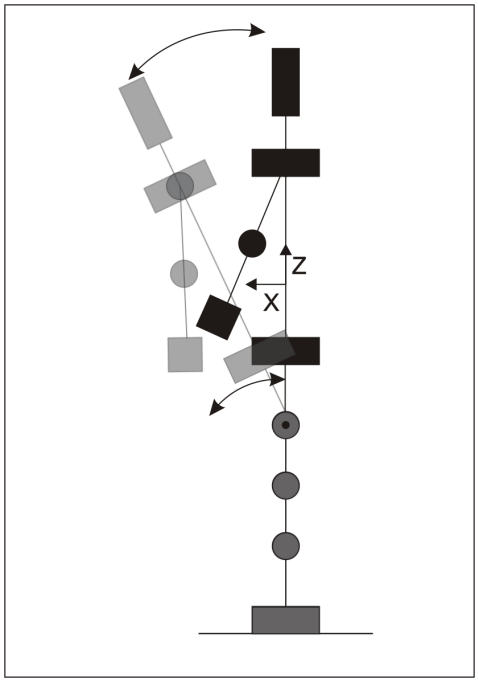
$$\omega_{Body_x}(t) = \frac{\pi}{4} BodyAmplitudeX \cdot (1 + \sin(4\pi(t + BodyPhaseX)))$$

Der Winkel $\omega_{Body_x}(t)$ wird additiv dem Winkel der beiden Fußgelenke 2 hinzugefügt, da durch die inverse Kinematik ja sichergestellt ist, dass der Oberkörper stets aufrecht ausgerichtet ist.

[5] Entsprechend 90° oder 270° Laufwinkel, wobei die $0° - 360°$ wie auf einem Kompass aufzufassen sind.



(a) Die Y-Body-Trajektorie in der Y-Z-Ebene. In schwarz der Aufrecht stehende und in grau der durch die Y-Body-Trajektorie zur Seite ausgelenkte Roboter.



(b) Die X-Body-Trajektorie in der X-Z-Ebene. Hier ist in schwarz der Aufrecht stehende und in grau der nach vorne geneigte Roboter zu sehen.

Abbildung 4.5: Schematische Darstellung der beiden Body-Trajektorien.

So kann, unabhängig von der Position der Füße und ferner unabhängig von den gesetzten Winkeln der Beingelenke, mit einer Addition des Winkels $\omega_{Body_x}(t)$ die gewünschte Auslenkung des Oberkörpers erreicht werden. Für ein omnidirektionales Laufen wird $\omega_{Body_x}(t)$ ferner mit dem Cosinus des Laufwinkels skaliert, was auch automatisch dazu führt, das für ein Rückwärtslaufen der Oberkörper entsprechend nach hinten wippt, da für Lauf-Winkel α mit $90° < \alpha < 270°$ der Cosinus negativ wird und somit den Oberkörper zur anderen Seite herüberschwenkt. Der Phasenversatz $BodyPhaseX = 0$ ist hier konstant, damit der Oberkörper mit jedem einzelnen Schritt in die Laufrichtung wippt. Ob diese Bewegung für das Laufen des Roboters wirklich von Vorteil ist, kann hier noch nicht abgeschätzt werden. Es ist jedoch anzunehmen, dass das Optimierverfahren die Amplitude dieser Bewegung auf Null setzt, falls die Laufbewegung durch diese ansonsten nachteilig beeinflusst würde.

4.2 Inverse Kinematik

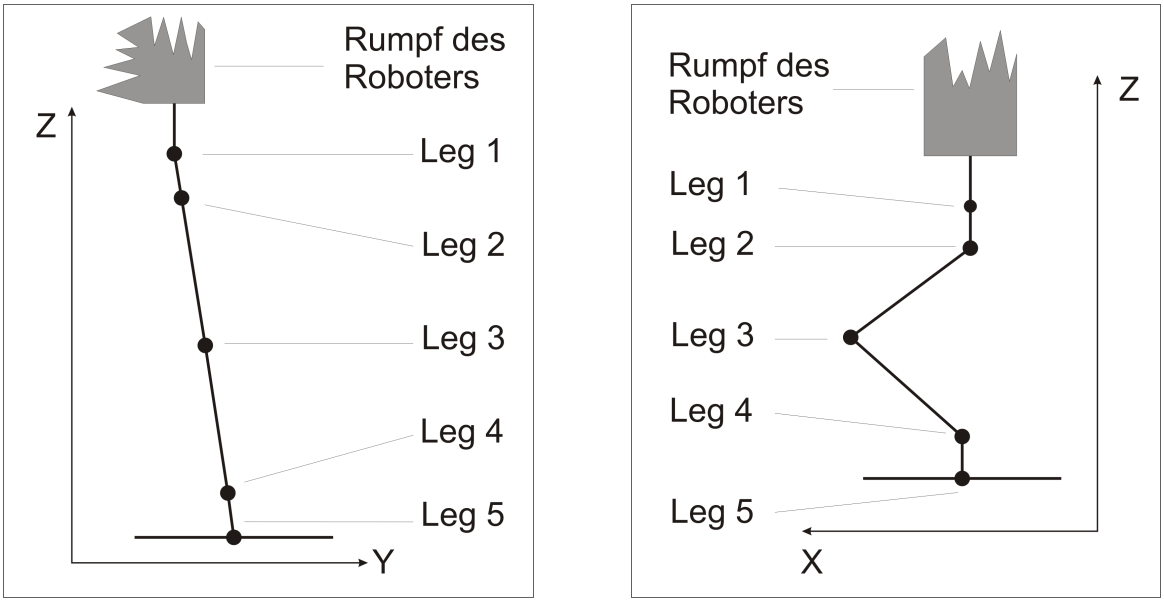
Bei der *Inversen Kinematik* geht es um das Problem, zu einer für einen Teil des Roboters angegebenen Soll-Position im Raum die dafür benötigten Winkel und Translationen der Roboter-Aktoren zu errechnen. Hier wird die Berechnung der inversen Kinematik durchgeführt, um die benötigten Gelenkwinkel für die Verfolgung der Trajektorien zu ermitteln.

4.2.1 Definitionen

Bevor das Problem der inversen Kinematik näher erläutert wird, werden noch einige Begriffe geklärt. Ein Roboter (egal, ob ein einzelner Roboter-Arm aus der Industrie oder ein humanoider Roboter wie der Kondo KHR-1) besteht aus beweglichen Teilen. Brillowski [Bri04] definiert hierzu die *Kinematische Kette*:

„Die mechanische Struktur eines Industrie-Roboters kann näherungsweise als ein Mehrkörpersystem modelliert werden, das aus miteinander verketteten massebehafteten Starrkörpern besteht, die zueinander Relativbewegungen ausführen. Diese Starrkörper werden im folgenden als Glieder oder Achskörper in einer kinematischen Kette bezeichnet. Zwischen diesen Gliedern existieren an diskreten Punkten Gelenke, die die Bewegungen einschränken.“

Diese Definition, die sich auf Industrieroboter mit rotatorischen Gelenken bezieht, reicht für die Arbeit mit dem Kondo KHR-1 in dem hier betrachteten Ausmaß vollkommen aus, da der Kondo KHR-1 nur über rotatorische und nicht über translatorische Gelenke verfügt.



(a) Das rechte Bein in der Y-Z-Ebene, als Ansicht von vorne mit den einzelnen Gelenk-Bezeichnungen.

(b) Das rechte Bein in der X-Z-Ebene als Ansicht von rechts mit den Gelenk-Beschriftungen.

Abbildung 4.6: Schematische Ansicht der Beingelenke des Kondo KHR.

Zum Thema *Koordinatensystem* fährt Brillowski fort:

„Eine kinematische Kette trägt üblicherweise einen Endeffektor. Das ist ein Werkzeug oder ein Greifer, über den der Roboter mit seiner Umgebung in Kontakt tritt. Um die Position und Orientierung des Endeffektors mathematisch beschreiben zu können, werden den Gliedern eines Roboters Koordinatensysteme zugeordnet. Die Koordinaten legen die Lage eines Gliedes in der kinematischen Kette bezüglich des vorherigen und nachfolgenden Gliedes fest. Verschiebungen und Drehungen der Glieder können dann mit Hilfe von Transformationen und Vektorbeziehun-

gen zwischen den zugeordneten Koordinatensystemen beschrieben werden.“

Im Falle der Laufsteuerung des Kondo KHR-1 interessieren als Endeffektoren lediglich die Füße, die durch eine Berührung, die alleine durch die Schwerkraft erfolgt, in Kontakt mit ihrer Umwelt treten. Im Speziellen sollen ja die Füße den vorgesehenen Trajektorien entlang geführt werden. Dies wirft allerdings ein Problem auf. Wenn die einzelnen Winkel der Gelenke bekannt sind, kann die Raum-Position des Endeffektors über die oben erwähnten Transformationen und Vektorbeziehungen sukzessive über alle der beteiligten Koordinatensysteme leicht errechnet werden.

Bei der Verfolgung der Trajektorien in diesem Falle ist die Aufgabenstellung jedoch genau umgekehrt: Es werden Positionen im Raum vorgegeben und die dafür benötigten Gelenkwinkel gesucht. Dieses Problem wird als *Inverse Kinematik* bezeichnet. Aufgrund des nichtlinearen Zusammenhangs ist diese Lösung für Kinematische Ketten im allgemeinen nicht eindeutig (es gibt verschiedene Winkeleinstellungen für die Gelenke, die alle dieselbe Position & Ausrichtung des Endeffektors bestimmen) und für umfangreichere Roboter-Konstruktionen auch nicht geschlossen lösbar. Für Details sei auf Riesler [Rie92] verwiesen.

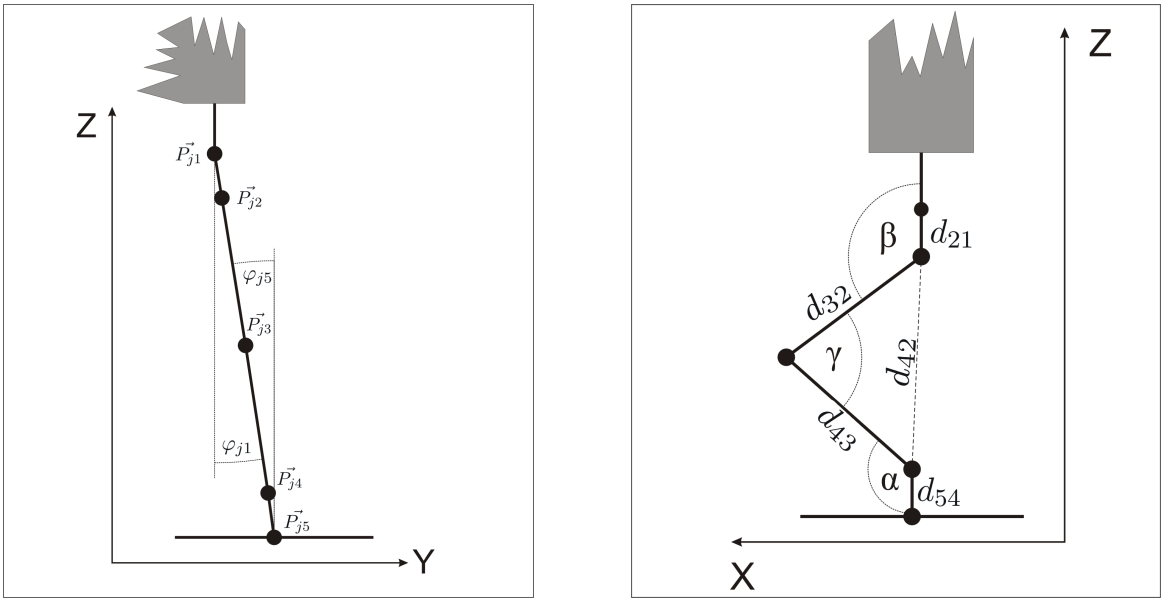
Nur für bestimmte Fälle kann die inverse Kinematik durch geometrische Berechnungen geschlossen gelöst werden, dies ist nur bei Robotern mit wenigen Gelenken und nur in bestimmten Anordnungen der Fall. Ansonsten kann die inverse Kinematik nur über Approximations-Verfahren genähert werden, z.B. beschreiben Meredith et al. [MM04] eine Alternative über die Invertierung der Jacobi-Matrix[6] des Roboters. Die Inverse der Jacobi-Matrix kann, soweit überhaupt existent, nur näherungsweise gelöst werden und ist zudem leider sehr rechenaufwändig. Versuche von Kindler [Kin06] haben gezeigt, dass die Berechnung der Fuß-Positionen des Kondo KHR-1 auf den vorgegebenen Trajektorien einen Zeitaufwand im Sekundenbereich auf einer aktuellen PC-Plattform in Anspruch nimmt und daher nicht geeignet ist für die schnelle Ansteuerung der Füße während des Laufens.

Allerdings ist es mit der Vorgabe, dass die Knie des Kondo-KHR1 nur nach vorne einknicken, möglich, das Problem der inversen Kinematik bei den Kondo-Beinen geschlossen über einen geometrischen Ansatz zu lösen. Wird zunächst o.B.d.A. davon ausgegangen, dass die Trajektorie lediglich in der X-Z-Ebene definiert ist (dies gilt für ein Laufen direkt nach vorne oder direkt nach hinten), so beruhen sämtliche Berechnungen auf der Tatsache, dass die einzigen Gelenke, die die Führung des Fußmittelpunktes auf dieser Trajektorie sicherstellen, die Beingelenke 2, 3 und 4 sind. Die Beingelenke 1 und 5 können die Beine lediglich seitwärts bewegen, wobei die Beine, bei Beschränkung auf diese beiden Gelenke, weiter nach oben gehoben würden. Eine Übersicht über die Bewegungsmöglichkeiten geben die Abbildungen 4.6(a) und 4.6(b).

4.2.2 Berechnung für den Kondo-KHR1

Bevor nun die Berechnung der inversen Kinematik, hier benutzt für die Positionierung der Roboter-Füße im dreidimensionalen Raum, erläutert wird, werden einige Bezeichnungen ein-

[6] Mit der Jacobi-Matrix eines Roboters wird der Zusammenhang zwischen der kartesischen Geschwindigkeit des Endeffektors, bezogen auf das Roboter-Basis-Koordinatensystem und die Gelenkgeschwindigkeiten, beschrieben.



(a) Vorderansicht des rechten Beines mit den Winkeln für die Beingelenke 1 und 5.

(b) Seitenansicht eines Beines mit dem Dreieck,welches aus den Beingelenken 2, 3 und 4 gebildet wird.

Abbildung 4.7: Die Bezeichnungen der Winkel und Abschnittslängen für ein Bein.

geführt (die Positionen der Gelenke sind jeweils bezogen auf das körpereigene Koordinatensystem im dreidimensionalen Raum):

$\vec{P_{j1}}...\vec{P_{j5}}$ = Position des Mittelpunktes von Beingelenk 1...5

$\varphi_{j1}...\varphi_{j5}$ = Winkel der Beingelenke 1...5

d_{ij} = Abstand zwischen den Mittelpunkten von Gelenk i und j

Dabei ist $\vec{P_{j1}}$ fest, da das Beingelenk 1 fest und unbeweglich mit der „Hüfte“ des Roboters verbunden ist. $\vec{P_{j5}}$ ist die gesuchte Position, die der Fuß auf der Trajektorie einnehmen soll (im Folgenden auch als $TargetPosition$ bezeichnet). Die nun folgenden Berechnungen beziehen sich lediglich auf das Abfahren der Fußtrajektorien durch den Roboter-Fuß über die WalkingEngine. Weitere Bewegungen der WalkingEngine, wie etwa das Pendeln des Oberkörpers (siehe Abschnitt 4.1.3), die Neigung des Körpers (Abschnitt 4.5, Parameter $FootTiltX$ und $FootTiltY$) oder das Auseinanderspreizen der Beine (Abschnitt 4.5, Parameter $YOffset$) werden hier aus Gründen der Übersichtlichkeit nicht behandelt. Eine Beschreibung ihres Einflusses auf die hier getätigten Berechnungen werden in Abschnitt 4.5 behandelt.

Die gewünschte Koordinate des Fußes in Y-Richtung ist relativ leicht zu errechnen, da hierfür lediglich die Beingelenke 1 und 5 zuständig sind, wie in Abbildung 4.7(a) leicht ersichtlich ist. Da die Fußsohle des Roboters stets parallel zum Oberkörper gehalten werden soll (wie

schon erwähnt werden Neigungen wie z.B. $FootTiltY$ hier nicht betrachtet), muss somit zwingend $j1 = -j5$ gelten. Es ist für $j1$ und $j5$ also der Winkel der Geraden durch $\vec{P_{j1}}$ und $\vec{P_{j5}} = TargetPosition$ zu berechnen, dies gilt für die Y-Z-Ebene, da eine Variation in der X-Ebene die beiden Beingelenke 1 und 5 aufgrund deren Ausrichtung ja nicht betrifft. Also errechnet sich:

$$\varphi_{j1} = \arctan\left(\frac{\vec{P_{j5_y}} - \vec{P_{j1_y}}}{\vec{P_{j5_z}} - \vec{P_{j1_z}}}\right)$$

wobei als $\vec{P_{j1_y}}$ die Y-Koordinate der Raum-Position von Beingelenk 1 zu verstehen ist. Zu beachten ist, dass zum Verändern der Y-Koordinate des Fußes dieser zwangsweise auch seine Z-Kooridinate ändert. Bei komplett ausgestrecken Beinen bedeutet dies, dass ein Abspreizen des Beines dazu führt, dass sich das Bein nach oben in Richtung Rumpf bewegt. Dies muss bedacht werden, da hierdurch unter anderem die maximale Schritthöhe sinkt.

Nun bleiben noch drei Winkel zu errechnen, dies sind die Beinwinkel 2, 3 und 4. Dabei wird ja Beingelenk 3 gemäß Vorgabe nur nach vorne eingeknickt, also nur positive Winkel sind erlaubt. Da diese drei Gelenke allesamt ihre Achse in der Y-Koordinatenrichtung angebracht haben, können nur Änderungen in den X-und Z-Koordinaten erfolgen. Zunächst wird die Position des Gelenkes 2 durch die Winkelstellung von Gelenk 1 errechnet, dies geschieht mittels:

$$\vec{P_{j2}} = \vec{P_{j1}} + \begin{pmatrix} 0 \\ d_{21}\sin(j1) \\ d_{21}\cos(j1) \end{pmatrix}$$

ähnlich verhält es sich mit dem Winkel von Gelenk 4:

$$\vec{P_{j4}} = \vec{P_{j5}} - \begin{pmatrix} 0 \\ d_{54}\sin(j1) \\ d_{54}\cos(j1) \end{pmatrix}$$

da ja die Verbindungsstücke zwischen Gelenk 4 und 5 einerseits und Gelenk 1 und 2 andererseits den selben Winkel in der X-Z-Ebene bezüglich des Roboter-Koordinatenursprungs haben müssen, damit die Fußsohlen parallel zum Roboter-Rumpf bleiben. Nun wird das Gelenk 2 in Richtung der Soll-Position bewegt:

$$\varphi_{j2} = \arctan\left(\frac{\sin(j1)(\vec{P_{j4_y}} - \vec{P_{j2_y}}) - \cos(j1)(\vec{P_{j4_z}} - \vec{P_{j2_z}})}{\vec{P_{j4_x}} - \vec{P_{j2_x}}}\right)$$

Für die weiteren Berechnungen wird ferner der Abstand zwischen Gelenk 2 und Gelenk 4 benötigt, dieser kann leicht über die bereits ermittelten Positionen dieser beiden Gelenke errechnet werden:

$$d_{42} = |\vec{P_{j4}} - \vec{P_{j2}}| = \sqrt{(\vec{P_{j4_x}} - \vec{P_{j2_x}})^2 + (\vec{P_{j4_y}} - \vec{P_{j2_y}})^2 + (\vec{P_{j4_z}} - \vec{P_{j2_z}})^2}$$

Als letzte Winkel fehlen nun noch α, β und γ, wie in Abbildung 4.7(b) dargestellt. Es handelt sich hier um das Dreieck, welches durch die Punkte $\vec{P_{j2}}$, $\vec{P_{j3}}$ und $\vec{P_{j4}}$ in der X-Z-Ebene gegeben ist. Das Dreieck ist durch dessen drei Seiten d_{32}, d_{43} und d_{42} eindeutig bestimmt, die Winkel

können nun über den Cosinus-Satz sowie die Tatsache, dass die Winkelsumme im Dreieck 180° beträgt, leicht errechnet werden:

$$\begin{aligned} \alpha &= \arccos\left(\frac{d_{43}^2+d_{42}^2-d_{32}^2}{2d_{43}d_{42}}\right) \\ \beta &= \arccos\left(\frac{d_{32}^2+d_{42}^2-d_{43}^2}{2d_{32}d_{42}}\right) \\ \gamma &= \pi-\alpha-\beta \end{aligned}$$

Um das Einknicken des Knies nun zu erreichen, wird β nun zu dem bereits errechneten Wert von φ_{j4} addiert, weiterhin wird gesetzt:

$$\begin{aligned} \varphi_{j3} &= \gamma - 180° \\ \varphi_{j4} &= -\varphi_{j2} - \varphi_{j3} \end{aligned}$$

Wie bereits oben beschrieben, wird $\varphi_{j5} = -\varphi_{j1}$ gesetzt. Nun sind alle Winkel der Fußgelenke berechnet, und der Fuß kann bewegt werden, wie von der Trajektorie vorgegeben. Diese Berechnung ist für beide Füße identisch (auch wenn die Soll-Positionen der Füße beim Laufen unterschiedlich sind), lediglich für die Gelenke 1 und 5 gelten unterschiedliche Vorzeichen.

4.2.3 Anhebung der Trajektorie

Ein Problem bei diesen Berechnungen wurde bisher jedoch nicht behandelt: Wenn man von komplett ausgestreckten Beinen beim aufrecht stehenden Roboter ausgeht (die Position beider Füße sei im Mittelpunkt des unteren Trajektorien-Abschnitts), so sind die Endpunkte der Trajektorie (beim Wechsel vom oberen zum unteren Trajektorien-Abschnitt) mit den Füßen nicht zu erreichen, da ja die Füße bereits im Mittelpunkt des unteren Trajektorien-Abschnitts gestreckt sind und der Weg zum Trajektorien-Endpunkt weiter vom Rumpf entfernt ist, als der untere Trajektorien-Mittelpunkt. Zum Erreichen der Trajektorien-Endpunkte müssten die Beine also noch weiter gestreckt werden, was per Voraussetzung ja nicht möglich ist. Ein Ausweg aus diesem Problem ist, die Beine beim senkrechten Stehen so weit anzuwinkeln, dass mit gestrecktem Bein die Trajektorien-Endpunkte gerade erreicht werden können. Es wird also die gesamte Fußtrajektorie in Richtung Roboter-Rumpf angehoben, diese Entfernung wird hier im Folgenden mit $HeightOffset$ bezeichnet.

Das Problem ist nun vorstellbar als rechtwinkliges Dreieck, dessen rechter Winkel vom senkrechten Roboterbein und dem Boden gebildet wird (eine Übersicht gibt Abbildung 4.8). Die Hypothenuse des Dreiecks ist das ausgestreckte Bein zwischen Rumpf und Trajektorien-Endpunkt. Die Kathete, die dem Hüftgelenk gegenüber liegt und vom Bodenpunkt senkrecht unter dem Roboter zum Trajektorien-Endpunkt führt, ist $StepLength/2$ lang, die Länge der anderen Kathete ist hier gesucht. Die Strecke, um die das Bein nun in Richtung Rumpf angehoben werden muss, ist $HeightOffset = Beinlänge - Kathetenlänge$. So berechnet sich dann die Länge der Kathete über den Satz des Pythagoras als:

$$Kathetenlänge = \sqrt{Beinlänge^2 - \left(\frac{StepLength}{2}^2\right)}$$

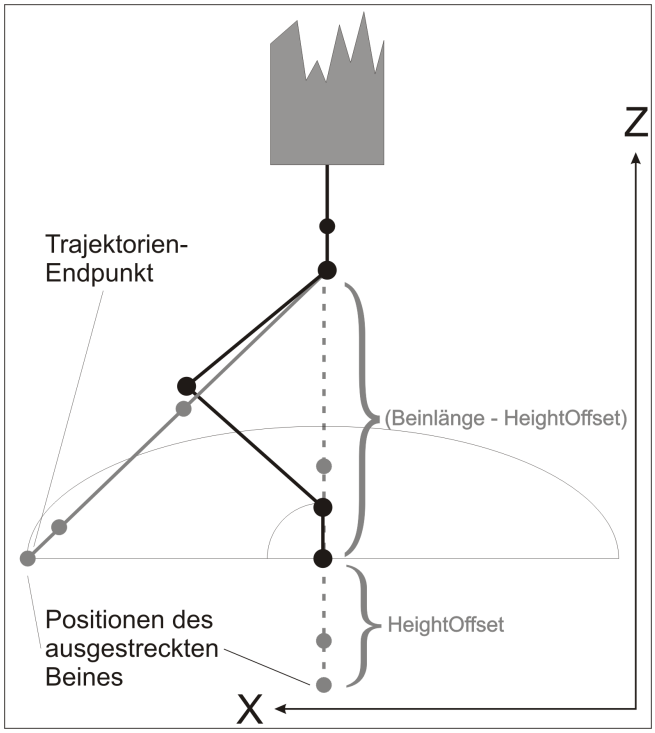


Abbildung 4.8: Schematische Darstellung der Berechnung von *HeightOffset*. Es ist ein Bein aus der Seitenansicht zu sehen. Die Trajektorie wird um *HeightOffset* angehoben, damit das ausgestreckte Roboterbein alle Trajektorien-Punkte erreichen kann.

Wegen $HeightOffset = Beinlänge - Kathetenlänge$ haben wir also:

$$HeightOffset = Beinlänge - \sqrt{Beinlänge^2 - \left(\frac{StepLength}{2}^2\right)}$$

Diese Anhebung der Trajektorie wird vor der Berechnung der inversen Kinematik für die Fuß-Ansteuerung in Trajektorien-Form durchgeführt. Dies findet in einer eigenen Methode statt, die vom Modul für die Berechnung der inversen Kinematik aufgerufen wird. Da ein Auseinanderspreizen der Beine (siehe Parameter $YOffset$ in Abschnitt 4.5) auch ein Anheben der Trajektorie notwendig macht, wird in diesem Fall analog verfahren und die Trajektorie um die für $YOffset$ notwendige Strecke noch weiter erhöht. Nun sind alle Berechnungen der inversen Kinematik komplett, die Gelenk-Winkel des Roboters können gemäß den Berechnungen gesetzt werden.

4.3 Timing der Bewegungen

Wie schon in Abschnitt 4.1 beschrieben, wird der Parameter t mit $0 \leq t < 1$ benutzt, um die Position in der jeweiligen Trajektorie zu bestimmen. Dieser Parameter wird über die aktuelle Systemzeit determiniert, er wird auf den Bereich $[0, 1]$ normiert anhand:

$$t = \frac{AktuelleZeit \mod (StepDuration)}{StepDuration}$$

Mit dem Parameter *StepDuration*, welcher die Zeit für einen vollen Schritt mit beiden Beinen in [ms] angibt. Je nachdem, welchen Wert die aktuelle Systemzeit gerade annimmt, befindet sich t also zwischen 0 und 1. Dies bedeutet aber, dass beim ersten Aufruf der WalkingEngine in den allermeisten Fällen mit einem Wert von $t \neq 0$ begonnen wird, was dazu führt, dass beim ersten Durchlauf der WalkingEngine-Schleife direkt eine Position innerhalb des Schrittes sprungartig von den Servos angefahren wird. Dies sorgt nicht nur für einen teilweise enormen Ruck durch den Roboter, der diesen zu Fall bringen kann, sondern sorgt auch, in Abhängigkeit der aktuellen Zeit, für einen ständig anderen Übergang in die Laufbewegung. Um ständig den gleichen Start in die Laufbewegung zu erreichen, ist die WalkingEngine so gestaltet worden, dass bei ihrem Start, also jeweils beim Aufruf eines WalkRequests (siehe Abschnitt 2.3.1), der aktuelle Wert von t für den weiteren Verlauf gespeichert und als Offset vom jeweils errechneten Wert von t abgezogen wird. Unterschreitet der Wert die Grenze 0, so wird wiederum eine 1 addiert. So werden auch die Grenzen $0 \leq t < 1$ eingehalten, aber beim Start der WalkingEngine beginnt die Bewegung deterministisch mit $t = 0$, so dass beide Beine zu Beginn auf dem Boden stehen. Bei jedem Neustart der WalkingEngine, also bei jedem neuen WalkRequest, der nach einer inaktiven Phase der WalkingEngine folgt, wird dieser Offset neu errechnet und gespeichert.

Da zum Erreichen dieser Position innerhalb eines Durchlaufes der WalkingEngine (also innerhalb 20ms) die Füße aber immer noch um die vorgegebene Schrittlänge auseinandergerissen werden müssten, wird eine Beschleunigung eingebaut, die die Schrittweite stufenweise bis auf den geforderten Wert erhöht. So wird die WalkingEngine bei $t = 0$ so gestartet, dass die Schrittlänge 0 anliegt, der Roboter also auf der Stelle tritt. Mit jedem vollständigen Schritt wird die Schrittlänge um einen konstanten Betrag erhöht, so dass erst nach einigen Schritten des Roboters die geforderte Schrittlänge und somit auch die geforderte Geschwindigkeit erreicht wird. Dies ist auch noch aus einem weiteren Grund erforderlich: Der Roboter kann nicht aus dem Stillstand auf hohe Geschwindigkeiten beschleunigt werden, es wird eine Beschleunigungs-Zeit benötigt. Zudem ist es für die Stabilität vorteilhaft, wenn sich der Roboter auf die Laufbewegung einschwingen kann und erst langsam die Schrittgröße erhöht. Das initiale Treten auf der Stelle ist dafür ein guter Ansatz. Bei einer geforderten Verringerung der Laufgeschwindigkeit, auch bei gefordertem Stillstand, wird analog verfahren und die Schrittweite sukzessive verringert.

4.4 Omnidirektionales Laufen

Bisher wurde das Laufen nur für gerade Richtungen, also in einer Ebene beschrieben. Nun wird beim Roboterfußball allerdings ein *omnidirektionales* Laufen, also ein Laufen in beliebige Richtungen benötigt. Dies kann eine translatorische Bewegung, also ein geradliniges Laufen in Richtung eines Translations-Winkels[7] α sein, oder eine rotatorische Bewegung, bei welcher der Roboter sich um seine eigene Achse dreht. Die rotatorische Bewegung kann dabei auch mit der translatorischen kombiniert werden, so dass vom Roboter ein Kreisbogen mit beliebigem Radius gelaufen wird.

Wie bereits erwähnt, ist der Kondo KHR-1 aufgrund seiner Bauart nicht direkt befähigt, sich

[7] Vergleichbar mit einem Kompass, ein Winkel von 0° entspricht einem Laufen direkt nach vorne, 45° führen diagonal nach vorne rechts, 90° ist ein Seitwärts-Laufen nach rechts, usw..

mit Hilfe eines dafür vorgesehenen Servos auf der Stelle oder beim Laufen zu drehen, da die entsprechenden Gelenke mit einer Achse in Z-Koordinaten-Richtung fehlen. Als Ausweg kann sich der Roboter auf der Stelle drehen, indem die Beine auseinandergezogen werden (durch einen Halbschritt nach vorne oder nach hinten), und der Roboter dann mit beiden Füßen auf dem Boden die Beine wieder zusammenzieht. Dadurch dreht sich der Roboter auf einem Boden, der den Füßen ein Rutschen gestattet, in die Richtung des Fußes, der zuvor hinten stand. Dies funktioniert jedoch nur auf rutschendem Boden, ferner kann nicht genau errechnet werden, wie weit sich der Roboter drehen wird, da es dabei einen Versatz durch die Bodenreibung gibt.

Eine Kombination von Rotation und Translation ist nur dadurch möglich, dass der Roboter ständig zwischen Translation und Rotation wechselt. Dies bedingt dann einen sehr unsteten und langsamen Lauf. Alternativ kann der Roboter mit dem Oberkörper verstärkt auf eine Seite gelehnt werden, was ein „Schleifen“ des Fußes an der betroffenen Körperseite verursacht. Dadurch wird der Roboter auf dieser Seite verlangsamt, was zu einer Drehbewegung in genau diese Richtung führt. Diese Methode ist zwar auch einsetzbar, aber leider ist das Ausmaß der Drehung kaum berechenbar, da sie von der Reibung auf dem Boden abhängt. Ferner ist der Roboter ständig gefährdet, umzufallen.

Eine translatorische Bewegung um einen beliebigen Translationswinkel α ist wesentlich einfacher zu erreichen: In der hier vorgestellten WalkingEngine werden die Trajektorien der Füße um den geforderten Winkel gedreht. Dazu wird die X-Koordinate der Fußposition mit dem Cosinus des Winkels α multipliziert; die Y-Koordinate dagegen mit dessen Sinus. Dadurch wird der Fuß auf einen Punkt auf dem Kreis um den Robotermittelpunkt gesetzt, und zwar um genau den gewünschten Winkel α. Die Entfernung des Fußes vom Roboter-Mittelpunkt wird hierdurch nicht verändert, da stets die beschriebene Kreisbahn eingehalten wird. Allerdings würde alleine dieses Vorgehen zu schweren Problemen beim Seitwärtsgehen führen: die Beine würden mittig unter dem Oberkörper kollidieren, weil sich dort die Trajektorien überlappen würden.

Um diese Kollision zu verhindern werden die Beine entsprechend weiter gespreizt, und zwar um:

$$Offset_{feet_y} = \frac{StepLength}{2} |\sin(\alpha)|$$

Dieser Offset wird nun auf die Y-Koordinaten der Fußpositionen addiert, bevor die inverse Kinematik berechnet wird. Dabei wird $Offset_{feet_y}$ zur Y-Koordinate des rechten Fußes und $-Offset_{feet_y}$ zur Y-Koordinate des linken Fußes addiert. Es wird hierbei auch sichergestellt, dass die Schrittlänge stets eingehalten wird; dadurch dass sich die beiden Trajektorien nun praktisch „hintereinander“ befinden, wird jeder Fuß nun nur noch um $StepLength/2$ vorangesetzt, was sich bei den beiden hintereinander gelagerten Trajektorien der Füße dann genau wieder zu $StepLength$ pro Gesamtschritt summiert.

4.5 Die Parameter der WalkingEngine

Einige der Parameter der WalkingEngine sind in den oberen Abschnitten kurz erwähnt worden, hier folgt nun eine komplette Übersicht aller 15 Parameter der vorgestellten WalkingEngine. Dies sind:

- **StepDuration**
  StepDuration bestimmt die Zeit für einen kompletten Schritt (beide Beine fahren ihre Trajektorie jeweils einmal komplett ab, sonstige Bewegungen entsprechend gekoppelt) in [ms]. Wertebereich: $700 \leq StepDuration \leq 3000$, wobei *StepDuration* min. 700 aus Gründen der maximalen Servo-Geschwindigkeit und der Stabilität des Roboters bei zu hohen Bewegungs-Geschwindigkeiten und *StepDuration* max. 3000, weil noch höhere Werte zu sehr langsamen Bewegungen führen.

- **StepHeight**
  Die Schritthöhe und somit die maximale Höhe der Beintrajektorie in [mm] wird per $StepHeight$ definiert. Wertebereich: $0 \leq StepHeight \leq 50$ aufgrund der Roboter-Dimensionen (Länge der Beine).

- **StepLength**
  StepLength ist die Schrittlänge und somit die Länge des unteren Fußtrajektorien-Bereiches in [mm]. Wertebereich: $0 \leq StepLength \leq 100$ aufgrund der Roboter-Dimensionen (Länge der Beine)

- **ArmAmplitudeX**
  Die maximale Auslenkung der Arme in der X-Z-Ebene. Wertebereich:
  $0 \leq ArmAmplitudeX \leq 1$. Dabei entspricht $ArmAmplitudeX = 0$ einem unbeweglichen Arm, $ArmAmplitudeX = 1$ lässt den Arm mit maximaler Amplitude nach vorne und hinten pendeln.

- **ArmPhaseX**
  Die Phasenlage der Armtrajektorie in Bezug auf die Beinbewegung. Wertebereich: $0 \leq ArmPhaseX \leq 1$. Wird hier konstant gesetzt: $ArmPhaseX = 0,25$, so dass der linke Arm nach vorne pendelt, wenn sich das rechte Bein nach vorne bewegt, und umgekehrt. Eine Variation dieses Parameters macht wegen der Spiegel-symmetrischen Art des Laufens keinen Sinn.

- **ArmAmplitudeY**
  Die maximale Auslenkung der Arme in der Y-Z-Ebene. Wertebereich:
  $0 \leq ArmAmplitudeY \leq 1$. $ArmAmplitudeY = 0$ entspricht einem unbeweglichen, senkrecht nach unten gerichteten Arm (Arm-Gelenk 2 um 90° nach unten geknickt), $ArmAmplitudeY = 1$ lässt den Arm mit maximaler Amplitude zur Seite bis zur Waagerechten (Winkel 0°) anheben. Ein Wert von z.B. $ArmAmplitudeY = 0,5$ lässt den Arm aus der senkrecht nach unten abgewinkelten Position um 45° nach oben heben, das heisst, dass der untere Wendepunkt dieser Pendel-Bewegung immer beim um 90° nach unten ausgerichteten Arm liegt.

- **ArmPhaseY**
  Die Phasenlage der Armtrajektorie in Bezug auf die Beinbewegung. Wertebereich: $0 \leq ArmPhaseY \leq 1$. Wird hier konstant gesetzt: $ArmPhaseY = 0,5$, so dass der linke Arm nach oben pendelt ($ArmAmplitudeY > 0$ vorausgesetzt), wenn sich das rechte Bein nach oben bewegt, und umgekehrt. Eine Variation dieses Parameters macht wegen der Spiegel-symmetrischen Art des Laufens keinen Sinn.

- **BodyAmplitudeX**
  Die maximale Auslenkung des Oberkörpers in X-Richtung. Wertebereich:

$0 \leq BodyAmplitudeX \leq 1$. $BodyAmplitudeX = 0$ entspricht einem unbeweglichen, senkrecht aufgerichteten Oberkörper, $BodyAmplitudeX = 1$ ließe den Oberkörper um 90° nach vorne herunter beugen[8]. Bei einem Rückwärtslaufen wird der Oberkörper automatisch jeweils nach hinten herübergebeugt. Zudem wird die Amplitude für Laufwinkel $\alpha \notin \{0°, 180°\}$ mit dem Cosinus des Laufwinkels skaliert, um zu verhindern, dass z.B. bei einem Seitwärtslaufen der Oberkörper nach vorne wippt und um beim diagonalen Laufen den Einfluss dieser Bewegung entsprechend zu vermindern.

- **BodyPhaseX**
  Die Phasenlage der X-Oberkörpertrajektorie in Bezug auf die Beinbewegung. Wertebereich: $0 \leq BodyPhaseX \leq 1$. Wird hier konstant gesetzt: $BodyPhaseX = 0$, so dass der Oberkörper mit jedem einzelnen Schritt, egal, ob vom rechten oder linken Roboterbein ($BodyAmplitudeX > 0$ vorausgesetzt), nach vorne wippt. Eine Variation dieses Parameters macht keinen Sinn, da sich das Oberkörper-Gewicht nach vorne verlagern soll, um den Schritt in Laufrichtung mit seinem Schwung zu unterstützen.

- **BodyAmplitudeY**
  Die maximale Auslenkung des Oberkörpers in Y-Richtung. Wertebereich:
  $0 \leq BodyAmplitudeY \leq 0,4$, da bei Werten $BodyAmplitudeY > 0,4$ die Beine mit dem Rumpf des Roboters kollidieren. $BodyAmplitudeY = 0$ entspricht einem unbeweglichen, senkrecht aufgerichteten Oberkörper, $BodyAmplitudeY = 0,4$ schiebt den Oberkörper parallel zum Boden so weit zur Seite, dass dass die „Oberschenkel“ des Roboters gerade noch nicht gegen den Rumpf (im „Hüft“-Bereich) stoßen.

- **BodyPhaseY**
  Die Phasenlage der Y-Oberkörpertrajektorie in Bezug auf die Beinbewegung. Wertebereich: $0 \leq BodyPhaseY \leq 1$. Wird hier konstant gesetzt: $BodyPhaseY = 0$, so dass sich der Oberkörper immer dann nach rechts verschiebt, wenn das linke Bein angehoben wird und umgekehrt. So wird der Oberkörper als Ausgleich für den durch das angehobene Bein seitlich verlagerten Roboter-Schwerpunkt benutzt. Eine Variation dieses Parameters macht keinen Sinn, da sich der Oberkörper zu genau der Seite bewegen muss, auf der das Bein Bodenkontakt hat.

- **YOffset**
  Mit diesem Parameter kann bestimmt werden, um wieviele Millimeter die Mittelpunkte der beiden Füße zur Seite auseinander bewegt werden, es wird also ein Spreizen der Beine verursacht. Das rechte Bein wird um $YOffset/2$ nach rechts bewegt, das linke Bein um $YOffset/2$ nach links. Dabei werden zwangsweise die Beine weiter in Richtung Oberkörper angehoben, der Roboter verringert somit seine Höhe (siehe Abschnitt 4.2.3). Wertebereich: $8 \leq YOffset \leq 50$. Kleinere Werte als 8 mm verursachen ein Berühren der Füße, größere Werte als 50 mm sorgen dafür, dass der Roboter seine Beine so weit spreizt, dass ein Bein aufgrund der breitbeinigen Lage nicht mehr angehoben werden kann, weil das Roboter-Gewicht nicht mehr auf das andere Bein verlagert werden kann. Die Bewegung der Füße um $YOffset$ auseinander, sowie das dafür benötigte Anheben der Füße (natürlich resultierend in einem Absenken des Körpers, da ja Bodenkontakt

[8] Bei einem solch hohen Wert von $BodyAmplitudeX$ wäre der Roboter allerdings schon längst nach vorne gekippt und auf den Kopf gefallen, ein Aufstehen nur mittels der Body-X-Trajektorie wäre aufgrund des verlagerten Schwerpunktes dann nicht mehr möglich.

gehalten wird) wird durch inverse Kinematik berechnet (Abschnitt 4.2), indem die Y-Position der Füße um den entsprechenden Betrag versetzt wird.

- **ZOffset**
  Mit $ZOffset$ kann der Roboter durch ein Einknicken der Beine abgesenkt werde, er geht sprichwörtlich „in die Knie“. Dadurch wird insgesamt der Schwerpunkt weiter nach unten verlagert. Dies ist für ein stabileres und weniger schwankendes Laufen gegebenenfalls von Vorteil[9]. Wertebereich: $0 \leq ZOffset \leq (50 - StepHeight)$. Bei $ZOffset = 0$ sind die Beine gerade gestreckt, sie können nicht weiter als 50 mm angehoben werden, da ansonsten die Bein-Servos so weit angewinkelt werden, dass sie aneinander stoßen. Da der Parameter $StepHeight$ bereits die Beine anhebt, kann bei maximaler Beinhöhe von 50 mm der $ZOffset$ maximal $(50 - StepHeight)$ betragen.

- **FootTiltX**
  Mittels FootTiltX kann der gesamte Roboterkörper geneigt werden, der Wert wird direkt auf den Winkel der Fußgelenke 4 addiert, wodurch sich der Körper, abhängig vom Vorzeichen von $FootTiltX$, entweder nach vorne oder nach hinten neigt. Mit dieser Neigung ist keine Bewegung verbunden, es handelt sich um einen konstanten Wert, der zur Justierung von Ungleichmäßigkeiten des Roboters genutzt werden kann. Da der Roboter z.B. nach der Montage des PocketPCs an den Oberkörper durch dessen Gewicht nach vorne geneigt wird, kann $FootTiltX$ helfen, dies auszugleichen. Wertebereich: $-\frac{\pi}{2} \leq FootTiltX \leq \frac{\pi}{2}$. Bei $FootTiltX = \frac{\pi}{2}$ liegt der Roboter nach vorne gekippt auf dem Boden, die Füße berühren dabei den Untergrund mit der gesamten Fläche. Für $FootTiltX = -\frac{\pi}{2}$ liegt der Roboter entsprechend auf dem Rücken. Bei $FootTiltX = 0$ wird der Winkel der Fußgelenke 4 nicht verändert, der Roboter würde ohne Beeinflussung von außen und bei angenommener Symmetrie der Gewichts-Verteilung senkrecht stehen.

- **FootTiltY**
  $FootTiltY$ ist das Pendant zu $FootTiltX$ in der Y-Richtung. Durch die Addition von $FootTiltY$ auf die Fußgelenke 5 wird der Roboter nach links (bei positiven Werten von $FootTiltY$) oder rechts (bei negativen Werten) geneigt. Auch hier ist der Wertebereich: $-\frac{\pi}{2} \leq FootTiltX \leq \frac{\pi}{2}$. Der Parameter $FootTiltY$ kann zur Justierung von Unsymmetrien im Roboter-Aufbau genutzt werden, ferner ist es auch möglich, mit großen Werten dieses Parameters den laufenden Roboter zu einem Kurvenlaufen zu bewegen, Details werden in Abschnitt 4.4 beschrieben.

4.6 Das Modul „DortmundWalkingEngine“

Wie in Abschnitt 2.2 schon beschrieben, wurde die hier vorgestellte WalkingEngine sowohl für das Framework *KondoControl* (Abschnitt 2.2.1), als auch für das *BreDoBrothers*-Framework (Abschnitt 2.2.2) implementiert. Da alle weiteren Arbeiten mit der WalkingEngine, insbesondere die Optimierung der Parameter (Abschnitt 6) aber mit dem BreDoBrothers-Framework durchgeführt wurden, wird hier im Speziellen auf das Modul *DortmundWalkingEngine* für das BreDoBrothers-Framework eingegangen.

[9] Für eine Laufart, die eine größere Gewichtsverlagerung voraussetzt, ist zum Teil ein niedrigerer Wert von ZOffset von Vorteil, da der dadurch weiter nach oben verlagerte Roboter-Schwerpunkt als ein verlängertes Pendel funktioniert.

Eine WalkingEngine für das BreDoBrothers-Framework hat bestimme Schnittstellen zu bedienen, anhand deren sichergestellt wird, dass die verschiedenen WalkingEngines sich auch problemlos integrieren lassen. So lassen sich verschiedene WalkingEngines als separate Module implementieren und sogar im laufenden Betrieb gegeneinander austauschen. Die WalkingEngine erhält über diese Schnittstelle Zugriff auf die Gelenkwinkel des Roboters, auf die Art des aktuell geforderten Bewegungszustandes sowie auf Sensor-Daten. Die WalkingEngine wird alle 20 ms automatisch vom Scheduler des Frameworks aufgerufen und darf nur in Aktion treten, wenn ein *WalkRequest*[10] über die Schnittstelle *MotionRequest* anliegt. Über die Schnittstelle *MotionRequest* werden Bewegungen wie z.B. Stillstehen, Aufstehen, Ball schießen, Ball blocken (als Torwart), aber auch Laufen angefordert. Liegt nun ein WalkRequest an, so hat die WalkingEngine die Roboter-Gelenke entsprechend zu bewegen. Ansonsten muß sich die WalkingEngine passiv verhalten.

Mit dem WalkRequest werden noch zwei Kenndaten für die Bewegung übergeben: die geforderte Translation, sowie die gewünschte Rotation. Bei der Translation werden die geforderten Geschwindigkeiten in X- und Y-Richtung[11] jeweils in [mm/s], bei der Rotation hingegen die Winkelgeschwindigkeit in $[°/s]$ angegeben. Der weiter oben mehrfach erwähnte Translationswinkel α wird aus den Rotationsgeschwindigkeiten errechnet als:

$$\alpha = \arctan\left(\frac{Translation_x}{Translation_y}\right)$$

Die Laufgeschwindigkeit in Richtung des Translationswinkels ist gleich der Länge des aus $Translation_x$ und $Translation_y$ aufgespannten Vektors, also:

$$v_{abs} = \sqrt{Translation_x^2 + Translation_y^2}$$

Mit dem Wert von α wird nun wie in Abschnitt 4.4 die Fußtrajektorie gedreht; mit v_{abs} (in [mmm/s]) wird die geforderte Geschwindigkeit in Laufrichtung α bestimmt. Die resultierende Laufgeschwindigkeit ist allerdings schwierig zu bestimmen. Rein rechnerisch ist die Laufgeschwindigkeit :

$$y_{abs} = \frac{1000 \cdot StepLength}{StepDuration}$$

Leider wird in der Praxis diese errechnete Geschwindigkeit nie erreicht. Der Roboter kann auf dem Boden leicht rutschen und dadurch schlecht von der Stelle kommen, oder er kann am Boden hängenbleiben, so dass er gebremst wird. Die Servo-Motoren können ggf. die Gelenke nicht wie gefordert bewegen, weil die dafür benötigten Kräfte (z.B. durch das Gewicht des Roboters, das durch einzelne Servos zu großem Teil getragen werden muss) von den Servos nicht aufgewendet werden können. Zudem hängt die Abweichung von der errechneten Geschwindigkeit stark vom Laufparameter-Satz und der dadurch festgelegten Art des Laufens ab. So ist es für ein ordnungsgemäßes Funktionieren der WalkingEngine notwendig, die Laufparameter-Sätze für die verschiedenen Geschwindigkeiten und die verschiedenen Translationswinkel alle mit dem realen Roboter bezüglich der Laufgeschwindigkeit zu vermessen. Diese Werte müssen

[10] Ein *WalkRequest* ist eine Anforderung, mit einem Laufen zu beginnen. Dieser Request wird ggf. nicht sofort ausgeführt, z.B. wenn andere Bewegungen des Roboters noch nicht abgeschlossen sind.

[11] Bezogen auf das körpereigene Koordinatensystem des Roboters, siehe Abschnitt 2.1.4.

mit dem Laufparametersatz gespeichert werden, so dass für ein vorgegebenes Paar aus Geschwindigkeit und Translationswinkel ein passender Parametersatz ausgewählt werden kann. Diese Funktion wird dann von der WalkingEngine automatisch ausgeführt.

Ferner informiert die WalkingEngine über den Zustand des Laufens, indem über einen eigenen Methoden-Aufruf ausgegeben werden kann, ob die WalkingEngine derzeit verlassen werden kann. Dies ist erforderlich, weil eine Laufbewegung nicht jederzeit beliebig unterbrochen werden kann, weil der Roboter sonst stürzen könnte. In der DortmundWalkingEngine ist dies folgendermaßen implementiert: wenn der Roboter eine aktuelle Geschwindigkeit von unter 10 mm/s hat, so darf die WalkingEngine unterbrochen werden. Dies erfordert, dass vor einem Deaktivieren der WalkingEngine per WalkRequest die Geschwindigkeit auf unter 10 mm/s gesenkt werden muss. Ferner muss der Roboter durch ein stufenweises Herunterfahren der Schrittlänge (siehe Abschnitt 4.3) auch diese Geschwindigkeit erreicht haben.

5 Optimierverfahren

In diesem Kapitel werden die Verfahren vorgestellt, die in dieser Arbeit zur Optimierung der WalkingEngine-Parameter benutzt werden. Dabei liegt der Schwerpunkt bei den Evolutionären Algorithmen, die sich besonders bewährt haben bei Problemen, deren Struktur nicht bekannt oder nicht verstanden ist. Hier werden aus der mittlerweile sehr großen Menge der Evolutionären Algorithmen die Evolutionsstrategien benutzt. Evolutionsstrategien haben ihre Wurzeln in der Optimierung von technischen Problemen und operieren typischerweise auf dem $\mathbb{R}^n$. Nach der Beschreibung der Optimierverfahren folgt eine Erläuterung der Testfunktionen, die hier zur Verifikation der korrekten Algorithmen-Implementierung benutzt wurden.

5.1 Übersicht

Von Optimierung wird gesprochen, wenn ein System, egal aus welchem Umfeld (z.B. Mathematik, Technik, Medizin, Organisation, oder auch sogar Politik), hinsichtlich eines oder mehrerer ausgewählter Kriterien „verbessert“ werden soll. Um eine „Verbesserung“, deren Gestalt vom jeweiligen subjektiven Standpunkt abhängt, überhaupt erreichen zu können, muss es Alternativen geben. Das zu optimierende System muss in irgendeiner Weise so verändert werden können, dass sich das Gütekriterium ändert, und „bessere“ von „schlechteren“ Alternativen unterschieden werden können. Dabei kann eine Verbesserung entweder als Maximierung oder als Minimierung einer Größe definiert sein, wobei diese Größe zumindest einer partiellen Ordnung unterworfen sein muss.

In der hier betrachteten Weise der Optimierung wird von den veränderlichen Größen eines solchen Systems als „(Objekt-)Parameter“ oder „Parametersatz“, im Umfeld der Evolutionären Algorithmen aufgrund deren Wurzeln in der Biologie auch „Individuen“ gesprochen. Der Bewertungsmaßstab ist die „Güte“ oder „Fitness“ (bei den Evolutionären Algorithmen). Wenn eine mathematische Beschreibung des Systems vorliegt, wird von der „Zielfunktion“, bei Evolutionären Algorithmen auch von der „Fitnessfunktion“ gesprochen.

Die hier zu optimierenden Größen sind die in Abschnitt 4.5 beschriebenen Parameter der WalkingEngine. Der zur Optimierung benutzte Parametersatz besteht aus elf der insgesamt 15 Parameter. Die vier nicht benutzten Parameter sind die Phasen-Verschiebungen, die für die WalkingEngine aufgrund von Symmetrieen und Abhängigkeiten konstant gehalten werden. Wir haben es also hier mit einem Optimierproblem im elfdimensionalen Suchraum zu tun. Da alle Parameter aus reellen Zahlen bestehen, entspricht der Suchraum einem Ausschnitt aus dem $\mathbb{R}^{11}$, da für alle Parameter vorgegebene Grenzen, d.h. Minimal- und Maximalwerte existieren. Diese Wertegrenzen sind in Abschnitt 4.5 aufgeführt und dürfen vom Optimierverfahren nicht verletzt werden.

Das Gütekriterium für unser Lauf-Optimierproblem ist hier die Geschwindigkeit der resultierenden Laufbewegung. Wie bereits in Abschnitt 3.2.2 angedeutet, wird hier kein weiteres

Kriterium benutzt, um schnell brauchbare und einfach zu untersuchende Resultate zu erzielen. Für zukünftige Arbeiten auf diesem Gebiet wäre aber auch die Einbeziehung weiterer Kriterien, wie die „Glattheit“ der Laufbewegung, die Abweichung von der vorgegebenen Laufrichtung oder andere Gütemaße sinnvoll. Die Geschwindigkeit wird hier maximiert, der niedrigste Wert entspricht der 0, einen Maximalwert gibt es zumindest prinzipiell nicht. Da die hier beschriebenen Optimierverfahren allesamt auch mit negativen Fitnesswerten arbeiten können, werden für bestimmte Situationen auch negative Geschwindigkeitswerte benutzt, dies dient jedoch lediglich dem einfacheren Umgang mit z.B. einem Umfallen des Roboters oder auch der Markierung noch nicht evaluierter Parametersätze.

5.2 Auswahl eines Optimierverfahrens

Nun ist die Auswahl eines Optimierverfahrens angesichts deren großer Anzahl nicht leicht. Jedoch kann diese Auswahl durch die Eigenheiten des hier vorliegenden Optimier-Problems eingeschränkt werden. Wir haben mit der Optimierung der Laufparameter ein Problem, welches wir nicht mathematisch beschreiben können. Die einzige Möglichkeit zur Handhabung ist es, für einzelne Parametersätze die Güte zu messen. Ferner handelt es sich bei der Messung dieser Güte um eine verrauschte Messung, wie Abschnitt 7.1 erläutert. Schlussendlich ist die Funktion der einzelnen Parameter, welche es zu optimieren gilt, nicht vollkommen klar. So ist bislang unbekannt, ob z.B. die Bewegung der Arme überhaupt notwendig ist, und wie sie mit den anderen Parametern wechselwirkt. Das gleiche gilt z.B. für die Bewegung des Oberkörpers in die X-Richtung oder für das Absenken des Oberkörpers.

Diese Voraussetzungen geben nun einen schon recht engen Rahmen für die Auswahl eines Optimierverfahren. So betrachten wir zunächst die fehlende Problemformulierung. Die Optimierung solcher Systeme wird als „Black Box-Optimierung“ bezeichnet. Eine „Black Box“ ist ein System, welches eine Eingabe in Form der zu optimierenden Parameter akzeptiert und eine Ausgabe in Form des Zielfunktionswertes, also des Gütekriteriums bereitstellt. Die Vorgänge im Innern der Black Box werden nicht nach außen veröffentlicht und interessieren für das Optimier-Verfahren nicht.

Da keine mathematische Beschreibung des Systems vorliegt, können analytische Verfahren nicht benutzt werden. Analytische Verfahren nutzen bestimmte Eigenschaften der Zielfunktion an der Position des Optimums aus, wie z.B. Steigung des Graphs an dieser Stelle. Dazu werden jedoch neben dem Funktionsterm weiterhin Ableitungen benötigt, was in vielen Fällen nicht möglich ist. Somit scheiden diese direkten Verfahren hier aus.

Es gibt ferner die so genannten „iterativen Verfahren“, welche eine schrittweise Näherung an das gesuchte Optimum berechnen, der aktuelle Zielfunktionswert wird in jedem Schritt verbessert. Falls die Güte des Funktionswertes ausreicht, kann auch vor Erreichen des Optimums abgebrochen werden. Dabei muss betont werden, dass die exakte Position des Optimums jedoch normalerweise von den iterativen Verfahren nicht ermittelt werden kann, dies bleibt den analytischen Verfahren vorbehalten. Bei den unterschiedlichen Arten von iterativen Verfahren gibt es auch einige, die neben den reinen Zielfunktionswerten an den zu testenden Suchpunkten zusätzlich die erste oder auch zweite Ableitung der Zielfunktion an dem Suchpunkt nutzen. Hier seien exemplarisch die Gradienten-Strategien genannt, welche die ersten Ableitungen der

Zielfunktion benutzen, sowie die Strategie der konjugierten Richtungen, welche zudem die zweiten Ableitungen verwenden. Für eine weitere Auflistung sei auf Schwefel [Sch95] verwiesen. Da hier aber auch keine Ableitungen angegeben werden können, da eine mathematische Formulierung des Problems nicht vorliegt, können diese Verfahren hier ebenfalls nicht benutzt werden.

Allerdings existieren immer noch sehr viele Verfahren, die der Black-Box-Optimierung genügen. Als einfachste sind die so genannten „Hill-Climber“ zu nennen, die bei der Suche im Parameter-Raum einen Weg suchen, auf dem die Funktionswerte steigen (bei einer Maximierung der Zielfunktionswerte). Das Problem dieser Verfahren ist, dass sie in lokalen Maxima stecken bleiben, also bei Punkten des Suchraums, deren Nachbar-Punkte allesamt schlechtere Funktionswerte haben. Auf eine Aufzählung von Hill-Climbing-Verfahren sei hier verzichtet, auch hier gibt Schwefel [Sch95] eine Übersicht. Zudem gehen diese Verfahren meist von korrekten Funktionswerten aus, was hier durch Messfehler bei der experimentellen Messung der Laufgeschwindigkeit nicht sichergestellt werden kann.

So bieten sich hier nun Heuristiken an, die von der Natur inspiriert sind. Als bedeutendste Klasse seien hier die Evolutionären Algorithmen genannt, welche einer Black-Box-Optimierung genügen, in bestimmten Varianten mit verrauschten Größen umgehen können und in wiederum speziellen Varianten auch von selbst auf die Struktur der Parameter eingehen können. Andere Alternativen wären z.B. Particle Swarm Algorithmen, welche das Verhalten von Schwärmen (wie etwa Fische oder Vögel) nachahmen, um zu optimieren. Ein Particle Swarm Algorithmus wurde zu Beginn dieser Arbeit auch implementiert und getestet, jedoch sorgte die Auswahl der verschiedenen Parameter, welche das Schwarm-Verhalten beeinflussen, für geringe Erfolge. Da Versuche sowohl im BreDoBrothers-Simulator, als auch mit dem realen Roboter extrem zeitaufwändig sind, wurde hier verzichtet, aufwändige Experimente zur Wahl der Particle-Swarm-Parameter durchzuführen.

5.3 Evolutionäre Algorithmen

Bei den „Evolutionären Algorithmen“ handelt es sich um randomisierte Heuristiken, welche ihre Wurzeln in der Entwicklung des Lebens in der Natur hat. So werden die Phänomene der Vererbung, der Rekombination, der Mutation von Erbgut sowie das wahrscheinlichere Überleben angepassterer Spezies (bekannt geworden als „survival of the fittest“ nach Darwin) ausgenutzt, um Optimierungen in der Technik und der Wissenschaft durchzuführen. Entwickelt wurden diese Verfahren als erstes in den sechziger Jahren des 20. Jahrhunderts und zwar unabhängig voneinander in Deutschland und den USA. Die Pioniere waren Schwefel [Sch75] und Rechenberg [Rec73] in Deutschland mit den „Evolutionsstrategien“, Holland [Hol75] in den USA mit den „Genetischen Algorithmen“ sowie Fogel [FOW66] ebenfalls in den USA mit dem „Evolutionären Programmieren“.

Bei den Evolutionären Algorithmen werden je nach Typ die Parametersätze aus dem Suchraum des Optimierproblems, auch „Individuen“ genannt, miteinander „rekombiniert“ oder „mutiert“. Bei der Rekombination werden mindestens zwei Individuen vermischt, wobei es dort unterschiedliche Varianten gibt. So können die einzelnen Objektparameter des aus der Rekombination entstehenden Individuums z.B. zufällig von den verschiedenen Eltern gewählt werden,

es kann für jeden einzelnen Parameter des Individuums der Durchschnitt der entsprechenden Elternparameter gewählt werden, oder ähnliches. Bei der Mutation werden die einzelnen Parameter basierend auf Zufallswerten mit bestimmten Verteilungen variiert. Abbildung 5.1 zeigt den Pseudo-Code eines Evolutionären Algorithmus.

```
BEGIN
  Setze Generationszähler t := 0;
  Initialisiere eine Population von Suchpunkten;
  Bewerte diese Population anhand des Fitnesskriteriums;
  wiederhole
    Erzeuge Nachkommen durch Rekombination aus der Population;
    Mutiere die Nachkommen;
    Bewerte die die Nachkommen anhand des Fitnesskriteriums;
    Selektiere Individuen zur Ersetzung;
    Setze t:= t+1;
  bis Abbruchbedingung erfüllt (wie z.B. ausreichende
  Fitness, Generationen-Limit überschritten, u.v.m.);
END
```

Abbildung 5.1: Der Pseudo-Code eines Evolutionären Algorithmus.

Die Durchführung der Optimierung erfolgt in „Generationen“, in denen auf der „Population“, also die Menge der jeweils aktuellen Individuen, momentan operiert wird. Die Population kann dabei sowohl aus nur einem einzigen, als auch aus mehreren Individuen bestehen; die Größe der Population wird mit μ bezeichnet. In jeder Generation werden λ (auch $\lambda = 1$ ist dabei möglich) Nachkommen, die „Offsprings“ aus der Population (in diesem Fall dann auch als „Eltern“ oder „Parents“ bezeichnet) erzeugt. Dies geschieht mittels der erwähnten Rekombination und/oder Mutation. Für die Rekombination werden die Eltern entweder abhängig von ihrer Fitness und/oder zufällig gewählt.

Die Idee hinter der Rekombination und der Mutation ist es, neue Suchpunkte auszuwählen. So ist die grundsätzliche Annahme bei den Evolutionären Algorithmen, dass die Topologie des Suchraumes einer Form entspricht, bei der „gute“ Funktionswerte in der Nähe anderer guter Werte liegen. Der Rekombinations-Operator sorgt nun vor allem dafür, bessere Individuen aus guten Teilen der Eltern zu erzeugen, es werden neue Regionen im Suchraum erreicht. Die Mutation dagegen dient eher dem Ziel leichte Veränderungen zu erzeugen, wobei diese Änderungen nicht zu groß sein sollen, damit in der Nähe von bereits gefundenen guten Punkten noch bessere Punkte gefunden werden.

Da bei den Evolutionären Algorithmen in der Regel konstante Populations-Größen gefordert sind, muss am Ende einer Generation bei der „Selektion“ ausgewählt werden, welche Individuen in die neue Generation übernommen werden. Dies kann anhand der Fitness geschehen, oder es werden deterministisch alle erzeugten Nachkommen zu den Eltern der nächsten Generation. Wird die Eltern-Generation nicht in die nächste Generation übernommen, so wird von einer „Komma-Selektion“ gesprochen. Werden die besten Individuen aus den Eltern und aus den Nachkommen zusammen zur Übernahme in die nächste Generation ausgewählt, so wird dies als „Plus-Selektion“ bezeichnet. Ferner kann für die Plus-Selektion eine maximale Lebensdauer

von κ Generationen für die Eltern vorgegeben werden.

Dabei wird die Selektion zur Rekombination bei den Genetischen Algorithmen (siehe Abschnitt 5.3.1) anhand der Fitnesswerte vorgenommen, bei den Evolutionsstrategien (siehe Abschnitt 5.3.2) dagegen, sofern überhaupt Rekombination durchgeführt wird, zufällig. Die Mutation hat bei den Evolutionsstrategien einen höheren Stellenwert als bei den Genetischen Algorithmen, dort wird die Rekombination als der wichtigere Variationsoperator angesehen, siehe [BS93]. Bei den Evolutionsstrategien gibt es ferner die Möglichkeit, die Stärke der Mutationen zu ändern. So gibt es unter anderem das Verfahren der „Selbstadaptation“, bei dem der Algorithmus die erforderlichen Mutationsstärken selbst ermittelt. Da hier beim Problem der Lauf-Optimierung kein Wissen über die genauen Auswirkungen der Parametereinstellungen vorhanden ist, werden im weiteren Verlauf Evolutionsstrategien benutzt, da diese sich mit der Selbstadaptation eigenständig einstellen.

Das Vorbild der Evolution in der Natur wird auch benutzt beim „Genetischen Programmieren“, welches von Koza [Koz92] in den neunziger Jahren des 20. Jahrhunderts entwickelt wurde. Dabei geht es jedoch nicht um die Optimierung von Parameter-Einstellungen, sondern um die Entwicklung von Programmen mittels Mechanismen der Evolution. Es werden dabei entweder Baumstrukturen oder lineare Listen aus Programmcode entwickelt, dabei werden diese Programme aus elementaren Bestandteilen, wie Operationen oder Konstanten zusammengesetzt. Dieses Verfahren kann prinzipiell auch für die Entwicklung einer Laufsteuerung für Roboter benutzt werden (wie etwa von Ziegler [Zie03] durchgeführt).

Der Einsatz von Genetischer Programmierung wurde auch für diese Arbeit erwogen. Jedoch wurde wegen des anfangs noch nicht vorhandenen Simulators und der damit noch nicht vorhandenen Möglichkeit, viele Evaluationen im Rechner stattfinden zu lassen, ein Ansatz gewählt, der möglichst viel Vorwissen in die Laufsteuerung mit einbringt. Bei der Genetischen Programmierung wird dagegen üblicherweise kein Vorwissen benutzt, so dass es unter Umständen sehr lange dauert, bis sich Programme entwickelt haben, die zu einem Laufverhalten führen. Dies ist natürlich ungünstig, wenn nur der reale Roboter benutzt wird. So ist es in diesem Fall sinnvoller, mit möglichst viel Vorwissen über das gewünschte Laufverhalten in die Optimierung zu starten, wie dies in dieser Arbeit mit der Vorgabe der Trajektorien getan wurde. So ist ein grundsätzliches Laufen bereits zu Beginn der Optimierung vorhanden, jenes muss nur noch verbessert werden.

5.3.1 Genetische Algorithmen

Genetische Algorithmen, entwickelt von Holland [Hol75], operieren in der ursprünglichen Version auf dem $\mathbb{B}^n$, also dem n-dimensionalen Suchraum der Binärzahlen, welcher auch als $\{0,1\}^n$ ausgedrückt werden kann. Es handelt sich bei den Rekombinations- und Mutations-Operationen also um Bit-Manipulationen. In neueren Fassungen kann mit den Genetischen Algorithmen aber auch im $\mathbb{R}^n$, und sogar auf Permutationen (zum Sortieren) operiert werden. Die Fitness der einzelnen Individuen ist typischerweise die Menge der reellen Zahlen $\mathbb{R}$. Bei den Genetischen Algorithmen wird eine Populationsgröße von mehr als eins gewählt, damit Rekombination möglich ist. Gemäß der Ideologie der Genetische Algorithmen ist die Rekombination der wichtigste Variations-Operator, Mutationen werden als weniger bedeutend angesehen, aber auch benutzt (wenn auch mit meist geringerer Wahrscheinlichkeit).

Die „Codierung der Chromosomen“ bildet den Suchraum des gegebenen Optimier-Problems auf den bei den Genetischen Algorithmen benutzten Suchraum $\mathbb{B}^n$ ab. Diese Codierung zu finden ist eines der Hauptprobleme bei den Genetischen Algorithmen. Durch die Einführung dieser Codierung ist es allerdings auch erst möglich, einen stets gleichen Algorithmus auf die unterschiedlichsten Probleme anzuwenden. Üblicherweise werden die zu rekombinierenden Individuen zufällig, allerdings abhängig von deren Fitness ausgewählt. So wird erreicht, dass sich fittere Individuen eher fortpflanzen. Allerdings haben auch Individuen mit geringerer Fitness eine Chance, sich fortzupflanzen. Dadurch kann ein Ausweg aus lokalen Optima gefunden werden. Die verwendete „Selektion zur Ersetzung“ bei den Genetischen Algorithmen ist üblicherweise die Komma-Selektion. Die Eltern werden dabei, unabhängig von ihrer Fitness, verworfen. Obwohl im Prinzip alle Nachkommen schlechter sein können, als die Eltern, wird davon ausgegangen, dass sich die Fitness der Populationen mit der Zeit trotzdem verbessert, da bei der Selektion zur Reproduktion bessere Eltern bevorzugt ausgewählt werden.

Die Rekombination (das Crossover) kann aus verschiedenen Methoden gewählt werden. Häufig wird das 1-Punkt-Crossover benutzt, bei dem ein Kreuzungspunkt innerhalb der $\{1..n-1\}$ Stellen zwischen den Bits des Individuums zufällig gewählt wird. Die Bits bis zum Kreuzungspunkt kommen von Elter 1, die restlichen werden von Elter 2 übernommen. Gibt es mehrere Kreuzungspunkte, so wird von „k-Punkt-Crossover“ gesprochen, die einzelnen Abschnitte des Bitstrings werden alternierend von beiden Eltern gewählt. Ferner können die Bits aus mehreren Eltern gewählt werden, dann wird zufällig gewählt, proportional zur Gesamthäufigkeit der einzelnen Bits in der Menge der beteiligten Eltern. Die Mutation erfolgt nicht immer, sondern mit einer vorher festzulegenden Wahrscheinlichkeit.

In dieser Arbeit werden Genetische Algorithmen nicht verwendet, da die Wahl der externen Parameter (Crossoverwahrscheinlichkeit, Mutationswahrscheinlichkeit, Mutationsstärke) nicht klar ist. Genetische Algorithmen führen keine „Selbstadaptation“ durch, wie es die fortgeschritteneren Evolutionsstrategien tun. Zudem sind die Evolutionsstrategien von ihrer Entwicklung her für reellwertige Probleme aus dem experimentellen Umfeld prädestiniert.

5.3.2 Evolutionsstrategien

Evolutionsstrategien (abgekürzt „ES“) wurden erstmals Anfang der 60-er Jahre des 20. Jahrhunderts von Schwefel [Sch75] und Rechenberg [Rec73] entwickelt und benutzt, um praktische technische Probleme zu optimieren. Dabei wurden konkret eine variable Tragfläche in einem Windkanal, sowie die Form eines Rohr-Winkelstücks für den Flüssigkeitstransport optimiert. Die erste Evolutionsstrategie benutzte eine Population von einem Individuum, der einzige Nachkomme wurde jeweils pro Generation alleine durch Mutationen erzeugt. Bei besserer Fitness wurde das Elter durch den Nachkommen ersetzt, es handelte sich also um eine Plus-Selektion. Sämtliche Objektparameter des Optimierproblems waren aufgrund der Struktur aus dem Bereich der reellen Zahlen, von daher wurden die Evolutionsstrategien für den $\mathbb{R}^n$ entwickelt. Die Mutationen pro Parameter wurden normalverteilt erzeugt, so dass kleinere Mutationen wahrscheinlicher als große waren, was genau der Philosophie der Evolutionären Algorithmen entspricht, dass gute Lösungen in der Nähe anderer guter Lösungen zu finden sind.

Rechenberg untersuchte die (1+1)-Evolutionsstrategie 1973 in seiner Dissertation [Rec73] und stellte die 1/5-Erfolgsregel vor. Diese Heuristik wird benutzt, um die Stärke der Mutationen zu steuern: wenn zu viele Erfolge (Verbesserungen der Fitness) im Verlauf der Generationen vorkommen, kann davon ausgegangen werden, dass das Optimum noch weit entfernt ist. Die Mutationsschrittweite wird dann angehoben. Bei zu wenigen Erfolgen wird davon ausgegangen, dass man aufgrund zu hoher Mutationsstärken ständig am Optimum „vorbeispringt“, und verringert die Mutationsschrittweiten. Rechenberg bewies an zwei Modellen, dass eine Erfolgsrate von 1/5 für diese Modelle optimal ist. Diese Art der (1+1)-Evolutionsstrategien wurde auch an der Optimierung der Laufparameter für die WalkingEngine getestet, die Ergebnisse werden in Kapitel 7 präsentiert.

Schwefel entwickelte in seiner Dissertation [Sch75] die Evolutionsstrategien weiter und stellte die „Mehrgliedrige Evolutionsstrategie“ vor, in der die Population einer Größe $\mu \geq 1$ und Nachkommen-Mengen mit einer Größe $\lambda \geq 1$ benutzt werden. Ferner stellte Schwefel in seiner Arbeit eine neue Form der Mutations-Schrittweitensteuerung vor: es wird nicht eine Schrittweite für alle Objektparameter gemeinsam wie bei der 1/5-Erfolgsregel benutzt. Auch wird nicht deterministisch bei der Auswahl der Schrittweiten vorgegangen. Schwefel führt für jeden zu optimierenden Objektparameter einen „Strategieparameter“ ein, der die Mutationsschrittweite bestimmt. Diese Strategieparameter werden nun auch der Evolution unterworfen. Die Ideologie ist, dass gute Strategieparameter zu guten Objektparametern führen, der Algorithmus stellt sich selbst auf die Topologie der zu optimierenden Black-Box-Funktion ein. Dieser Vorgang wird als „Selbstadaptation“ bezeichnet. Selbstadaptive Evolutionsstrategien werden in dieser Arbeit auch benutzt, weiter unten werden eine $(1, \lambda)$ES, eine (μ, λ)ES und eine $(\mu + \lambda)$ES beschrieben. Bei letzterer wird eine maximale Lebensdauer der Eltern vorgegeben.

So überlebt ein Elter nur maximal κ Generationen, falls es nicht aufgrund unzureichender Fitness nicht schon vorher durch einen Nachkommen ersetzt wurde. Dies wird durchgeführt, da sich reine „Plus“-Strategien bei der Selbstadaptation in Sachen Konvergenzgeschwindigkeit eher schlecht verhalten. So kann es Eltern geben, die zufälligerweise eine hohe Fitness haben, aber deren Strategieparameter nicht gut eingestellt sind. Wenn diese Eltern nun lediglich nach ihrer Fitness beurteilt werden, so geben sie ihre schlechten Strategie-Parameter an die Nachkommen weiter, die deswegen nur schwer bessere Objektparameter erzeugen und somit auch nur schwer eine bessere Fitness erreichen können. Durch die „Komma“-Selektion jedoch oder auch durch die Maximal-Lebensdauer κ wird erreicht, dass Eltern zwangsweise ersetzt werden und neben den eigentlichen Parameterwerten auch neue Strategie-Parameter benutzt werden. Ferner erleichtert die Komma-Selektion, bzw. die Plus-Selektion mit Lebensdauer κ das Entkommen aus lokalen Optima, weil Verschlechterungen ermöglicht werden.

Die Rekombination kann entweder „intermediär“ oder „diskret“ erfolgen. Bei der intermediären Rekombination wird für den Nachkommen pro Parameter der Durchschnitt der entsprechenden Werte der Eltern gewählt. Bei der diskreten Rekombination werden die einzelnen Parameter zufällig gleichverteilt und unabhängig von einem der Eltern übernommen. Die Auswahl der Eltern für die Rekombination erfolgt üblicherweise zufällig gleichverteilt, und nicht basierend auf der Fitness, wie es bei den Genetischen Algorithmen gehandhabt wird. Die Mutation erfolgt, wie bereits erwähnt, normalverteilt mit Erwartungswert 0. Die Varianz der Verteilung entspricht der Mutationsschrittweite, welche je nach Typ der Evolutionsstrategie unterschiedlich bestimmt wird.

Das Ersetzungsschema ist fitnessbasiert, es werden bei der Plus-Selektion aus allen Individuen, also Eltern und Nachkommen gemeinsam, die μ besten Individuen gewählt. Bei der Komma-Selektion werden nur aus den λ Nachkommen die μ besten gewählt. Damit die Optimierung nicht zu einer zufälligen Suche verkommt, muss demnach bei der Komma-Selektion zwingend $\lambda > \mu$ gelten. Für eine detaillierte Beschreibung der Evolutionsstrategien sei auf Schwefel [Sch95] verwiesen.

Es folgen nun die in dieser Arbeit bezüglich des Optimierproblems der WalkingEngine-Parameter implementierten und untersuchten Algorithmen. Wie schon erwähnt, handelt es sich ausschließlich um Evolutionsstrategien, da sich diese aufgrund des $\mathbb{R}^n$ als Suchraum anbieten und sich vor allem im Bereich der experimentellen Optimierung als erfolgreich erwiesen haben.

5.3.2.1 $(1+1)$-ES mit 1/5-Erfolgsregel

Die (1+1)-Evolutionsstrategie ist die einfachste Variante der Evolutionsstrategien, dies war trotz ihrer im Weiteren beschriebenen Schwächen der Grund, warum sie hier als erstes implementiert wurde. Die betrachtete Population der Punkte im Suchraum besteht aus nur einem Individuum, die Nachkommen werden lediglich durch Mutationen des Elter-Individuums erzeugt, wobei es in jeder Generation nur einen Nachkommen gibt. Wenn die Fitness des Nachkommen größer als die des Elter ist, so wird der Nachkomme in der nächsten Generation zum neuen Elter. Zur Erzeugung des Nachkommen wird zu jeder einzelnen der insgesamt n Parameter des Elter-Individuums eine normalverteilte Zufallszahl addiert. Die Verteilung dieser Zufallszahl entspricht der Normalverteilung um den Erwartungswert 0 mit der aktuellen Mutationsschrittweite als Varianz. Der Wert des jeweiligen Nachkommen-Objektparameters y_i mit $i \in \{1..n\}$ wird aus den Werten der Elter-Objektparameter erzeugt durch Mutation als $y_i^{(t)} = x_i^{(t)} + \sigma^{(t)} \cdot N_i(0,1)$. Diese Zufallszahlen $N_i(0,1)$ werden für jeden der n Parameter unabhängig voneinander erzeugt, dabei ist $N_i(0,1)$ die normalverteilte Zufallszahl um Erwartungswert 0 mit der Varianz 1; $\sigma^{(t)}$ ist die aktuelle Mutationsschrittweite.

Alle $10 \cdot n$ Generationen wird der Anteil der Erfolge (von Erfolg wird gesprochen, wenn der Nachkomme eine bessere Fitness hat als das Eltern-Individuum) errechnet. Gab es in den letzten $10 \cdot n$ Generationen mehr als 20% Erfolg, so wird die Mutationschrittweite σ durch $0,85$ geteilt, also erhöht. Lag der Erfolg niedriger als 1/5, so wird σ durch Multiplikation mit $0,85$ verringert. Diese als „1/5-Erfolgsregel“ bezeichnete Anpassung der Mutationsschrittweite erfolgt, um zu Anfang der Optimierung mit großen Schritten in Richtung Optimum zu gelangen, und im späteren Verlauf mit kleinen Schritten dem Optimum sehr nahe zu kommen. Allerdings gilt die Mutationsschrittweite σ dabei für alle Parameter des Optimier-Problems gemeinsam, so wird zunächst nicht davon ausgegangen, dass die Zielfunkstions-Topologie für die einzelnen Parameter unterschiedliche Maßstäbe verlangt.

Beim Problem der Optimierung der WalkingEngine-Parameter gibt es jedoch zum Teil sehr unterschiedliche Maßstäbe in den Parameterwerten. So bewegen sich die Amplituden der Trajektorien für die Arm- und Körper-Bewegungen z.B. im Wertebereich $[0;1]$, dagegen liegt der Wert für die Zeit eines Schrittes („StepDuration“) zwischen 700 und 3000. Für die Amplituden ist nun eine Mutationsschrittweite von mehr als eins nicht zu empfehlen, da diese meist aus dem erlaubten Wertebereich herausführen würde. Dagegen wäre für StepDuration

auch eine Schrittweite von weit über 100 kein Problem. Um dieser Ungleichheit zu begegnen, wurde die gemeinsame Mutationsschrittweite σ bei der (1+1)-ES für jeden einzelnen Parameter der WalkingEngine skaliert. Der Skalierungsfaktor errechnete sich aus der Differenz der Wertebereichs-Grenzen des jeweiligen Parameters. Bei den erwähnten Amplituden handelt es sich also um den Faktor 1; bei der „StepDuration“ wird der Wert 2300 benutzt. So wird nun jeder Parameter gemäß seines Wertebereiches skaliert; die Verwendung einer gemeinsamen Mutationsschrittweite erzeugt ein ähnliches Verhalten auf allen Parametern. Die initiale Mutationsschrittweite wird zum Start der Optimierung auf den Wert $0,1$ gesetzt, so dass beispielsweise die Amplituden anfangs mit einer Varianz von $0,1$ verändert werden; die StepDuration wirde mit Varianz 230 geändert.

Wird der Wertebereich der Parameter verlassen[1], so wird die Fitness des Individuums sofort auf 0 gesetzt, ohne überhaupt zu versuchen, die Laufgeschwindigkeit mit diesem Individuum als Parametersatz zu ermitteln. Dies wäre z.B. im Fall von negativen Werten für die Amplituden auch nicht möglich, oder für zu niedrige Werte von StepDuration potentiell schädlich für den Roboter. Eine solche Verletzung der Wertegrenzen wird auch als Misserfolg gewertet und wirkt sich auf die Steuerung der Mutationsschrittweite aus. Ferner wird auch ein Umfallen des Roboters beim Laufen mit Fitness 0 „bestraft“.

Der entscheidende Nachteil der (1+1)-ES ist neben der mit Größe eins minimalst kleinen Population und der damit einhergehenden fehlenden Diversität auch das Problem, dass falsche Messwerte sich aufgrund der Plus-Selektion sehr lange in der Population halten können. Gerade bei experimentellen Optimierungen, bei denen die Fitnesswerte erst gemessen werden müssen, sind falsche Fitnesswerte aufgrund von Messfehlern möglich. Wird nun die Geschwindigkeit eines Parametersatzes als irrtümlich zu hoch gemessen (im weiteren wird von der „scheinbaren“ Fitness gesprochen), so wird sich das Individuum aufgrund der Plus-Selektion in der Population halten, obwohl der „echte“ Fitnesswert (weiterhin mit „tatsächlicher“ Fitness bezeichnet) eigentlich viel niedriger ist. Die Nachkommen werden, sofern sie nicht auch durch Messfehler als zu gut abschneiden, eine Fitness im Bereich der tatsächlichen Fitness des Elters haben, und somit nicht mit der scheinbaren Fitness konkurrieren können.

Dies führt dazu, dass sich die Evolutionsstrategie lange in einem falschen Bereich des Suchraums aufhalten wird, weil einmal eine falsche Messung durchgeführt wurde. Wenn zufällig ein wirklich besserer Funktionswert gefunden wird, kann die Optimierung wieder erfolgreich weiterverlaufen. Dasselbe trifft zu, wenn wiederum eine falsche Fitness gemessen wird, die das Individuum als zu gut beurteilt. Dann sind wir jedoch genauso in der Situation wie zuvor. Zudem wirkt sich diese Beibehaltung des falschen Messwertes in der Population auf die Steuerung der Mutationsschrittweite aus. Wenn wir noch weit vom Optimum entfernt sind, und durch einen hohen Messfehler ein Individuum in der Population haben, welches sich nun kaum noch verdrängen lässt, so erreichen wir keine weiteren Erfolge. Dadurch wird die Mutationsschrittweite ständig weiter abgesenkt, und die Wahrscheinlichkeit, den falschen Messwert durch ein tatsächlich besseres Individuuem zu ersetzen, schwindet stetig. So wird die (1+1)-ES aufgrund von Messfehlern schnell ohne weitere Verbesserungen steckenbleiben.

[1] Die Individuen, deren Parameter die vorgegebenen Grenzen verletzen, werden analog zur Biologie als „letal“, also nicht überlebensfähig bezeichnet.

Um diese Problematik, welche bei der hier verwendeten Messung der Laufgeschwindigkeit auch auftreten könnte, zu umgehen, wurde die Mutationsschrittweitensteuerung testweise verändert: wenn die Mutationsschrittweite unter den Schwellenwert von $0,001$ absank, wurde sie auf den initialen Wert von $0,1$ angehoben. Dadurch ist es potentiell möglich, der „Messfehler-Falle“ zu entgehen, allerdings mit dem entscheidenden Nachteil, dass nun die Mutationsschrittweiten erst wieder umständlich angepasst werden mussten. Ferner funktioniert das Ersetzen des fehlerhaft gemessenen Individuums auch nur, wenn noch tatsächlich bessere Funktionswerte zu erreichen sind. Alles in allem bleibt festzuhalten, dass die (1+1)-ES für diese Optimier-Aufgabe nur eingeschränkt verwendbar ist, dies liegt neben dem Problem mit den verrauschten Messwerten auch an dem Nachteil, dass hier nur eine Mutationsschrittweite für alle Parameter gemeinsam verwaltet wird. Vor allem die Unkenntnis über die benötigten Mutationsschrittweiten und die Sensibilität der einzelnen Parameter aus Variationen spricht eher für ein selbstadaptives Verfahren, welches die Mutationssteuerung einzeln für die Parameter vornimmt.

5.3.2.2 $(1, \lambda)$-ES mit σ-Selbstadaptation

Wie bereits erwähnt, entwickelte Schwefel für die „mehrgliedrige Evolutionsstrategie“ das Verfahren der „Selbstadaptation“, bei dem die Individuen zusätzlich zu den n zu optimierenden Objektparametern noch einen Satz von n „Strategieparametern“ bekommen, welche die Mutationsschrittweiten für jeden einzelnen Objektparameter speichern. Die Idee dahinter ist, dass gute Strategieparameter auch zu guten Individuen führen, wobei die Strategieparameter auch der Evolution durch Rekombination, Mutation und Selektion unterworfen sind.

Die Objektparameter werden wie bei der (1+1)-ES mutiert, mit dem einzigen Unterschied, dass es für jeden einzelnen Objektparameter x_i einen eigenen Strategieparameter σ_i gibt. Somit erfolgt die Erzeugung der Nachkommen y_i durch Mutation der Objektparameter gemäß: $y_i^{(t)} = x_i^{(t)} + \sigma_i^{(t)} \cdot N(0,1)$. Dabei werden vor dieser Mutation die $\sigma_i^{(t)}$ generiert, und zwar mittels:

$$\sigma_i^{(t)} = \sigma_i^{(t-1)} \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)}$$

mit den Konstanten

$$\tau' = \frac{1}{\sqrt{2n}}$$

$$\tau = \frac{1}{\sqrt{\sqrt{2n}}}$$

Es wird hier also multiplikativ mutiert, im Gegensatz zu der additiven Mutation bei den zu optimierenden Parametern x_i. Da hier $\mu = 1$ gilt, entfällt die Rekombination bei diesem Algorithmus. Wichtig ist, dass zunächst die neuen Strategie-Parameter erzeugt werden und erst danach auf die Objektparameter angewandt werden. Nur so kann der Effekt auftreten, dass „gute“ Strategieparameter auch zu „guten“ Objektparametern führen, was ja die Idee hinter der Einführung der Strategieparameter ist. So werden in der Berechnung der $\sigma_i^{(t)}$ auch sowohl eine gemeinsame Zufallszahl $N(0,1)$ für alle Strategieparameter genutzt, als auch für jeden Stratgieparameter eine individuelle Zufallszahl $N_i(0,1)$. Dadurch wird gewährleistet, dass der gesamte Strategieparametersatz des Individuums, abhängig vom Skalierungsfaktor τ' durch eine gemeinsame Mutationsstärke verändert wird, jedoch wird ferner jeder einzelne Strategieparameter zudem gering mit einer individuellen Änderung, in Abhängigkeit von τ variiert.

Die Komma-Strategie stellt sicher, dass das Eltern-Individuum nach jeder Generation wieder verworfen wird, von den λ Nachkommen wird jeweils das beste als Elter der nächsten Generation ausgewählt. Bei letalen Individuen, oder bei Individuen, durch die der Roboter beim Laufen umfällt, wird wiederum der Fitnesswert 0 vergeben. Wird ein letales Individuum erzeugt, so wird hier bei der Optimierung der WalkingEngine-Parameter so lange neu rekombiniert und mutiert, bis alle Parameterwerte innerhalb der erlaubten Grenzen liegen. Diese Vorgehensweise gilt auch für die weiter unten beschriebene (μ, λ)-ES und die $(\mu + \lambda)$-ES.

Die $(1, \lambda)$-ES wurde hier nur der Vollständigkeit halber entwickelt, um zu untersuchen, ob bei dem vorliegenden Optimier-Problem eine Population von einem einzigen Individuum ausreicht, oder ob $\mu > 1$ gewählt werden sollte. Ferner sollte hiermit untersucht werden, wie sich die Adaptation der Schrittweiten verhält. Es wird vermutet, dass eine Evolutionsstrategie mit $\mu > 1$ aufgrund der Diversität der Population und der dort erst möglichen Rekombination vorteilhafter als die $(1, \lambda)$-ES ist.

Die $(1, \lambda)$-ES wurde hier als (1,30)-ES implementiert. Der Wert von $\lambda = 30$ erklärt sich hier anhand der weiter unten beschriebenen (5,30)-ES. Um diese beiden Strategien besser miteinander vergleichen zu können, wurden dieselben Werte für λ gewählt, obwohl man ansonsten für die $(1, \lambda)$-ES kleinere Werte bevorzugen würde.

Da bei Komma-Selektionen die Eltern von den Nachkommen verdrängt werden, können einmal gefundene Punkte im Suchraum verloren gehen. Da hier das Ziel der Optimierung eben das Erreichen solch guter Punkte ist, unabhängig von der Art und der Funktionsweise der benutzten Algorithmen, wird ein Archiv verwaltet, welches das jeweils beste erreichte Individuum zusätzlich zu der sowieso durchgeführten Speicherung aller Individuen (siehe Abschnitt 6) speichert. So kann der beste Wert, an dem wir ja interessiert sind, nicht mehr verloren werden. Für den Fall von stärkeren Messfehlern sollte vor der Verwendung dieses Parametersatzes allerdings eine wiederholte Messung der Laufgeschwindigkeit mit diesem Parametersatzes durchgeführt werden. Die Speicherung des jeweils besten gefundenen Individuums wird auch von den weiter unten vorgestellten (μ, λ) und $(\mu + \lambda)$-ES benutzt, da auch dort die Eltern (spätestens nach κ Generationen) wieder verworfen werden.

5.3.2.3 (μ, λ)-ES mit σ-Selbstadaptation

Bei der (μ, λ)-Evolutionsstrategie kann aufgrund der Populationsgröße Rekombination durchgeführt werden, hier immer mit jeweils zwei Eltern. In diesem Fall handelt es sich dabei um eine diskrete Rekombination der Parameter zur Optimierung und eine intermediäre Rekombination der Strategie-Parameter. Die Mutation verläuft wie bei der $(1, \lambda)$-ES und wird auf die durch Reproduktion erzeugten Nachkommen angewandt. Die Individuen zur Rekombination werden zufällig gleichverteilt und unabhängig voneinander aus der Population gewählt. Bei der diskreten Rekombination der Parameter wird mit der Wahrscheinlichkeit $p_m = 1/2$ unabhängig voneinander für jeden einzelnen der n Parameter entweder Elter 1 oder Elter 2 gewählt. Bei der Selektion zur Ersetzung werden die besten μ Nachkommen die Eltern der nächsten Generation.

Hier wurde eine (5,30)-ES gewählt als Kompromiss aus großer Population und geringem Aufwand pro Generation. Um die Entwicklung der Parameter in Richtung Optimum zu beschleu-

nigen, ist man an kleinen Populationen interessiert, um den Aufwand für die Fitness-Messung pro Generation gering zu halten. Dagegen haben größere Populationen den Vorteil, dass die Wahrscheinlichkeit für bessere Fitnesswerte größer ist. Die Wahl von $\lambda = 30$ wurde aufgrund von Schwefels Empfehlung [Sch87] für $\lambda \approx 7 \cdot \mu$ getroffen.

5.3.2.4 $(\mu + \lambda)$-ES mit σ-Selbstadaptation und Lebensdauer κ

Die reine Komma-Selektion hat nun den Nachteil, dass eventuell gute Individuen nur für eine Generation in der Population bleiben und so ihre Parameterwerte nur an wenige Nachkommen weitergeben können. Dagegen würde natürlich eine Plus-Strategie helfen, jedoch funktioniert diese nur schlecht bei der Selbstadaptation, da Individuen mit guten Fitnesswerten nicht zwangsläufig gute Strategieparameter haben müssen. Weiterhin würde sich ein bereits in frühen Generationen nahe des Optimums entwickeltes Individuum bei der Plus-Selektion aufgrund seines Fitnesswertes lange in der Population halten, aufgrund zu hoher Strategie-Parameter es jedoch nicht seinen Nachkommen ermöglichen können, in der Nähe des Optimums zu bleiben.

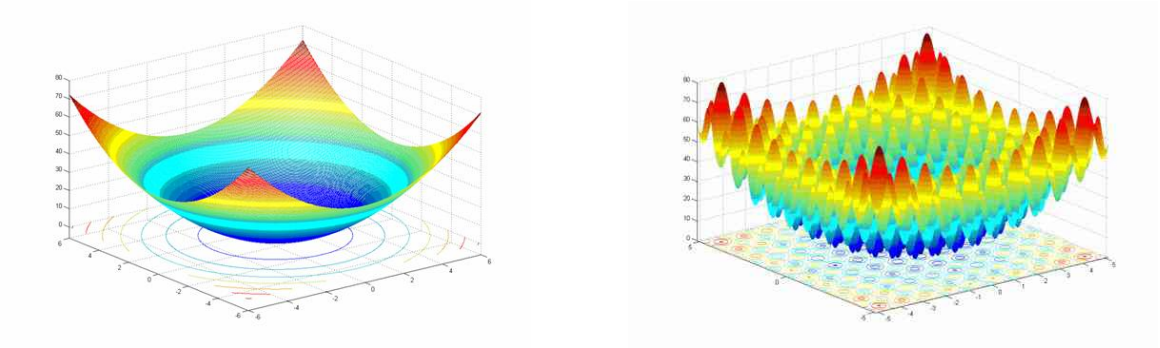
So bietet sich als Ausweg der Kompromiss an, eine Plus-Selektion durchzuführen, jedoch die Lebensdauer der Individuen zu begrenzen. Dies geschieht nun bei der hier vorgestellten $(\mu + \lambda)$-Evolutionsstrategie mit dem Parameter κ, welcher angibt, wieviele Generationen ein Individuum maximal in der Eltern-Population verbleiben darf, falls es nicht schon vorher durch fittere Nachkommen ersetzt wird. Hier wurde $\kappa = 3$ gewählt als Kompromiss zwischen längerer Überlebensdauer guter Parametereinstellungen und der Notwendigkeit der Neubildung der Strategieparameter. Diese Wahl hat sich auch in [HNF07] bei einem ähnlichen Problem als gut herausgestellt. Aufgrund des hohen Zeitbedarfs einzelner Experimente mit den Algorithmen im BreDoBrothers-Simulator wurde an dieser Stelle nicht evaluiert, welche Werte für κ an dieser Stelle empfehlenswert sind. Jedoch ist schon leicht einzusehen, dass sich bei Werten unter drei der Einfluss der längeren Lebensdauer der Individuen kaum bemerkbar macht; bei wesentlich höheren Werten würde dagegen ein eventueller Einfluss falscher Messwerte zu stark werden.

5.4 Testfunktionen

Um die Korrektheit der implementierten Optimier-Algorithmen zu evaluieren, wurden Testfunktionen implementiert und auf die Algorithmen angewandt. Testfunktionen sind standardisierte Probleme, deren Lösungen bekannt sind. Dies gilt insbesondere für die Optima. Sie ermöglichen es, Algorithmen zu testen, ihr Verhalten zu beobachten und die Ergebnisse mit den bekannten Optima zu vergleichen. Um dies zu erleichtern, handelt es sich hier meist um relativ einfache Funktionen, die leicht zu verstehen, zu implementieren und zu berechnen sind.

Es wurden zum Testen der Algorithmen zwei klassische Standard-Testfunktionen implementiert: die SPHERE-Funktion und die RASTRIGIN-Funktion. Diese Funktionen wurden von dem Behavior, welches auch für die Optimierung der Laufparameter im Simulator zuständig ist (siehe Abschnitt 6), benutzt. Die Testfunktionen wurden anstelle der Fitness-Messung (die ansonsten über die Messung der Roboter-Geschwindigkeit erfolgt) aufgerufen.

Sphere Die SPHERE-Funktion (Abbildung 5.2(a)) ist das $n-$Dimensionale Pendant zur Normalparabel. Sie ist definiert als:



(a) Die Testfunktion SPHERE.

(b) Die Testfunktion RASTRIGIN.

Abbildung 5.2: Die Testfunktionen im $\mathbb{R}^2$.

$$f_{SPH}(\vec{x}) = \sum_{i=1}^{n} x_i^2$$

Die SPHERE-Funktion ist unimodal und im gesamten $\mathbb{R}^n$ definiert. Das Minimum der SPHERE-Funktion liegt bei $\vec{x} = (0, 0, ..., 0)$ und hat den Wert 0. Da die Laufoptimierung des Roboters eine Maximierung (der Laufgeschwindigkeit) ist, wurde zum Testen der Algorithmen das Negative der SPHERE-Funktion, also $-f_{SPH}(\vec{x})$ benutzt und maximiert. Die Negative SPHERE-Funktion liefert somit nur Werte ≤ 0 und hat ihr Maximum bei 0.

Rastrigin Die RASTRIGIN-Funktion (Abbildung 5.2(b)) ist definiert als:

$$f_{RASTR}(\vec{x}) = 10n + \sum_{i=1}^{n} x_i^2 - 10 \cdot \cos(2\pi x_i)$$

Das Minimum der im gesamten $\mathbb{R}^n$ definierten Funktion liegt ebenfalls bei $\vec{x} = (0, 0, ..., 0)$ und hat den Wert 0. Die Funktion ist multimodal; der Cosinus-Term erzeugt viele lokale Minima, die diese Funktion somit schwieriger machen als die SPHERE-Funktion, welche nur ein Optimum hat. Da die Laufoptimierung des Roboters eine Maximierung (der Laufgeschwindigkeit) ist, wurde zum Testen der Algorithmen das Negative der RASTRIGIN-Funktion, also $-f_{RASTR}(\vec{x})$ benutzt und maximiert. Die Negative RASTRIGIN-Funktion liefert somit nur Werte ≤ 0 und hat ihr Maximum bei 0.

Die RASTRIGIN-Funktion kann im Allgemeinen[2] nicht von Evolutionsstrategien optimiert werden (siehe Hoffmeister et al. [HB92]), da diese aufgrund ihrer Präferenz für kleine Änderungen der Parameter in lokalen Optima der Funktion stecken bleiben. Trotzdem wurde sie hier implementiert, um eben dieses Verhalten der implementierten Optimier-Algorithmen zu beobachten. So konnte genauestens getestet werden, ob die Steuerung der Mutationsschrittweiten der Algorithmen korrekt funktioniert, auch wenn das globale Optimum der Funktion nicht erreicht wird. Wenn die Algorithmen in lokalen Optima stecken bleiben, so sollen sie dort

[2] Ausgenommen sind Algorithmen-Starts, bei denen sich die Population der Evolutionsstrategie bereits nahe des Optimums befindet.

immerhin die Mutationsschrittweiten herunterfahren, um die genaue Position dieses lokalen Optimums zu ermitteln.

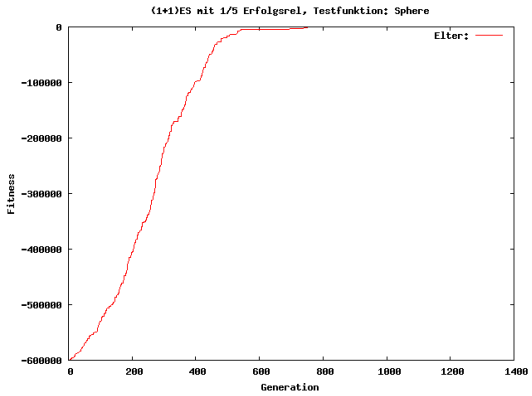
5.4.1 Ergebnisse der Tests

Um zu verifizieren, dass die implementierten Algorithmen korrekt funktionieren, wurden sie mit den beiden oben vorgestellten Testfunktionen validiert. Für alle vier in dieser Arbeit benutzten Optimier-Algorithmen ($(1+1)$-ES, $(1,\lambda)$-ES, (μ,λ)-ES und $(\mu+\lambda)$-ES mit κ) wurden die Behaviors so implementiert, dass von der Geschwindigkeits-Messung des simulierten Roboters als Fitnesswert auch auf die Testfunktionen umgeschaltet werden konnte. Dabei wurden alle der möglichen 15 Parameter des Parametersatzes der DortmundWalkingEngine zur Optimierung benutzt, d.h. die Testfunktionen optimierten im $\mathbb{R}^{15}$.

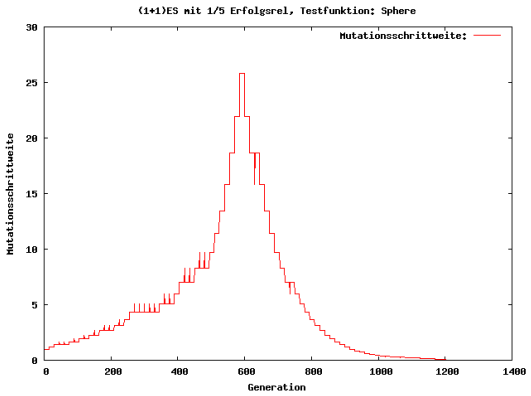
Gestartet wurde mit Initial-Parametersätzen, bei denen die Parameterwerte jeweils auf 200 gesetzt waren; die Mutationsschrittweiten, respektive σ-Werte waren initial alle auf 1 gesetzt. Diese Werte der Mutationsschrittweiten sind für eine schnell konvergierende Optimierung nun offensichtlich sehr niedrig, doch wurden sie mit Absicht so gewählt. Durch die niedrigen Initial-Mutationsschrittweiten konnte überprüft werden, ob die Algorithmen die automatische Schrittweiten-Adaptation auch korrekt durchführen. So muss der zu testende Algorithmus bei den hohen Startwerten der Parameter (200) die Mutationsschrittweiten zunächst erhöhen, um eine schnelle Konvergenz zu ermöglichen. Nahe des Optimums sind diese hohen Schrittweiten dann von Nachteil, so dass sie wieder stark verringert werden müssen. Die Optimierung wurde bei der SPHERE-Funktion gestoppt, wenn der jeweils beste Zielfunktionswert der Generation über $-0,1$ lag. Bei der RASTRIGIN-Funktion wurde abgebrochen, wenn sich über einen längeren Zeitraum keine Verbesserung einstellte.

Vor allem bei der RASTRIGIN-Funktion ist diese Art der Auswahl initialer Parameter natürlich sehr schlecht, da bei den Algorithmen mit Rekombination und einer Population aus mehr als einem Individuum wegen fehlender Diversität in der Initial-Generation kein guter Fortschritt bei der Optimierung zu erwarten ist. Da aber bekannt war, dass RASTRIGIN von Evolutionsstrategien sowieso nur mit äußerst geringer Wahrscheinlichkeit optimiert werden kann, kann dies hier vernachlässigt werden. Ferner wurde RASTRIGIN auch mit unterschiedlichen Initial-Individuen getestet. So wurden von Individuum zu Individuum der Startpopulation die Parameterwerte um jeweils 10 und die Strategie-Parameterwerte um jeweils 1 erhöht. Schon nach wenigen Generationen war die anfängliche Diversität auf ein Minimum geschrumpft und die Optimierung wurde mit Individuen fortgesetzt, die denen entsprachen, die bei einem Algorithmen-Lauf auftraten, bei dessen Initial-Population alle Individuen gleich waren.

Der Test der implementierten Algorithmen mit den Testfunktionen hat sich als sehr lohnenswert herausgestellt, da hierdurch ein Implementierungsfehler in dem Mutationsoperator aufgedeckt wurde, der von der $(1,\lambda)$, der (μ,λ) und der $(\mu+\lambda,\kappa)$-Evolutionsstrategie benutzt wurde. Hierdurch konnten somit schwere Fehler in der Optimierung und in den Untersuchungen der Algorithmen vermieden werden.



(a) Fitnessverlauf über die Generationen.



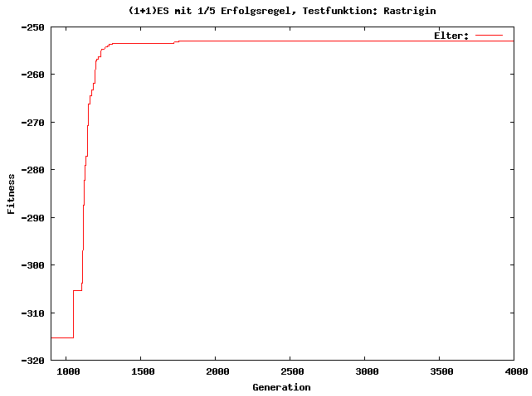
(b) Verlauf der Mutationsschrittweiten vom jeweiligen Elter der Population.

Abbildung 5.3: Verlauf der Fitnesswerte und der Mutationsschrittweiten der $(1 + 1)$-ES auf SPHERE. Vergrößerungen dieser Diagramme befinden sich im Anhang als Abbildung A.1 und Abbildung A.2.

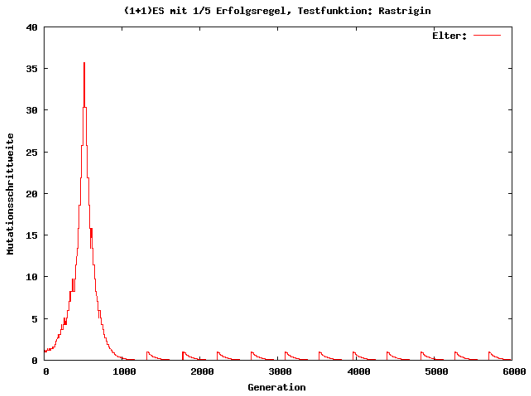
5.4.1.1 $(1 + 1)$-ES mit 1/5-Erfolgsregel

Bei der SPHERE-Funktion hatte die $(1 + 1)$-Evolutionsstrategie (Siehe Abbildung 5.3(a)) keine Probleme und optimierte mit schnellen Fortschritten in 1200 Funktionsauswertungen auf einen Wert über $-0,1$. Die Mutationsschrittweiten wurden anfangs wie erwartet stark gesteigert. Je näher der Algorithmus dem Optimum der Testfunktion kam, um so stärker hinderten die bis dahin stark angewachsenen Mutationsschrittweiten einen weiteren Fortschritt; die Mutationen waren einfach zu groß, um bessere Punkte zu erreichen. So wurden die Mutationsschrittweiten sukzessive wieder verringert (siehe Abbildung 5.3(b)). Der Algorithmus nimmt die Schrittweiten-Steuerung also korrekt vor. Dass der Algorithmus hier sehr schnell Erfolg hatte, liegt aber auch in der symmetrischen Form der SPHERE-Funktion: Alle Variablen haben dasselbe Gewicht. So ist die Mutationsschrittweite, die ja auf alle Variablen gleich angewandt wird, auch für alle Variablen auch angemessen.

Etwas anders sieht dies bei der RASTRIGIN-Funktion aus. Obwohl diese auch symmetrisch ist, so können in den einzelnen lokalen Optima der Funktion verschieden große Mutationsschrittweiten für die verschiedenen Variablen des Problems von Vorteil sein, was die (1+1)-ES jedoch nicht leisten kann. So blieb die Optimierung bereits nach etwa 1500 Generationen bei einem Fitnesswert von -260 in einem lokalen Optimum stecken (siehe Abbildung 5.4(a)) und konnte, da die (1+1)-ES ja keine Verschlechterungen akzeptiert, auch nicht aus diesem entkommen. Da es keine Verbesserungen gab, wurde die Mutationsschrittweite immer weiter abgesenkt, was erst recht nicht aus dem lokalen Optimum herausführte. Hier konnte auch die Modifikation der Strategie, bei einem Absinken der Mutationsschrittweite unter $0,01$ diese wieder auf 1 anzuheben (siehe Abbildung 5.4(b)), nicht aushelfen. Dies liegt allerdings auch daran, dass die Grenzwerte der Mutationsschrittweiten von 1 und $0,01$ auf das Problem der Laufparameter-Optimierung abgestimmt sind, und für die RASTRIGIN-Funktion nicht angemessen sind. Dass (1+1)-ES für multimodale Probleme schlecht geeignet sind, zeigt sich hier deutlich.



(a) Fitnessverlauf über die Generationen, vergrößert ab Generation 800.



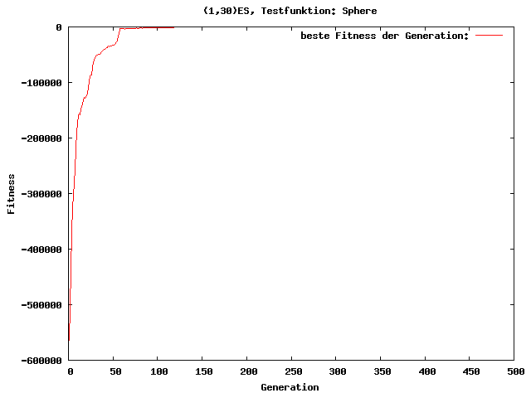
(b) Der Verlauf der Mutationsschrittweiten vom jeweiligen Elter der Population.

Abbildung 5.4: Verlauf der Fitnesswerte und der Mutationsschrittweiten der $(1 + 1)$-ES auf RASTRIGIN. Vergrößerungen dieser Diagramme befinden sich im Anhang als Abbildung A.3 und Abbildung A.4.

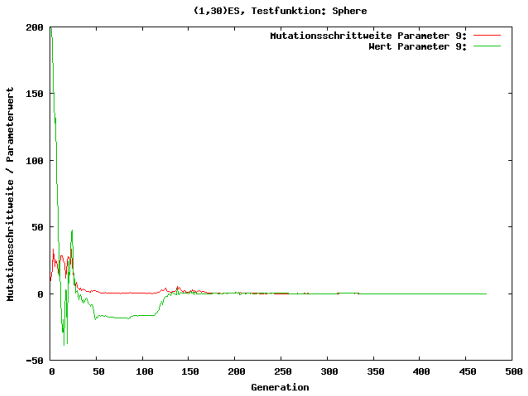
5.4.1.2 $(1, 30)$-ES mit σ-Selbstadaptation

Auf der SPHERE-Funktion wurde eine Fitness von unter -0.1 nach 470 Generationen erreicht, was 14.100 Funktionsauswertungen entspricht. Diese im Vergleich zu der (1+1)-ES hohe Anzahl von Funktionsauswertungen war jedoch zu erwarten, da eine $(1, \lambda)$-Evolutionsstrategie nicht nur, wie jede Komma-Strategie, Verschlechterungen in Kauf nimmt sondern auch noch nur *ein* Elter der jeweiligen Generation in die nächste übernommen wird, und so nur sehr wenig Diversität in den Parametern erhalten bleibt. Es wird ja jeweils zwar das fitteste Individuum der Offsprings übernommen, jedoch entfällt bei der Generierung der nächsten Offsprings aus diesem die Rekombination. Sind nun einzelne Parameterwerte schlecht (dies gilt auch für die Strategie-Parameter), so bleiben diese gezwungenermaßen erhalten. Bei einer Rekombination von mehreren Eltern würden diese schlechten Parameter mit großer Wahrscheinlichkeit jedoch nicht übernommen, sondern durch die besseren Parameter-Komponenten anderer Eltern ersetzt. So ist zu erwarten, dass bei einer (μ, λ)-ES mit $\mu > 1$ der Forschritt pro Generation wesentlich größer ist, da sich die einzelnen guten Parameter des Parametersatzes zu einem neuen, wesentlich besseren Parametersatz zusammenfinden. Dieses Verhalten kann in Abbildung 5.5(b) exemplarisch für den Parameter Nr. 9 gesehen werden: erst nach längerer Zeit wird dieser durch ein erneutes Ansteigen der Mutationsschrittweite wieder in Richtung 0 abgesenkt, was letztendlich zu einer starken Verbesserung des Fitnesswertes führt.

Die $(1, 30)$-Evolutionsstrategie funktionierte auch auf der RASTRIGIN-Funktion schlecht. So gab es zwar in den ersten 80 Generationen ständig große Verbesserungen und annähernd keine Veschlechterung zu den Nachfolge-Generationen, jedoch änderte sich dies ab etwa Generation 100 langsam und ab Generation 370 deutlich. Von Generation 400 bis 600 gab es keinen Fortschritt bei den Fitness-Werten, die Optimierung steckte fest bei einem Fitnesswert um -513. So wurde der Algorithmus nach 600 Generationen abgebrochen, da es keine Erhöhung der Mutationsschrittweiten mehr gab. Bei den jeweils 30 Offsprings pro Generation entsprach dies bereits 18.000 Funktions-Auswertungen. Wie schon bei den Ergebnissen der $(1, 30)$-ES



(a) Fitnessverlauf des jeweiligen Elter über die Generationen.



(b) Die Mutationsschrittweiten eines ausgewählten Parameters der jeweiligen Elter über die Generationen.

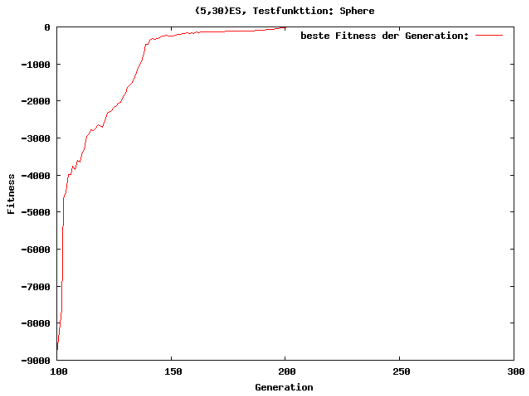
Abbildung 5.5: Die $(1, \lambda)$-ES auf SPHERE. Vergrößerungen dieser Diagramme befinden sich im Anhang als Abbildung A.5 und Abbildung A.6.

auf SPHERE beschrieben, liegt dies auch hier an der fehlenden Rekombination durch $\mu = 1$ in diesem Fall. So deuten auch hier die Ergebnisse der Testfunktionen schon darauf hin, dass dieser Algorithmus auch bei der Optimierung der Laufparameter für den Kondo KHR-1 nicht gut funktionieren wird.

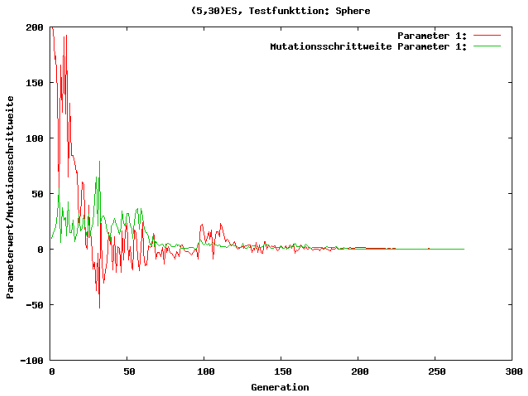
5.4.1.3 $(5, 30)$-ES mit σ-Selbstadaptation

Auf der SPHERE-Funktion erreicht die $(5, 30)$-Evolutionsstrategie schnell den geforderten Minimal-Abstand zum Optimum von $0,1$, nach bereits 200 Generationen, entsprechend 6000 Funktionsauswertungen erreichte der Algorithmus den Schwellenwert (siehe Abbildung 5.6(a)). Auch die Mutationsschrittweiten wurden korrekt adaptiert, beispielhaft zeigt dies Abbildung 5.6(b) für den ersten der insgesamt 15 Parameter, die von dem Algorithmus auf der SPHERE-Funktion optimiert werden mussten.

Auf der RASTRIGIN-Funktion konnte die $(5, 30)$-ES nicht überzeugen; von Generation 200 bis Generation 900 gab es keine Verbesserungen. Der Algorithmus blieb mit Fitness -79 in einem lokalen Optimum stecken, so dass nach den insgesamt 27.000 Funktionsauswertungen abgebrochen wurde, ohne in die geforderte Nähe zum Optimum gelangt zu sein. Weitere Tests wurden auf RASTRIGIN mit der (μ, λ)-ES für $\mu = 10$ und $\lambda = 100$ gemacht; auch hier blieb der Algorithmus in einem lokalen Optimum stecken, in dem Fall mit dem etwas besseren Fitness-Wert von $-8,9$ nach 140 Generationen, was 14.000 Funktionsauswertungen entspricht. Da auch nach 300 Generationen, respektive 30.000 Funktionsauswertungen keine Änderungen mehr auftraten und sich die bis dahin sehr niedrigen Mutationsschrittweiten nicht wieder vergößerten, wurde der Algorithmen-Lauf abgebrochen, ohne dass die geforderte Nähe zum globalen Optimum erreicht worden wäre.



(a) Fitnessverlauf des jeweils fittesten Elter über die Generationen, vergrößert ab Generation 100.

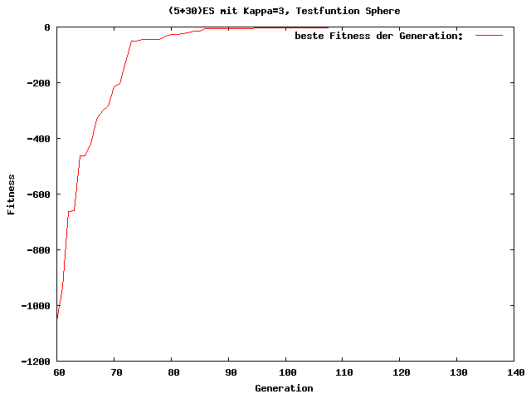


(b) Die Mutationsschrittweiten eines ausgewählten Parameters, sowie die Werte dieses Parameters für das jeweils fitteste Elter der Generation.

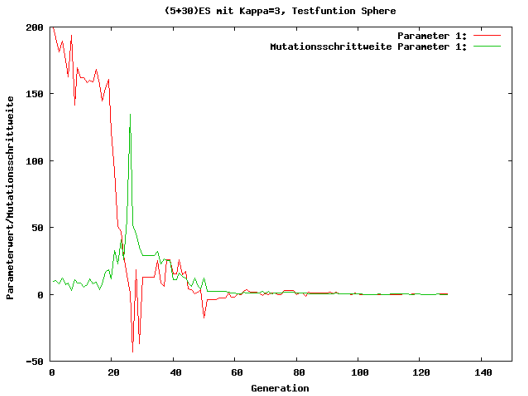
Abbildung 5.6: Die (5,30)-ES auf SPHERE. Vergrößerungen dieser Diagramme befinden sich im Anhang als Abbildung A.7 und Abbildung A.8.

5.4.1.4 (5+30)-ES mit σ-Selbstadaptation und Maximal-Lebensdauer $\kappa = 3$

Die (5+30)-ES mit $\kappa = 3$ überschritt den geforderten Minimal-Fitnesswert auf SPHERE bereits nach 125 Generationen, was 3750 Funktionsauswertungen entspricht. Abbildung 5.7(a) zeigt den zugehörigen Fitnessverlauf. Dabei sind zum Teil kleine Stufen zu erkennen, was darauf hindeutet, dass hier das fitteste Elter der Generation über mehrere Generationen (bis maximal $\kappa = 3$ Generationen) in der Population blieb. Die Seltenheit dieser Stufen spricht dafür, dass jedoch meistens pro Generation bessere Individuen gefunden wurden. Abbildung 5.7(b) zeigt den zugehörigen Verlauf eines ausgesuchten Parameters sowie dessen Mutationsschrittweite σ. Auch hier sind die die Stufen erkennbar.



(a) Fitnessverlauf des jeweils fittesten Elter über die Generationen, vergrößert ab Generation 60.



(b) Die Mutationsschrittweiten eines ausgewählten Parameters, sowie die Werte dieses Parameters für das jeweils fitteste Elter der Generation.

Abbildung 5.7: Die (5+30)-ES auf SPHERE. Vergrößerungen dieser Diagramme befinden sich im Anhang als Abbildung A.9 und Abbildung A.10.

Auch die (5+30)-ES konnte RASTRIGIN nicht bis zum globalen Optimum verfolgen; nach 1000 Generationen, entsprechend 30.000 Funktionsauswertungen wurde abgebrochen, da sich der Fitnesswert über 500 Generationen in der Nähe von -27 befand und nicht weiter änderte.

6 Laufevolution

In diesem Kapitel wird erläutert, wie die Optimierung der Laufparameter für den Kondo-KHR1 durchgeführt wurde. Hierbei wurden die Laufparameter sowohl im physikalischen Simulator des BreDoBrothers-Framework (Abschnitt 2.2.2) optimiert, als auch mit dem realen Roboter. Der Simulator wurde aufgrund seiner Abweichung von der realen Welt, insbesondere was die Wechselwirkung zwischen Roboterfuß und Untergrund angeht, nur für grundsätzliche Untersuchungen benutzt. So wurde z.B. das Verhalten der verschiedenen Optimier-Algorithmen oder der Zusammenhang der Laufparameter für verschiedene Laufrichtungen (translatorisch) untersucht.

6.1 Laufoptimierung im Simulator

Da der Roboter im Simulator nur virtuell läuft, können viele Vorteile ausgenutzt werden, die es beim realen Roboter nicht gibt. So kann die aktuelle Position und darüber auch indirekt die aktuelle Geschwindigkeit des Roboters direkt aus dem Zustand der Simulation abgefragt werden, es werden keine Messaufbauten benötigt. Ferner wird überhaupt kein realer Roboter benötigt, dies ist vom finanziellen Aspekt her attraktiv. Außerdem wird kein Personal benötigt, das den Roboter beaufsichtigt, um z.B. Beschädigungen oder ungewollte Zustände während der Bewegungen (ein Liegenbleiben des Roboters, ein Verkeilen mit Hindernissen, etc.) zu verhindern. Schlussendlich wird der reale Roboter auch nicht abgenutzt oder sogar beschädigt. Je nach Rechner-Geschwindigkeit ist es auch möglich, dass die Simulation wesentlich schneller abläuft, so dass Tests im Simulator wesentlich schneller beendet sind, als es mit dem realen Roboter möglich wäre. Bei dem Simulator des BreDoBrothers-Frameworks konnte dies jedoch nicht erreicht werden. Auf einem aktuellen Pentium 4-PC mit 3GHz und 1GB Arbeitsspeicher ist die Simulations-Zeit nur etwa halb so schnell wie die Realität.

Um vor allem Tests mit den für diese Arbeit benutzten Optimier-Algorithmen zu erleichtern, wurden diese zunächst alle im Simulator getestet. Dazu wurde ein *Behavior* (siehe Abschnitt 2.3.2) für das BreDoBrothers-Framework implementiert, in dem der Optimier-Algorithmus nebst Messung des Gütekriteriums ausgeführt wurde. Hier bestand das Güte-Kriterium, im Folgenden *Fitness* genannt, alleine aus der Laufgeschwindigkeit des Roboters bei jeweiligem Laufparameter-Satz. Die Fitness-Messung erfolgte dabei über ein Abfragen der Roboter-Position, sowie eine Messung der benötigten Zeit für eine vorgegebene Laufstrecke. Daraus konnte die Geschwindigkeit und damit der Fitnesswert des Laufparameter-Satzes errechnet werden.

Für jeden der benutzten Algorithmen wurde nun ein eigenes Behavior implementiert. Alle gemeinsam genutzten Methoden (wie etwa die Detektion, ob der Roboter umgefallen ist, die Ermittlung des Fitness-Wertes, gemeinsam genutzte Operationen für die Optimier-Algorithmen, etc.) wurden dabei in einer gemeinsamen Klasse implementiert, die dann von allen Behaviors

benutzt wurde. Um die Benutzung dieser Behaviors, im Speziellen in Hinblick auf die parallele Simulation auf mehreren Rechnern, zu vereinfachen, wurde für jedes Behavior eine eigene „Szene“ erstellt. Diese „Szenen“ sind Konfigurations-Dateien für den Simulator, über die die Eigenschaften des Roboters, der Umgebung (hier eine endlose Ebene) und der verwendeten Software-Module (hier das entsprechende Behavior mit dem jeweiligen Optimier-Algorithmus) definiert werden. Der Simulator kann einfach mit der gewünschten „Szene“ gestartet werden, und die Optimierung der Laufparameter läuft fortan automatisch ab. Nur dadurch war es möglich, die parallele Simulation auf mehreren Rechnern einfach und ohne großen Aufwand durchzuführen.

Die Behaviors für die simulierte Optimierung der Laufparameter laufen folgendermaßen ab:

1. **Initialisierung**
   Es werden entweder die initialen Individuen (=Laufparameter) für den benutzten Optimier-Algorithmus oder, falls vorhanden, die Individuen mit der höchsten Generationsnummer geladen. Die initialen Individuen sind Parametersätze, die per Hand erstellt wurden, da eine zufällige Initialisierung in den allermeisten Fällen zu nicht funktionsfähigen Parametersätzen führen. Um den Optimieralgorithmen nicht bereits am Anfang mit einer Population starten zu lassen, die gar nicht erst lauffähig ist und damit eine Beurteilung der Fitness vorzuenthalten, wurde dieser Weg gewählt.

   Falls sich im Ausgabe-Verzeichnis des Algorithmus (hier werden alle Individuen nach Generationen und eindeutiger Nummer abgespeichert) keine gespeicherten Individuen befinden, wird die initiale Population geladen. Ansonsten wird die Population mit der höchsten Generationen-Nummer geladen. Dadurch kann ein gestoppter Algorithmus an der Abbruchstelle wieder fortgesetzt werden.

2. **Anwendung der Laufparameter**
   Die Laufparameter des aktuell zu evaluierenden Individuums werden an die WalkingEngine übergeben. Dies wird, falls aktuell noch nicht durchgeführt, für alle Individuen der Population sowie für alle neu erzeugten Nachkommen („Offsprings“) durchgeführt. Dabei wird beim MotionRequest (siehe Abschnitt 2.3.1) die Laufgeschwindigkeit 0 angegeben, so dass der Roboter auf der Stelle tritt.

3. **Einpendeln**
   Damit der Roboter sich auf die Laufbewegung einschwingen kann, wird zwei Sekunden lang gewartet, damit sich die Bewegung der WalkingEngine auswirken kann.

4. **Loslaufen**
   Erst nach Ablauf der 2 Sekunden Wartezeit zum Einschwingen läuft der Roboter los.

5. **Abfrage der Position**
   Wenn die in Abschnitt 4.3 beschriebene Beschleunigung der Laufgeschwindigkeit abgeschlossen ist, wird die Position des Roboters im Raum seiner simulierten Welt festgehalten. Der Roboter läuft nun mit konstanter Geschwindigkeit weiter.

6. **Anhalten**
   Bei jedem Durchlauf des Behaviors wird nun abgefragt, ob der Roboter die vorgegebene

Laufdistanz (hier 450 mm) absolviert hat. Darufhin wird die dafür benötigte Zeit gemessen. Aus den beiden Werten wird die Geschwindigkeit errechnet und als Fitnesswert für den aktuellen Laufparametersatz gespeichert. Die im MotionRequest angegebene geforderte Laufgeschwindigkeit wird auf 0 gesetzt.

7. **Stillstehen**
   Wenn die tatsächliche Laufgeschwindigkeit des Roboters gleich Null ist, wird der Roboter per MotionRequest zum Stillstehen gebracht, damit keine Bewegung des Laufparametersatzes mehr durchgeführt wird. Dadurch wird sichergestellt, dass beim nächsten Ausführen der WalkingEngine mit einem anderen Parametersatz das Umschalten vom einen Parametersatz auf den nächsten ohne Beeinflussung stattfindet.

8. **Individuum speichern**
   Das aktuelle Individuum wird mitsamt des gerade gemessenen Fitnesswertes auf die Festplatte des Rechners gespeichert. Für die Simulationen auf mehreren Rechnern parallel wurden die Individuen zusätzlich noch zentral auf einem Netzlaufwerk abgespeichert. Das einzelne Abspeichern ermöglicht einerseits die spätere Analyse der Optimierung und befähigt das Behavior zudem, nach einer Unterbrechung später an der Abbruchstelle fortzufahren.

9. **Nächstes Individuum**
   Je nach Art des Optimier-Algorithmus werden nun ggf. spezielle Aktionen wie Rekombination, Mutation, Selektion, Erhöhung der Generationsnummer, o.ä. durchgeführt. Danach wird das nächste zu evaluierende Individuum bestimmt und bei Schritt 2. fortgefahren.

Die oben beschriebene Optimier-Schleife wird so lange wiederholt, bis sie vom Benutzer abgebrochen wird, oder bis die vorher festgelegte Anzahl maximaler Generationen des Algorithmus erreicht ist. Nach Ende des Algorithmen-Laufes können die Ergebnisse aus dem Output-Verzeichnis geladen werden. Für die Untersuchung der Outputs (Erstellung von Grafiken, Ermittelung einzelnen Fitnesswerte, usw.) wurden eigene Tools geschrieben, durch die die in dieser Arbeit verwendeten Diagramme erstellt wurden.

Bei Evolutionsstrategien mit „Komma“-Selektion wird pro Generation zudem das global beste Individuum ermittelt und im Output-Verzeichnis abgespeichert. Dadurch wird vermieden, dass ein einmal erzeugtes, sehr gutes Individuum wieder vergessen wird. Dies dient jedoch nur dem Benutzer, der an möglichst guten Laufparametern interessiert ist. Das global beste Individuum wird von keinem der hier benutzten Evolutionsstrategien verwendet.

Ein besonderes Augenmerk gilt nun noch dem Umfallen des Roboters. Je nach Art des verwendeten Parametersatzes für die WalkingEngine kann der Roboter umfallen. Dies ist nicht erwünscht, kann aber nicht im Voraus auf analytischem Weg abgefragt werden. Also muss damit gearbeitet werden, dass der Roboter auch in der Simulation stürzt. Da dies einen unerwünschten Parametersatz als Ursache hat, muss dieser Parametersatz, also das zu vermessende Individuum, hinsichtlich seines Fitnesswertes beurteilt werden. Hier gibt es zwei Möglichkeiten: entweder wird der Fitnesswert auf einen konstant niedrigen Wert gesetzt, oder es wird ein Strafbetrag vom Fitnesswert abgezogen. Da in diesem Fall kein sinnvoller Strafbetrag ange-

geben werden kann[1], wird der Fitnesswert konstant auf 0 gesetzt, wenn der Roboter gestürzt ist. Dies entspricht dem bei Geschwindigkeiten niedrigst möglichen Betrag.

Genauso wird verfahren, wenn die vom Optimieralgorithmus erzeugten Parameter des zu messenden Individuums außerhalb der erlaubten Grenzen[2] liegen. Der Fitnesswert wird dann auf 0 gesetzt. In allen Algorithmen außer der (1+1)-ES werden die Nachkommen in diesem Fall so lange erzeugt, bis die Parameter allesamt im erlaubten Bereich liegen. Dies gilt jedoch nicht für den Fitnesswert 0 durch ein Umfallen. Ferner wird die Fitness auch auf 0 gesetzt, wenn der Roboter nicht in einer Mindest-Zeit die vorgegebene Laufstrecke geschafft hat. Ansonsten könnten Laufparameter, die den Roboter womöglich gar nicht vorwärts bewegen, das System ohne Forschritt blockieren. Außerdem können damit Fälle abgefangen werden, in denen ein Sturz nicht als solcher erkannt wird. So ist es vor Einführung dieser Mindestzeit schon passiert, dass der Roboter auf den seitlich ausgestreckten Arm fiel und sich stundenlang auf der Stelle hin und her bewegte.

Ist nun der Roboter durch einen unzulänglichen Parametersatz umgefallen, so muss er für die Fortführung der Optimierung natürlich wieder aufstehen. Am einfachsten wäre dies durch einen Neustart des Simulators, was jedoch aus dem Behavior heraus nicht möglich ist, da es sich dabei ja um ein gekapseltes Modul handelt, welches dafür nie vorgesehen war. Also muss der simulierte Roboter, so wie der reale auch, selbstständig wieder aufstehen. Das Aufstehen wird über die *SpecialActions* ausgeführt, die vom MotionRequest angefordert werden. Über den Weltzustand des Simulators wird bei jedem Behavior-Durchlauf abgefragt, ob der Roboter gestürzt ist. Falls ja, so wird abgefragt, auf welcher Seite der Roboter liegt, daraufhin wird die entsprechende SpecialAction aufgerufen, das den Rooter aufstehen lässt. Danach wird mit Schritt 7. fortgefahren.

6.2 Laufoptimierung im Laufgestell

Um die Laufgeschwindigkeit der Parametersätze für den realen Roboter zu messen, musste ein anderer Weg als beim Simulator gewählt werden. Zum einen lässt sich nun die Geschwindigkeit nicht so leicht bestimmen, zum anderen ist der reale Roboter vor allem in Sachen Beschädigungen empfindlich. So musste eine Möglichkeit gefunden werden, bei welcher der Roboter bei den erwartungsgemäß häufig vorkommenden Stürzen keinen Schaden erleidet. Es wurde in Anlehnung an den Versuchsaufbau von Wolff und Nordin [WN02] ein Laufgestell entwickelt und am Insititut für Roboterforschung erbaut.

Das Laufgestell (in Abbildung 6.1 dargestellt) dient einerseits zur Messung der Laufgeschwindigkeit und verhindert andererseits, dass der Roboter bei einem Sturz beschädigt wird. Das Gestell besteht aus einer Laufbahn von $2,5$ m Länge und 30 cm Breite. Der Boden der Lauf-

[1] Die einzige Möglichkeit, die dem Algorithmus noch einen Aufschluss über die Güte geben könnte, wäre es, zu messen, nach welcher Laufstrecke der Roboter umgefallen ist. Dies würde womöglich schnelle Individuen bevorzugen, bei denen der Roboter erst nach längerem Laufen umgefallen ist. Dies würde aber zu Laufarten führen, die zwar schnell sind, aber den Roboter stürzen lassen, wenn auch spät. Bei dem Ansatz der Fitness = 0 bei jedem Sturz werden nur Individuen weiterverfolgt, die auch wirklich zu funktionierendem Laufen führen.

[2] Ein Amplitudenwert kann z.B. nicht negativ sein, ferner gibt es z.B. Minimal- und Maximalwerte für die Schrittlänge, die Schrittgeschwindigkeit, etc.. Für Details siehe Abschnitt 4.5.

bahn ist mit einem Teppich der Art bezogen, wie er auch für die Spielfelder beim Robocup benutzt wird. So ist sichergestellt, dass die Optimierung der Laufparameter in Hinsicht auf die Verwendung des Roboters bei Fußballspielen erfolgt. Seitlich der Laufbahn wurden zwei niedrige Wände angebracht, die dem Roboter Führung beim Laufen geben, und ferner zum Aufstehen nach einem Umfallen benutzt werden. Ohne diese Bande würde der Roboter zu oft seitlich aus dem Laufgestell herauslaufen. Dies passiert vor allem bei Laufparametern, die ein Geradeauslaufen nicht gut ermöglichen. Trotzdem werden solche Parametersätze vom Optimierverfahren benachteiligt, weil das Entlangschleifen an der Bande die Geschwindigkeit deutlich senkt. So werden ganz automatisch Parametersätze bevorzugt, die ein gerades Laufen ermöglichen.

In bis zu 50 cm Höhe verlaufen zwei Aluminium-Traversen, welche in ihrer Höhe stufenlos verstellbar sind. Um diese Aluminium-Traversen ist ein Drahtbügel gelegt, der am Kopf-Servo des Roboters befestigt ist. Wenn der Roboter nun beim Laufen kippt, verhindert dieser Drahtbügel ein komplettes Umfallen; der Roboter hängt dann mit dem Bügel an den Traversen. Die Höhenverstellung der Traversen dient dazu, das Laufgestell an die Roboter-Höhe, an Gangarten mit verschieden hoch aufgerichtetem Roboter und an verschiedene Drahtbügel-Konstruktionen schnell und einfach anzupassen. Durch das Lösen zweier Rändelschrauben kann die Höhe in Sekundenschnelle geändert werden.

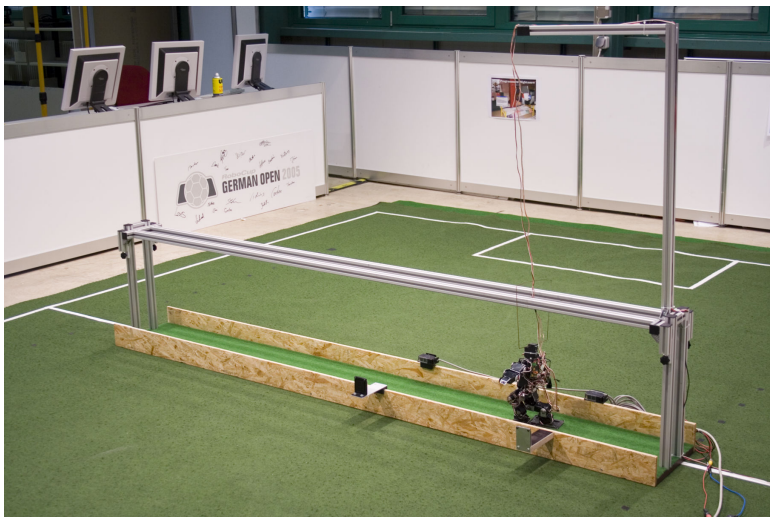


Abbildung 6.1: Das Laufgestell für die Optimierung der Laufparameter auf dem realen Roboter in der Übersicht. Von dem oben angebrachten Ausleger führen die Versorgungskabel zum laufenden Roboter herunter.

Da der Roboter die Optimierung über längere Zeit im Laufgestell durchführen können soll, wird der Roboter per Kabel mit Strom versorgt. Da zu diesem Zweck sowieso schon Kabel zum Roboter geführt werden müssen, wird der Roboter ferner per seriellem Kabel von einem PC angesteuert, wodurch das umständlichere Arbeiten mit einem PocketPC vermieden werden konnte. Vor allem die Tatsache, dass der PocketPC ja selbst auch mit Strom versorgt werden müsste, spricht dafür, so kann die Geschwindigkeits-Messung leichter durchgeführt werden. Die Kabel zum Roboter werden diesem von einem über dem Laufgestell angebrachten, 1 m

langen Arm von oben zugeführt. Dies ist die bisher schonendste Variante der Kabelführung, die bei Versuchen mit dem Roboter in dem Laufgestell gefunden wurde. Eine Anordnung der Kabelführung mit gleitenden Ösen, wie von Nordin et al. [WN02] benutzt, hat sich hier nicht bewährt, da durch diese Methode der Roboter durch die Reibung der Ösen zu stark beeinflusst wurde. Das Laufverhalten wich in diesem Fall stark von einem freien Laufen ab. Hier sind nun die Kabel fest am Rücken des Kondo KHR-1 angebracht, da dies die geringste Ablenkung des Roboters bedingt. Versuche mit der Kabel-Anbringung am Drahtbügel oder am Kopf waren nicht zufriedenstellend, da dort der Hebelarm, den das Kabelgewicht auf den Roboter ausüben kann, zu groß ist.

Zur Messung der Laufgeschwindigkeit wurden zwei Lichtschranken in 5 cm Höhe über der Lauffläche angebracht (siehe Abbildung 6.2). Die erste Lichtschranke ist fest 40 cm vom Anfang des Laufgestell entfernt[3], die zweite Lichtschranke kann variabel an jeder Stelle der restlichen Strecke angebracht werden. Die Abfrage der Lichtschranken erfolgt per paralleler Schnittstelle über das Behavior des BreDoBrothers-Framework, mit dem die Laufoptimierung per Laufgestell durchgeführt wird. Dazu wurden eigene Module für die Ansteuerung der parallelen Schnittstelle implementiert. So wird durch die Lichtschranken ermittelt, wann der Roboter den Start und das Ziel der Laufstrecke durchschreitet. Durch die Verschiebung der zweiten Lichtschranke kann die Länge der Laufstrecke variiert werden. Dabei wurde für die Laufoptimierung ein Kompromiss zwischen Laufgenauigkeit und Dauer der Messung von 70 cm benutzt.

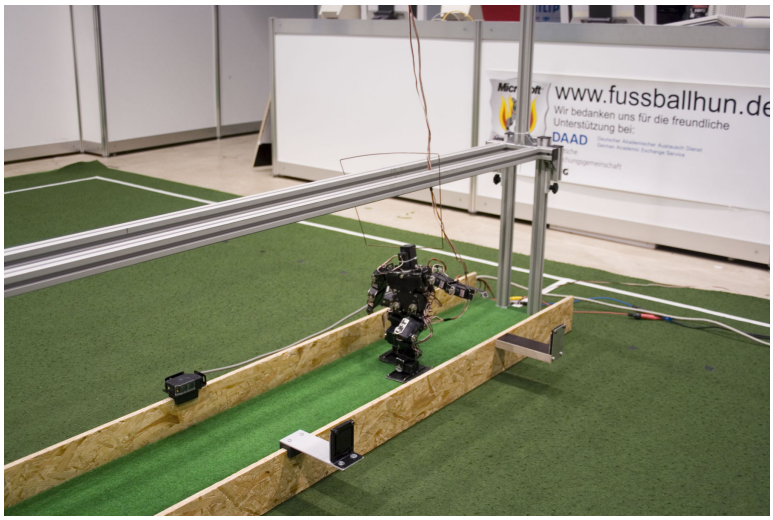


Abbildung 6.2: Nahaufnahme des Laufgestells mit Roboter. Hier sind die Lichtschranken auf der linken sowie deren Reflektoren auf der gegenüberliegenden Seite zu sehen. Ferner ist der am Roboter-Kopf angebrachte Haltebügel zu erkennen.

Da der Drahtbügel zum Auffangen des Roboters am drehbaren Kopf-Servo des Roboters befestigt wurde, ist es nun auch möglich, den Roboter auszurichten. Für ein schräges Laufen oder

[3]Durch diesen Streckenabschnitt erhält der Roboter die Möglichkeit, auf die volle angestrebte Geschwindigkeit zu beschleunigen, ehe er die erste Lichtschranke passiert. Dies gilt ebenso für das Zurücklaufen mit Lichtschranke Nummer 2 als Start.

sogar ein Seitwärtslaufen kann sich der Roboter, indem er die Füße hebt, an den Drahtbügel hängen. Durch anschließendes Drehen des Kopfservos um den gewünschten Translationswinkel α (siehe Abschnitt 4.4) wird der Roboter für die gewünschte Laufrichtung ausgerichtet, senkt seine Füße wieder herab und stellt sich dadurch wieder hin. Da der Roboter ja durch den Drahtbügel nicht in der Lage ist, sich um mehr als 90° zu drehen, wird diese Einschränkung genutzt, um die Optimierung der Laufparameter für ein Rückwärtslaufen gleich mit zu optimieren. So läuft der Roboter bei der Optimierung von Lichtschranke 1 zu Lichtschranke 2, bleibt dort stehen und läuft von dort rückwärts wieder bis zu Lichtschranke 1. So werden die beiden Optimierungen für das Vorwärts- und das Rückwärts-Laufen alternierend durchgeführt.

Aus diesem Grund sind alle Bestandteile des Behaviors doppelt für die Optimierung im Laufgestell: Es gibt zwei Populationen von Parametersätzen, es werden zwei Generationen-Zähler verwaltet, die Operatoren der Optimierverfahren werden für jede der beiden Laufrichtungen unabhängig voneinander durchgeführt, und so weiter. Somit wurde das Problem der Rückführung des Roboters zur Startposition genutzt, um die Laufparameter für gleich zwei Laufrichtungen zu optimieren.

Wie oben erwähnt, wird wegen des Problems der Stromversorgung der Roboter ohne PocketPC, aber mit einer Kabelführung betrieben. Um jedoch die Beeinflussung des Roboters durch Kabel komplett auszuschließen, kann der Roboter das Optimier-Behavior für das Laufgestell auch auf einem am Roboter befestigten PocketPC durchführen. Damit die Unterbrechungen der Lichtschranken dem PocketPC signalisiert werden können, wurde eine modifizierte Version des „Game Controllers“ geschaffen. Bei dem Game Controller handelt es sich um ein Programm, welches beim Robocup zur Steuerung der Roboter genutzt wird. Dadurch können Ereignisse und Befehle per Wireless LAN an alle am Spiel beteiligten Roboter gesendet werden. Es handelt sich um Informationen wie „Spielstart“, „Freistoß“, „Ende“, Deaktivierung einzelner Roboter, usw.. Hier wurde das im BreDoBrothers Framework bereits integrierte Game Controller-Interface benutzt, um die Unterbrechung der Lichtschranken zu signalisieren. Das Optimier-Behavior läuft selbstständig auf dem PocketPC; ein mit den Lichtschranken verbundener PC sendet über ein eigenes Programm, den „Evolution Controller“ zwei ausgesuchte Status-Meldungen des Game-Controller-Interfaces und teilt dadurch mit, welche Lichtschranke gerade unterbrochen wurde. Diese Meldungen werden per WLAN an den PocketPC am Roboter versandt. So kann der Roboter frei und ohne Irritationen per Kabel die Laufparameter weiter optimieren, allerdings mit dem Problem, dass nach etwa 30 Minuten die Akku-Kapazität erschöpft ist.

Analog zum Optimier-Behavior für den Simulator folgt hier nun die Vorgehensweise bei der Parameter-Optimierung mit dem Laufgestell und dem realen Roboter:

1. **Initialisierung**
   Auch hier werden entweder die initialen Individuen (=Laufparameter) für den benutzten Optimier-Algorithmus oder, falls vorhanden, die Individuen mit der höchsten Generations-Nummer geladen. Die initialen Individuen sind wiederum Parametersätze, welche „per Hand“ erstellt wurden. Falls bereits Individuen im Output-Verzeichnis vorhanden sind, so wird mit diesen fortgefahren.

2. **Anwendung der Laufparameter**
   Die Laufparameter des aktuell zu evaluierenden Individuums werden an die WalkingEn-

gine übergeben. Auch hier hat, wie im Simulator, der Roboter auf der Stelle zu stehen und mit Laufgeschwindigkeit 0 auf der Stelle zu treten. Dabei muss der Roboter beim ersten Start einige Zentimeter hinter Lichtschranke 1 stehen.

3. **Einpendeln**
   Damit der Roboter sich auf die Laufbewegung einschwingen kann, wird zwei Sekunden gewartet, damit sich die Bewegung der WalkingEngine auswirken kann.

4. **Loslaufen**
   Erst nach Ablauf der 2 Sekunden Wartezeit zum Einschwingen läuft der Roboter los; die aktuelle Zeit wird genommen, wenn der Roboter nach Unterbrechung die Lichtschranke 1 wieder verlässt. Von der Startposition bis zur Lichtschranke 1 ist dabei genug Abstand, dass der Roboter auf die mit diesem Parameter maximal mögliche Geschwindigkeit beschleunigen kann.

5. **Laufen**
   Solange Lichtschranke 2 nicht erreicht ist, läuft der Roboter mit dem zu messenden Parametersatz.

6. **Erreichen der Lichtschranke 2**
   Wird nun Lichtschranke 2 erreicht, wird der Roboter noch 2 Sekunden weiter laufen gelassen, bevor der MotionRequest mit Laufgeschwindigkeit 0 abgesetzt wird. Dadurch bremst der Roboter bis auf Stillstand ab. Danach wird der Roboter zu einem Stillstehen veranlasst, damit verschiedene Laufparametersätze einander nicht beeinflussen können. Aus der Zeit vom Verlassen der Lichtschranke 1 bis Erreichen der Lichtschranke 2 wird, zusammen mit der Distanz der beiden Lichtschranken, die Geschwindigkeit errechnet und dem Individuum zugewiesen. Ggf. notwendige Operationen, wie Rekombination, Mutation, Inkrementierung der Generations-Zahl, o.ä. werden jetzt durchgeführt

7. **Zurücklaufen**
   Nun wird gewechselt auf die Optimierung der Parameter für das Rückwärtslaufen. Es wird die für das Rückwärtslaufen zuständige Population benutzt; die zu messenden Individuen sind unabhängig von denen zum Vorwärtslaufen. Der Roboter wird mit dem zu beurteilenden Parametersatz laufen gelassen. Da er jetzt nach der vorherigen Abbremsphase und den zusätzlichen 2 Sekunden Laufzeit einige Zentimeter hinter Lichtschranke 2 steht, hat er nun genug Platz, um auf die maximale Laufgeschwindigkeit zu beschleunigen. Bei Verlassen der Lichtschranke wird wiederum die Zeit festgehalten.

8. **Erreichen der Lichtschranke 1**
   Hier passiert nun dasselbe wie in Punkt 6. Das Individuum wird bewertet, evtl. notwendige Operationen werden durchgeführt, der Roboter läuft anschließend wieder vorwärts in Richtung Lichtschranke 2.

Bei dieser Optimierung wird nach jeder Fitnessmessung das aktuelle Individuum abgespeichert, damit der Optimier-Vorgang jederzeit abgebrochen und später an derselben Stelle fortgeführt werden kann. Die beiden Populationen für das Vorwärts- und Rückwärtslaufen werden dabei in getrennten Verzeichnissen gespeichert.

Ein besonderes Augenmerk beim Laufgestell gilt dem Umfallen. Auch hier kommt es gelegentlich zum Umfallen des Roboters aufgrund schlechter Lauf-Parameter. Wie bereits erwähnt, hängt dann der Roboter an dem Drahtbügel. Um ein Umfallen zu detektieren, wird der eingebaute Beschleunigungs-Sensor (siehe Abschnitt 2.1.1) benutzt. Dieser liefert nun beim Laufen aufgrund der Pendel-Bewegungen nach vorne und zu den Seiten, sowie aufgrund des Aufstampfens auf den Boden, ständig schwankende Messwerte für alle drei Raumachsen. Diese Rohwerte sind nun nicht geeignet, um ein Umfallen zuverlässig zu erkennen, da die einzelnen Bewegungen teilweise Beschleunigungen bis zu $0,5$ g erzeugen.

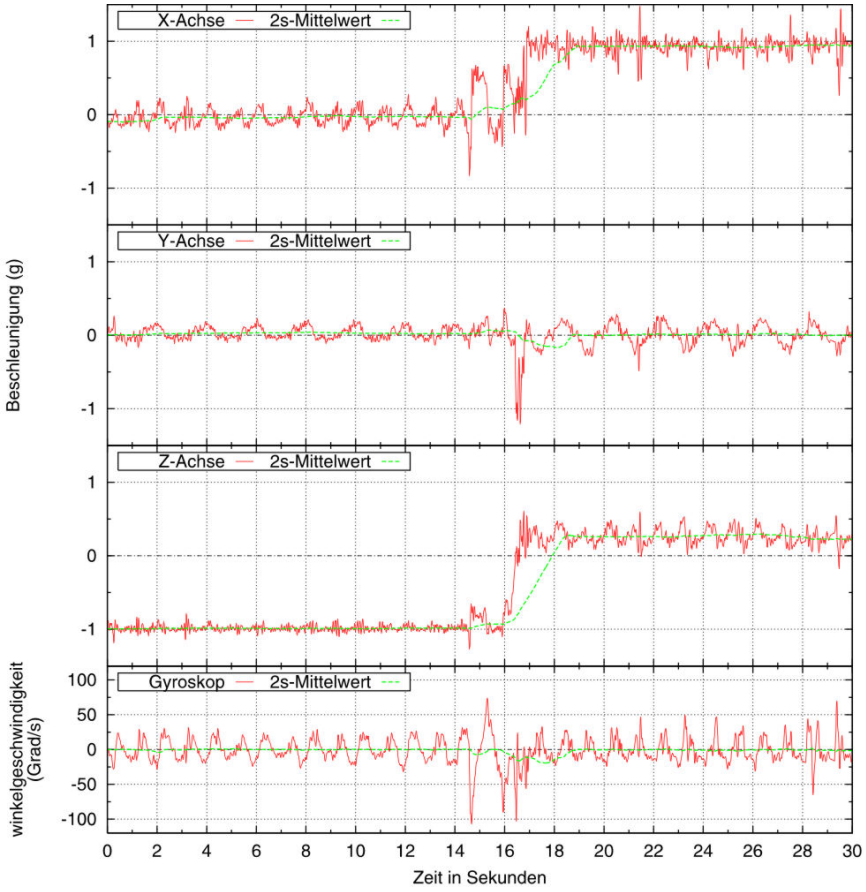


Abbildung 6.3: Die Roh-Sensordaten der Beschleunigungssensoren und des Gyroskops, sowie deren gleitender Durchschnitt beim Laufen und Fallen. Quelle: Kindler [Kin06].

Um diese schwankenden Bewegungen auszugleichen, wird eine Beobachtung ausgenutzt: Da es sich um jeweils periodische Schwingungen handelt und die Bewegungen alle symmetrisch verlaufen, sind die Beiträge mit negativem Anteil im Mittel so groß wie diejenigen mit positivem Vorzeichen. So ergibt sich für die Bewegungen in der X- sowie in der Y-Koordinatenrichtung als Mittelung über ausreichend viele Messwerte die Summe 0. Abbildung 6.3 zeigt die Sensor-Rohdaten sowie deren gleitender Durchschnitt über jeweils 100 Werte, also 2 Sekunden bei der 50 ms-Ansteuerung der WalkingEngine. Ermittelt wurden diese Daten von Kindler [Kin06] für die Fall-Erkennung des Kondo KHR-1.

In der Abbildung ist deutlich der periodische Verlauf der Werte für die X- und Y-Richtung zu erkennen. Entlang der Z-Achse verläuft der Beschleunigungs-Wert ständig nahe -1, da der Beschleunigungs-Sensor eine „echte“ Beschleunigung ja nicht von der Schwerkraft (entsprechen $1g = 9,80665m/s^2$) unterscheiden kann. Die Kurve des gleitenden Durchschnitts verläuft bei der X- und der Y-Richtung bei 0; für die Z-Achse bei -1. Zum Zeitpunkt 16 Sekunden fällt der Roboter. Die Verlagerung der Schwerkraft von der negativen Z-Richung (also nach „unten“) in Richtung der positiven X-Achse (die Vorderseite des Roboters) ist im Verlauf der Kurve erkennbar. So kann ein Umfallen leicht durch einen Vergleich der Werte erkannt werden. Die Grafik zeigt ferner, dass die Daten des Gyroskops nicht genau genug sind, um ein Umfallen deutlich zu erkennen. Zwar sieht das menschliche Auge das Fallen als leichte Anomalie, jedoch ist die Abweichung vom normalen Wert beim Laufen sehr gering. Um die Sturz-Erkennung robust zu halten, werden hier also nur die Werte des Beschleunigungs-Sensors benutzt.

Für die Ermittlung des gleitenden Durchschnitts wird der aktuelle Beschleunigungs-Vektor in jedem Durchlauf des Behaviors, also alle 20 ms erfasst. Da beim Laufgestell ein Umfallen möglichst nach kurzer Zeit erkannt werden soll, wurde nach Experimenten die Anzahl der Messwerte für den gleitenden Durchschnitt auf 50 gesenkt, es werden also die letzten 50 Werte, entsprechend den Messungen der letzten Sekunde gemittelt. Ferner wurden durch Versuche Grenzwerte für die gemittelten Beschleunigungswerte aller drei Raumachsen festgelegt. So wird nun vom Optimier-Behavior für das Laufgestell ein Umfallen angenommen, falls folgender Ausdruck wahr wird:

$$(|accX| > 0,4) \vee (|accY| > 0,4) \vee (|accZ| < 0,65)$$

mit $accX$ als gleitend gemittelte Beschleunigung in X-Richtung, $accY$ für die Y- und $accZ$ für die Z-Richtung, jeweils gemessen in [g]. Diese Art der Messung erkennt es nun nicht nur, wenn der Roboter nach einem Sturz am Drahtbügel hängt, sondern auch, wenn er Laufformen benutzt, die so ruckartig und unregelmäßig sind, dass es nach kurzer Zeit zum Sturz kommen muss. So können unvorteilhafte Laufparameter schon erkannt werden, bevor der Roboter umfällt. Sie werden vom Optimierverfahren dann genauso behandelt wie Parametersätze, die bereits zum Umfallen geführt haben.

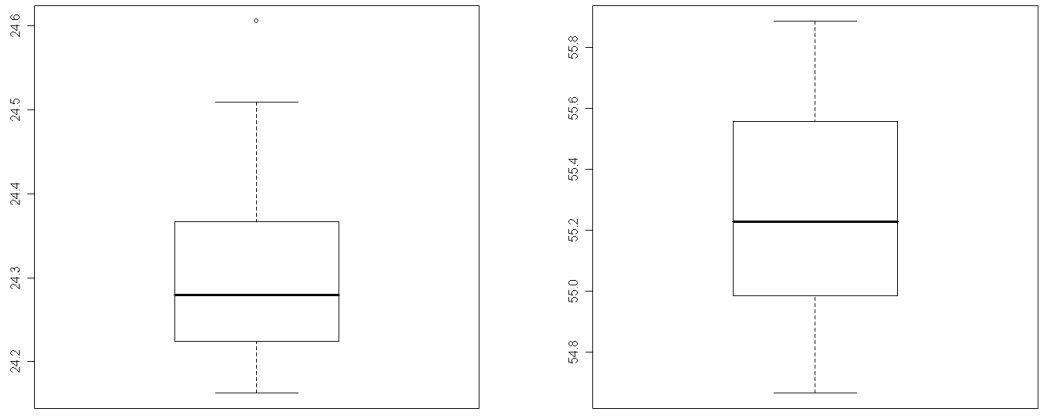
Ist nun ein Umfallen erkannt worden, so wird wie beim Simulator eine SpecialAction zum Aufstehen benutzt. Diese ist jedoch eine eigens für das Laufgestell entwicklte Bewegung. Dabei spreizt der Roboter zunächst seine Beine soweit, dass die Fuß-Ränder an die Bande des Laufgestells stoßen. Weil die Füße eine gerade Umrandung haben, wird dadurch der Roboter genau in Laufrichtung ausgerichtet. Er klemmt sich zwischen den beiden Banden ein und kann sich nun wieder aufrichten (es gibt zwei verschiedene Aufsteh-Bewegungen, je nachdem, ob der Roboter nach vorne oder nach hinten gefallen ist). Steht er wieder, so werden die Beine wieder aufeinander zu gezogen. Der Roboter befindet sich nun stehend mittig zwischen den beiden Seitenwänden des Laufgestells und ist in Laufrichtung ausgerichtet. Jetzt wird noch der offensichtlich nicht lauffähige Parametersatz ausgetauscht gegen Parameter, die erprobt sind und den Roboter gut, wenn auch langsam, laufen lassen. Dabei läuft der Roboter mit diesem Standard-Parametersatz in genau die Richtung weiter, in die er bereits vor dem Umfallen gelaufen ist. Die Fitness des laufunfähigen Parametersatzes wird auf 0 gesetzt, wie dies auch beim Simulator gehandhabt wird. Nach Erreichen der Lichtschranke wird die Optimierung dann wie gehabt fortgeführt.

7 Ergebnisse

In diesem Kapitel werden die bei der Optimierung der WalkingEngine-Parameter gewonnenen Ergebnisse präsentiert. Nach einer Untersuchung der Messfehler bei der Optimierung im Simulator und mit dem realen Roboter werden die Experimente mit den verschiedenen Optimier-Algorithmen vorgestellt. Danach folgt eine Übersicht über das Verhalten der Parameter-Einstellungen für verschiedene Laufrichtungen sowie abschließend eine Beschreibung der Lauf-Optimierung mit dem realen Roboter.

7.1 Messfehler

Während der Messung der Roboter-Laufgeschwindigkeit kommt es unweigerlich zu Messfehlern. Diese sorgen für Probleme bei den Optimierverfahren und bedürfen daher besonderer Algorithmen, die mit verrauschten Werten umgehen können (siehe Abschnitt 5.2). Um einen Eindruck zu gewinnen, wie groß diese Messfehler sind, wurde die Varianz der Messgrößen gemessen, indem für ein und denselben Parametersatz der WalkingEngine die Messung mehrfach wiederholt wurde. Dies wurde sowohl für den Simulator als auch für das Laufgestell durchgeführt. Ferner wurden jeweils drei verschiedene Laufparametersätze (für langsames, mittelschnelles und schnelles Laufen) verwendet, um die Varianz bei verschieden hohen Geschwindigkeiten und den dafür verantwortlichen verschiedenen Laufbewegungen zu bestimmen.



(a) Für langsames Laufen.

(b) Für mittelschnelles Laufen.

Abbildung 7.1: Boxplots der Messwerte-Varianz des Simulators.

Zur Darstellung der Messgrößen-Varianz werden hier „Boxplots“ verwendet. Dabei handelt

es sich um ein Diagramm, in dem die Streuung der Werte dargestellt wird. Die „Box“, das Rechteck im Diagramm, umfasst dabei genau die Hälfte aller Messwerte. Die Ränder der Box nach oben und unten geben den höchsten, bzw. niedrigsten Wert dieser Box, sie werden „Quartile“ genannt. Es handelt sich also zusammen mit dem Median um die Grenzen zwischen den einzelnen Vierteln der Messwerte. Die Linie in der Mitte der Box gibt den Median an, die gestrichelten Linien (die „Whisker“) zeigen den Bereich der beiden fehlenden 25% der Messwerte an. Falls es Werte gibt, welche weiter als 1,5 mal von den Quartilen entfernt sind, als der Abstand der beiden Quartile zueinander beträgt, so werden diese „Ausreißer“ jenseits der Whisker-Grenzen eingezeichnt. Ein Boxplot ermöglicht es dadurch, einen einfachen Überblick über die Struktur der gemessenen Werte zu erhalten, man erkennt leicht, wie stark die Messwerte streuen und wie diese Streuung geartet ist.

7.1.1 Messfehler des Simulators

Für die Bestimmung des Messfehlers im Simulator wurden die gleichen Voraussetzungen geschaffen, die auch für die Optimierung der Laufparameter galten. So lief der Roboter eine Strecke von 450 mm; für diese Strecke wurde die Zeit gemessen. Die Fitness der Individuen wurde aus der Geschwindigkeit, hier in [mm/s], bestimmt. Dieser Wert wurde mit demselben Parametersatz 50 Mal hintereinander ermittelt. Dies wurde für drei verschiedene Laufgeschwindigkeiten durchgeführt, und zwar für „langsame“ Geschwindigkeit mit dem initialen Parametersatz, der Geschwindigkeiten um etwa 25 mm/s ermöglichte, mit einem Parametersatz für „mittlere“ Geschwindigkeiten um 55 mm/s und mit einem für „schnelles“ Laufen um 110 mm/s. Abbildung 7.1(a) zeigt als Boxplot die Messwerte für den „langsamen“ Parametersatz, hier divergieren die Hälfte der Messungen um weniger als 1%, der höchste Messwert ist lediglich um 2% größer als der niedrigste.

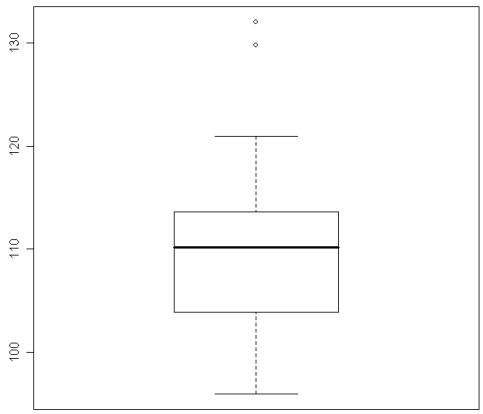


Abbildung 7.2: Boxplot der Messwerte-Varianz des Simulators für schnelles Laufen.

In Abbildung 7.1(b) ist die Varianz der Messwerte für mittlere Geschwindigkeiten (um 55 mm/s) dargestellt; auch hier sind die Abweichungen nur minimal, die Hälfte der Messwerte unterscheidet sich kaum mehr als um ein Prozent, auch die Extreme sind nicht weiter als zwei

Prozent des niedrigsten Messwertes voneinander entfernt. Anders sieht dies allerdings für hohe Geschwindigkeiten (Abbildung 7.2) aus: Hier streuen die Werte erstaunlich stark. Bei einem Median von 110 mm/s befinden sich die Quartile bei 104, bzw 114, was einer Abweichung von knapp 4% entspricht. Die Grenzen der Whisker liegen sogar bei 95 und 122, was einer Toleranz von bis zu knapp 14% des Medians entspricht. Der größte Ausweicher liegt sogar bei 132, was eine Abweichung von 20% bedeutet.

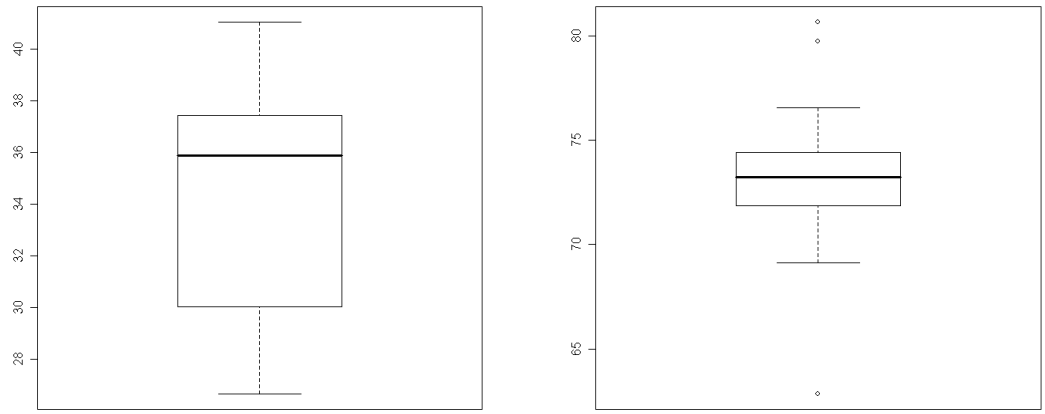
Hier bleibt festzuhalten, dass nicht nur die durchschnittliche Geschwindigkeit des Laufens mit einem Parametersatz den Messfehler bestimmt. Auch die Art des durch den gewählten Parametersatz resultierenden Laufens ist ausschlaggebend für die Abweichung der Messwerte. So ist ein Laufen, bei dem sich der Roboter stark aufschaukelt und somit nicht ständig mit dem gewünschten[1] Fuß den Boden berührt, sehr unregelmäßig und kann daher die Messung beeinträchtigen. Der Messfehler lässt sich in gewissen Grenzen minimieren, indem die Laufstrecke erhöht wird. Dies hat jedoch den gravierenden Nachteil, dass die sowieso schon lange Zeit für die Bewertung eines Laufparameters von 30-120 Sekunden (anbhängig von der Laufgeschwindigkeit des gerade zu messenden Parametersatzes) noch weiter steigen würde, was eine verlängerte Optimierzeit zur Folge hätte. Die hier benutzten 450 mm stellen somit einen Kompromiss zwischen Optimierzeit und Messgenauigkeit dar. Ganz verhindern lässt sich der Messfehler nicht, umsomehr sind Optimierverfahren gefragt, die mit verrauschten Fitness-Werten umgehen können.

7.1.2 Messfehler des Laufgestells

Auch für das Laufgestell und den realen Roboter wurde die Varianz der Geschwindigkeits-Messung bestimmt. Hier wurden abermals 50 Messungen des jeweils selben Parametersatzes durchgeführt, jeweils auch für die drei Geschwindigkeiten „langsam“ (um 35 mm/s), „mittelschnell“ (um 75 mm/s) und „schnell“ (um 110 mm/s). Wie bei der Optimierung der Laufparameter wurde hier eine Strecke von 70 cm gelaufen, für die über die Lichtschranken die Zeit bestimmt wurde. Auch hier wurden die Messwerte in [mm/s] gemessen.

Abbildung 7.3(a) zeigt einen Boxplot für die Messung des „langsamen“ Parametersatzes. Hier liegt der Median bei etwa 36 mm/s, die Quartile liegen bei 30 und $37,5$ mm/s. Dies entspricht Abweichungen von bis zu 20%, bzw 14%. Hier ist schon zu erkennen, dass die Abweichungen der Messwerte nach unten wesentlich breiter gestreut sind, als nach oben. Das untere Whisker beginnt bei 27 und der obere endet bei 41, was Abweichungen von maximal 25% entspricht. Die Abweichungen und vor allem die breitere Streuung nach unten lässt sich mit der Auswirkung des gewählten Parametersatzes auf das Laufverhalten erklären. Der gewählte Laufparameter bedingte ein sehr geringes Anheben der Roboter-Füße, wodurch der Robter häufig stolperte, bzw. sich teilweise leicht auf der Stelle drehte. Da dies unregelmäßig auftrat, wurden die Messwerte eben stark getreut. Da ein einmaliges Stolpern oft weiteres Stolpern nach sich zog, variieren besonders die niedrigen Messwerte stark. Wenn der Roboter mit nur wenigen Stolpern die Laufstrecke überwand, so lagen die Messwerte tendenziell näher beieinander.

[1] Mit „gewünschtem Fuß“ ist derjenige Fuß gemeint, der in der betreffenden Phase des Schrittes Bodenkontakt haben sollte, während der andere Fuß den oberen Teil der Trajektorie abfährt und sich somit in der Luft befindet.



(a) Für langsames Laufen.

(b) Für mittelschnelles Laufen.

Abbildung 7.3: Boxplots der Messwerte-Varianz des Laufgestells.

Abbildung 7.3(b) zeigt einen Boxplot der Messwerte für ein „mittelschnelles“ Laufen im Laufgestell, hier liegen die Messwerte sehr eng beieinander. Dies ist hier wieder vor allem mit der Auswahl des Lauf-Parametersatzes zu erklären. Der verwendete Parametersatz sorgte nicht nur für mittlere Laufgeschwindigkeiten um 75mm/s, sondern erzeugte auch ein sehr ausgeglichenes und stabiles Laufmuster. Der Roboter hob die Füße hoch genug, um nicht ein einziges Mal zu stolpern, auch wurde der Oberkörper mit einer Amplitude ausgelenkt, dass der Roboter nicht zu stark schwankte. So liegen bei diesen Messungen die Messwerte allesamt im Bereich $[69; 77]$, was bezogen auf den Median einer Abweichung von lediglich $\pm 5,5\%$ entspricht. Ferner sind die Werte sehr symmetrisch um den Median gruppiert, was mit der Gleichmäßigkeit der Laufbwegung zu erklären ist. In Abbildung 7.4 schließlich sind die Messwerte des „schnellen“ Parametersatzes als Boxplot dargestellt. Hier ist wieder eine stärkere Abweichung zu beobachten; die Laufbewegung war gekennzeichnet von einem gelegentlichem Aufschaukeln, was für ein Abbremsen der Geschwindigkeit sorgte. Die Bandbreite der gemessenen Werte reicht von $82 - 131$mm/s, was einer Abweichung von $\pm 25\%$ vom Median entspricht.

So wird auch hier deutlich, dass die Varianz der Messfehler weniger von der Laufgeschwindigkeit als viel mehr von der Art der Laufbewegung abhängt. Stolpern, Aufschaukeln oder auch eine nicht geradlinige Bewegung sorgt für Varianzen in der Messung, die dem verwendeten Optimier-Algorithmus falsche Hinweise auf die Form der dem Optimierproblem inneliegenden Funktion gibt. So ist es besonders wichtig, einen Algorithmus zu verwenden, der nicht an einmal gemessenen Werten für einen Parametersatz festhält, sondern Fehler akzeptiert und gegebenenfalls wiederholt oder nach einiger Zeit sogar gänzlich verwirft.

7.2 Die Laufoptimierung im Simulator

Um die in Abschnitt 5.3.2 vorgestellten vier Varianten der Evolutionsstrategien miteinander zu vergleichen und um zu prüfen, wie gut sich diese Art der Optimier-Algorithmen überhaupt

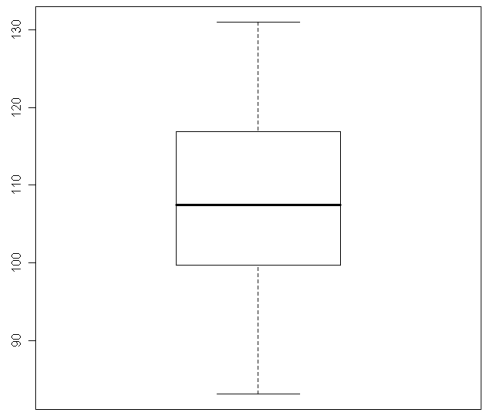


Abbildung 7.4: Boxplot der Messwerte-Varianz des Laufgestells für schnelles Laufen.

auf das Problem der Optimierung der WalkingEngine-Parameter anwenden lässt, wurden mit dem BreDoBrothers-Simulator Tests durchgeführt. Diese Untersuchungen wurden lediglich durchgeführt, um die Algorithmen zu bewerten und einen Einblick in die Machbarkeit der Parameter-Optimierung für das Laufen des Kondo KHR-1 zu erhalten. Bereits nach den ersten Tests mit den Optimier-Algorithmen war klar, dass die so gewonnenen Parameter für die WalkingEngine zwar im Simulator zu recht hohen Geschwindigkeiten (bei etwa 12-14 cm/s schon nach wenigen Generationen im Gegensatz zu 2,5 cm/s bei den initialen Parametereinstellungen) führten, jedoch war keiner dieser Parametersätze für den realen Kondo KHR-1 verwendbar. Im Simulator wird eine glatte, ebene Oberfläche des Bodens modelliert, was es dem simulierten Roboter ermöglicht, seine Füße auch über den Boden schleifen zu lassen. So entwickelten sich Laufparameter, welche zumindest kurzzeitig die Füße leicht über den Boden rutschen ließen.

Dieses Schleifen funktioniert auf dem realen Teppichboden, welcher beim Robocup für die Roboter-Fußballspiele benutzt wird, jedoch nicht. Die Fußkanten des Roboters verhaken sich in Faserschlingen, wodurch der Roboter stolpert. Ferner federt der Teppichboden leicht, was sich auf die Bewegungen der Roboter-Teile auswirkt. Es ist für den Teppichboden, für den das Laufen in dieser Arbeit ja optimiert werden sollte, also eine andere Einstellung der Laufparameter notwendig als für den Simulator. Eine Anpassung des Simulators dergestalt, dass ein Teppichboden nachgebildet wird, wäre im Rahmen dieser Arbeit zu komplex gewesen. Zudem bleibt anzuzweifeln, ob eine solche Modellierung die Realität exakt genug nachbilden könnte.

Der Simulator wurde nun benutzt, um die Optimierung mit verschiedenen Algorithmen zu testen und um die Einstellungen der Laufparameter für verschiedene Laufrichtungen (gemeint ist hier die translatorische Richtung, siehe Abschnitt 4.4) zu ermitteln. Bei der in Abschnitt 7.1 beschriebenen Laufstrecke von 450 mm im Simulator wird, je nach Geschwindigkeit des aktuellen Parametersatzes, eine Zeit von 30-60 Sekunden benötigt, da alle Bewegungen im Simulator auf aktuellen Rechnerplattformen nur etwa halb so schnell ablaufen wie in der Realität. So wird schnell klar, dass die Evolution der Laufparameter auch im Simulator sehr lange

Zeit in Anspruch nimmt und nicht in einigen Minuten erfolgen kann.

Parametername	**Wert**	**Mutationsschrittweite σ_i**
StepDuration	1500	150
StepHeight	18	1
StepLength	45	4
armAmplitudeX	0,5	0,1
armAmplitudeY	0,5	0,1
BodyAmplitudeX	0,1	0,01
BodyAmplitudeY	0,17	0,03
YOffset	20	2
ZOffset	25	2
FootTiltX	0	0,01
FootTiltY	0	0,01

Tabelle 7.1: Die initialen Parameter-Werte für die Optimierung.

Es sollten nun insgesamt vier Algorithmen getestet werden, ferner sollten mehrere Läufe pro Algorithmus stattfinden, da aufgrund der Verwendung von Zufallszahlen im Algorithmus zum Teil unterschiedliche Ergebnisse entstehen können. Geht man nun von fünf unabhängigen Läufen pro Algorithmus aus, werden für die vier Algorithmen insgesamt 20 unabhängige Algorithmen-Läufe benötigt. Wenn man nun einmal lediglich 10.000 Funktionsauswertugen für die Optimierung annimmt (bei der (5,30)-ES wären dies nur 333 Generationen), so erhält man insgesamt 200.000 Funktionsauswertungen, was bei einer Zeit von 30-60s pro Auswertung einem Rechenaufwand von etwa 70-140 Tagen entspricht. Dies wäre vom Zeitaufwand für diese Arbeit viel zu lange gewesen, da auch noch die Untersuchung der Parameter für verschiedene Laufrichtungen erfolgen sollte.

So wurde der Rechner-Pool des Institutes für Datenverarbeitungssysteme vom Fachbereich Elektrotechnik und Informationstechnik der Universität Dortmund genutzt, um diese Berechnungen zu parallelisieren. Da alle einzelnen Algorithmenläufe unabhängig voneinander sind, konnte jeder einzelne Lauf auf einem eigenen Rechner durchgeführt werden, so führten für das oben beschriebene Beispiel 20 PCs jeweils etwa eine Woche die benötigten Simulationen zur Optimierung durch. So war es möglich, mehrere verschiedene Tests durchzuführen, bei Fehlern[2] war nicht gleich das Ergebnis von Monaten der Berechnung umsonst gewesen.

Für alle Simulationen sowie für die Optimierung der Parameter mit dem Roboter im Laufgestell wurde jeweils derselbe initiale Parametersatz benutzt, welcher durch Experimente „von Hand“ entwickelt wurde. Hier wurde auf die Erzeugung zufälliger Werte für die Parameter verzichtet, da dies mit großer Wahrscheinlichkeit nur zu Parametern geführt hätte, durch die der Roboter beim Laufen umgefallen wäre. Dadurch hätten die benutzten Evolutionsstrategien keine Informationen über die Güte der Parameter bekommen (Parametersätze, die zu einem Umfallen führen, werden ja stets mit Fitness 0 bewertet), die Optimierung wäre in der Anfangsphase zu einer zufälligen Suche verkommen. Wenn dann nach einiger Zeit durch Zufall

[2]So gab es zunächst noch einige Probleme mit der Stabilität des Simulators, welchen z.B. durch ein speziell dafür implementiertes Watchdog-Programm begegnet werden konnte.

lauffähige Parameter gefunden wären, so wären durch die zufällige Suche zuvor die Mutationsschrittweiten stark degeneriert, so dass erst nach langer Zeit die Optimierung wie gewünscht verlaufen wäre.

Ferner wurden noch die Mutationsschrittweiten vorgegeben, für die (1+1)-ES war dies der Wert 0,1 (siehe Abschnitt 5.3.2.1). Für alle anderen Evolutionsstrategien, die ja alle eine Selbstadaptation der Mutationsschrittweiten durchführten, wurden für alle $i \in \{1..n\}$ die Werte für die Mutationsschrittweiten σ_i so voreingestellt, dass sie in ihrer Auswirkung der Empfindlichkeit des jeweiligen Parameters angemessen waren. Dies wurde analog zu den Skalierungs-Faktoren aus Abschnitt 5.3.2.1 durchgeführt. Die für die Simulation und für die Optimierung mit dem realen Roboter verwendeten Parameterwerte nebst Mutationsschrittweiten sind in Tabelle 7.1 aufgeführt.

Diese Werte sorgten nicht nur für eine funktionierende Laufbewegung, sondern lagen zudem innerhalb der vorgegebenen Wertebereiche und nicht auf den Rändern. So wurden bei ersten Versuchen die Arm-Amplituden auf die Werte 0 oder 1 gesetzt, was dazu führte, dass viele letale Individuen erzeugt wurden, da ja mit Wahrscheinlichkeit 1/2 eine Mutation den Parameterwert über diese Grenze setzt.

7.2.1 Vergleich der Algorithmen

Für den Vergleich der vier Algorithmen *(1+1)-ES*, *(1,30)-ES*, *(5,30)-ES* und *(5+30)-ES mit* $\kappa = 3$ wurde eine Optimierung der Laufparameter im Simulator für jeden dieser Algorithmen auf je fünf PCs für jeweils eine Woche ununterbrochen durchgeführt. Je nach Verlauf der Optimierung, nach Verzögerungen durch z.B. automatische Restarts der Simulationen durch den Watchdog nach Abstürzen wurden unterschiedlich viele Funktionsauswertungen durchgeführt, allerdings alle im Bereich von 10.000. Da sich die Ergebnisse hinsichtlich des Fitness-Verlaufs nicht gravierend unterschieden, folgen im weiteren nur Dokumentationen „typischer Läufe“. Die folgenden Ergebnisse beziehen sich alle auf eine Laufstrecke von 450 mm im Simulator bei einem Translationswinkel von 0°, also geradeaus nach vorne.

Die (1+1)-Evolutionsstrategie schnitt wie zu erwarten schlecht ab. Dabei waren nicht einmal zu hohe Messfehler das Problem, bei denen das Individuum irrtümlicherweise als zu gut beurteilt wurde. Abbildung 7.5 zeigt einen typischen Lauf der (1+1)-ES. Dabei ist zu erkennen, dass die Fitness (rote Linie) bei 107 mm/s stagniert, obwohl später bei den anderen Algorithmen Läufe erreicht werden, bei denen sich der Roboter mit nachweislich tatsächlichen 140 mm/s bewegt. So kann davon ausgegangen werden, dass die Evolutionsstrategie in einem lokalen Optimum feststeckte.

Dies ist weiter in Abbildung 7.5 daran zu erkennen, dass sich die Nachkommen-Fitness (die grünen Punkte) immer wieder in die Region der Elter-Fitness gelangen, diese aber ab Generation 5000 nicht weiter verbessern kann. Die vielen Nachkommen mit Fitness -1, welche als teilweise unterbrochene Linie im Diagramm zu erkennen sind, sind die Individuen, welche aufgrund von Überschreitungen der Parametergrenzen als letal eingestuft wurden. Der blaue Graph in Abbildung 7.5 zeigt den Verlauf der Mutationsschrittweite; hier ist das Wellenmuster, basierend auf dem in Abschnitt 5.3.2.1 beschriebenen Reset der Mutationsschrittweiten zu erkennen. Bei den höheren Generationen, in denen sich die Populations-Fitness kaum noch

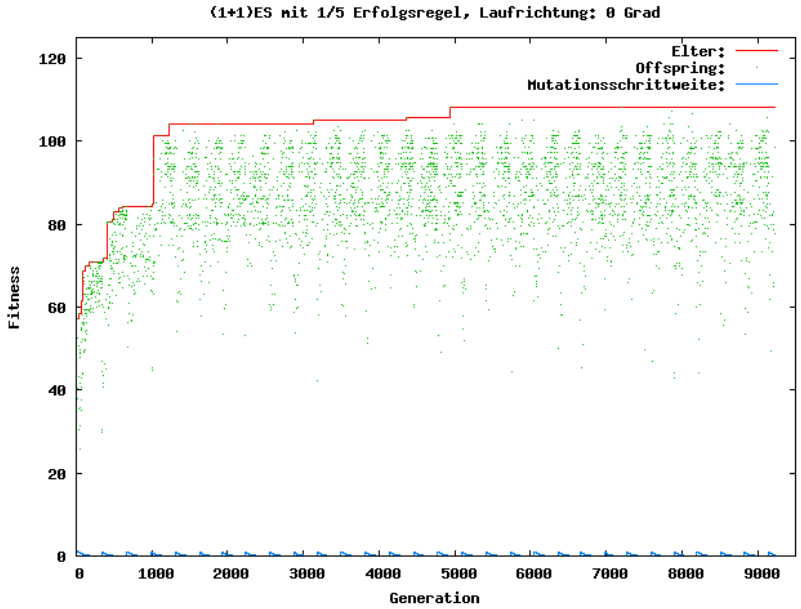


Abbildung 7.5: Verlauf der Fitnesswerte der (1+1)ES im Simulator.

ändert, ist dies zu verstehen, da ja bei ausbleibenden Erfolgen die Mutationsschrittweiten abgesenkt werden. Hier handelt es sich dabei um ein vom Algorithmus vermutetes Optimum. Der Reset der Mutationsschrittweiten zwischen Generation 1 und 200 ist trotz des offensichtlichen starken Erfolges dadurch zu erklären, dass häufig letale Individuen erzeugt werden, welche die Erfolgshäufigkeit stark drücken.

Die (1,30)-Evolutionsstrategie funktioniert im Gegensatz zur (1+1)-ES schon wesentlich besser, da hier ja auch zugunsten der Komma-Selektion auf die Plus-Selektion verzichtet wird. Dadurch können sich falsch gemessene Individuen nur für eine Generation in der Population halten; die Messfehler wirken sich also nicht so lange auf die Optimierung aus. Abbildung 7.6 zeigt einen typischen Lauf, wobei deutlich die Lücke mit Fitness 0 (die Fitness des Elter wird durch die rote Kurve dargestellt) zwischen Generation 170 und 248 ins Auge springt. Dieses Phänomen trat bei allen fünf Läufen der (1,30)-ES auf (wenn auch jeweils zu anderen Zeitpunkten) und ist folgendermaßen zu erklären: Da in der Population immer nur ein Individuum existiert und eine Komma-Selektion durchgeführt wird, ist es möglich, dass alle Nachkommen des Elter aufgrund der Unfähigkeit zu laufen eine Fitness von 0 bekommen. Aus diesen Nachkommen muss dann das nächste Eltern-Individuum gewählt werden, welches nun auch wieder Parameter hat, die nicht zum Laufen geeignet sind.

Bei der Komma-Strategie mit $\mu > 1$ ist dieses Problem unwahrscheinlicher, da die Diversität in der Population eine größere Wahrscheinlichkeit erzeugt, dass die Nachkommen lauffähig sind. Ferner kann es bei $\mu = 1$ dazu kommen, dass das Elter-Individuum Parameter besitzt, welche gerade eben noch ein Laufen ermöglichen, aber eine leichte Variation dieser Parameter dann mit hoher Wahrscheinlichkeit zu laufunfähigen Nachkommen führen. Schon aus diesem Grund ist die (1,30)-ES nicht empfehlenswert, jedoch wurde in Abschnitt 5.3.2.2 ja schon erläutert, dass dieser Algorithmus nur der Vollständigkeit halber benutzt wurde. Die Ergebnisse hier zei-

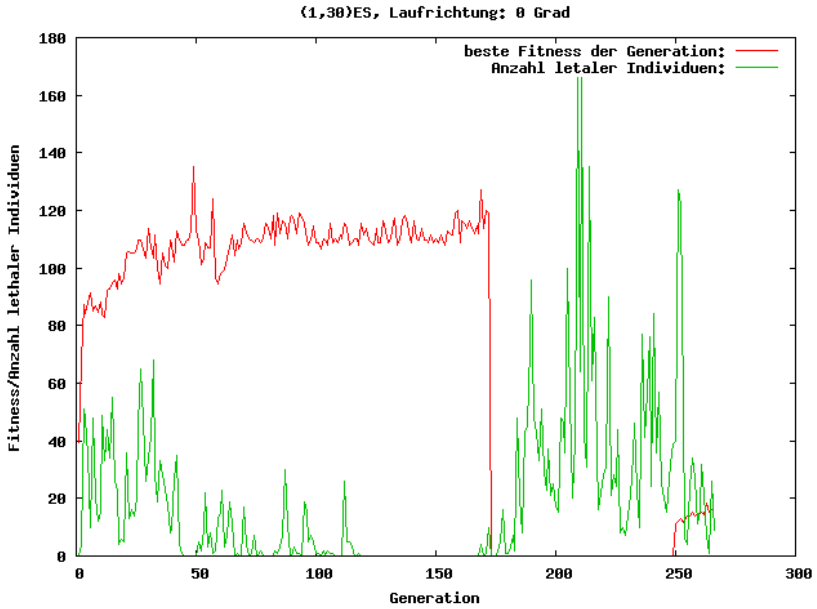


Abbildung 7.6: Verlauf der Fitnesswerte der (1,30)ES im Simulator.

gen somit auch anhand empirischer Daten, dass eine Population der Größe eins hier zu klein ist.

Dieses Phänomen des Fitnesswertes 0 über einen längeren Zeitraum entspricht übrigens in etwa dem beschriebenen für die zufällige Initialisierung der Laufparameter beim Algorithmen-Start. Da alle Nachkommen die Fitness 0 haben, wird aus der Evolutionsstrategie eine zufällige Suche, weshalb ein lauffähiges Individuum auch nur durch einen Zufall gefunden werden kann. Die Adaptation der Strategieparameter funktioniert nicht mehr wie gewünscht, da es keinen deterministischen Auswahlkriterien mehr gibt. Das Ergebnis besteht unter Anderem darin, dass die Anzahl der letalen Individuen aufgrund steigender Mutationsschrittweiten steigt, wie in Abbildung 7.6 an der grünen Kurve zu erkennen ist.

Wesentlich besser sieht nun der Fitnessverlauf eines typischen Laufes der (5,30)-Evolutionsstrategie, dargestellt in Abbildung 7.7, aus. Hier ist in der roten Kurve die Fitness des jeweils besten Individuums der Generation dargestellt, die grüne Kurve zeigt die Anzahl der letalen Individuen, welche erzeugt und wieder verworfen wurden, bis nur noch nicht-letale Nachkommen in der Generation existierten. Trotz starker Schwankungen der einzelnen Fitness-Werte ist der Trend steigender Fitnesswerte zu erkennen. So sind die schwankenden Werte zum Teil durch Messfehler zu erklären (dies trifft in besonderem Maße für den Wert von 142 mm/s in Generation 105 und für die 105 mm/s in Generation 310 zu), jedoch auch durch die starke Empfindlichkeit des Laufverhaltens auf Parameter-Änderungen.

Hier wird nun auch ersichtlich, welchen Vorteil eine Populations-Größe von $\mu > 1$ hat: in keinem der fünf Läufe der (5,30)-ES kam es zu Phasen, in denen die gesamte Population aus Individuen bestand, die nicht lauffähig waren und deshalb einen Fitnesswert von 0 bekamen. Es herrschte also eine ausreichende Diversität, so dass ständig mit Individuen operiert wurde, welche ein Laufen ohne Umfallen ermöglichten. Dies ist ein Effekt, der tatsächlich von der Grö-

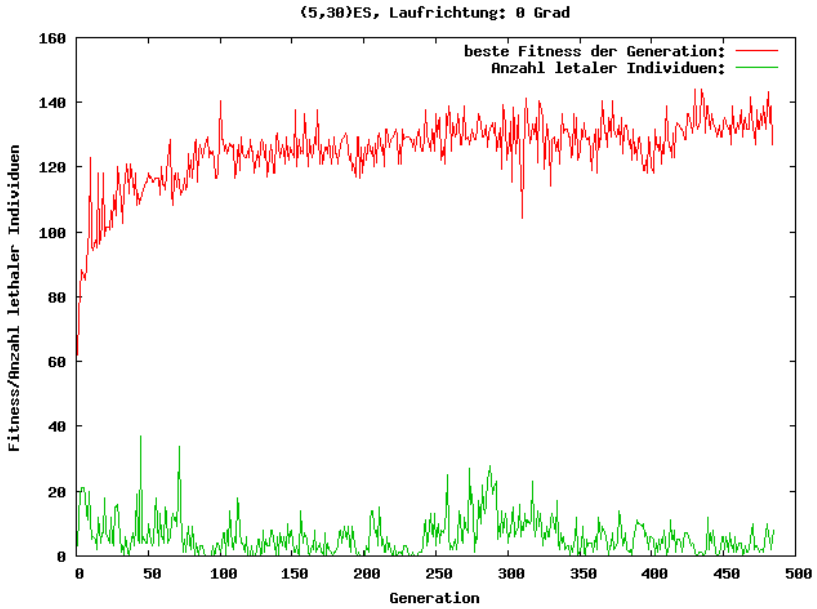


Abbildung 7.7: Verlauf der Fitnesswerte der (5,30)-ES im Simulator.

ße der Eltern-Population abhängt und nicht von λ, der Größe der Nachkommen-Population, da ja hier genauso wie bei der (1,30)-ES $\lambda = 30$ war.

Ferner wird in dieser Grafik deutlich, dass die mittleren Fitness-Werte der (5,30)-ES gesamt deutlich über denen der (1,30)-ES liegen. Dies traf auf alle fünf Läufe der (5,30)-ES zu, in allen Fällen war diese Variante mit der größeren Eltern-Population überlegen.

Abbildung 7.8 zeigt schließlich den Fitnessverlauf der (5+30)-Evolutionsstrategie mit $\kappa = 3$. Die rote Kurve zeigt auch hier die Fitness des fittesten Individuums der jeweiligen Generation, die grüne Kurve bezeichnet die Anzahl letaler Individuen der Generation. Hier fällt vor allem der stufige Verlauf der Fitness-Werte auf; hier ist zu sehen, dass sich die Individuen mit großer Fitness meist über volle drei Generationen in der Population hielten, bis sie aufgrund des maximalen Alters entfernt wurden. Ferner sind auch etliche hohe Ausschläge zu erkennen, welche eine Fitness von teilweise über 200 mm/s aufweisen. Diese hohen Werte sind durch Messfehler zu erklären; die tatsächliche Fitness liegt meist um 150 mm/s. So ist hier schon vorstellbar, dass der Wert von κ nicht sehr viel weiter über drei gesteigert werden sollte, da sonst die Optimierung in Bereiche läuft, in denen zu lange mit falschen Werten operiert wird.

Trotz dieser länger beibehaltenen Messfehler errreicht die (5+30)-ES hier im Schnitt höhere Fitnesswerte als die anderen untersuchten Algorithmen, dies gilt auch für die tatsächlichen Werte der Fitness. So wurden hier tatsächliche Fitnesswerte von 150 mm/s erreicht, was sonst kein Algorithmus erreichte. Von daher wurde diese (5+30)-ES auch für die endgültige Optimierung der WalkingEngine-Parameter im Laufgestell mit dem realen Roboter ausgesucht.

Dass trotz der relativ niedrigen Generationen-Anzahl von knapp über 500 das Optimierverfahren zumindest ein lokales Optimum erreicht hat, zeigt die Analyse der Parameter-Werte sowie

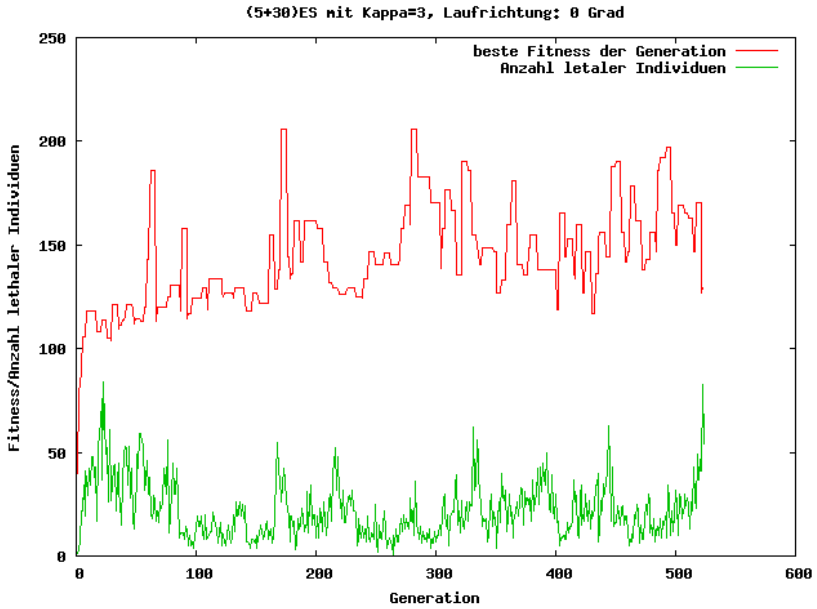


Abbildung 7.8: Verlauf der Fitnesswerte der (5+30)ES mit $\kappa = 3$ im Simulator.

deren Mutationsschrittweiten. In den Abbildungen 7.9 und 7.10 sind die Verläufe zweier Parameter sowie deren Mutationsschrittweiten σ_i über die Generationen dargestellt. Abbildung 7.9 zeigt exemplarisch den Verlauf des Parameters $ZOffset$, welcher den Roboter-Oberkörper um die angegebene Höhe in mm absenkt, indem der Roboter in die Knie geht. Hier ist zu erkennen, dass bereits nach 100 Generationen keine großen Schwankungen mehr auftreten, nach 450 Generationen bleibt der Wert mit knapp 14 mm nahezu konstant. Auch die Mutationsschrittweiten sinken auf nahezu Null herab, der Algorithmus hat für diesen Parameter ein lokales Optimum gefunden.

Ähnlich sieht es für den Parameter *BodyAmplitudeX* aus, welcher den Oberkörper nach vorne wippen lässt. Dessen Verlauf ist in Abbildung 7.10 dargestellt, wieder mit den zugehörigen Mutationsschrittweiten aufgetragen über die Generationen. Innerhalb der ersten 150 Generationen gibt es Schwankungen um den initialen Wert von 0,1 zwischen 0,057 und 0,131, jedoch wird der Wert danach stabil bei 0.096. Auch die Mutationsschrittweiten sinken dann stark ab auf nahe Null. So kann hier festgehalten werden, dass die Evolutionsstrategie diesen Parameter anfänglich variiert und nach relativ wenigen Generationen in einem engen Wertebereich festhält. Dasselbe gilt für nahezu alle der zu optimierenden Parameter, lediglich der Parameter *BodyAmplitudeY* schwankte anhaltend um den Wert 0,3 mit einer Varianz von 0,03, was aber auch nur einer Abweichung von maximal 10% entspricht.

Der Parameter *StepDuration* fand bereits nach gut 100 Generationen den endgültigen Wert von 715, was nur knapp über dem erlaubten Grenzwert von 700 lag. Die Mutationsschrittweite für *StepDuration* sank auf unter 0,2 ab, was bei den Werten um 700 keinen spürbaren Effekt mehr hat. Dieses Verhalten war aber auch zu erahnen, da ja die Schrittdauer direkt mit der Laufgeschwindigkeit korrespondiert. Lediglich für den Fall, dass eine solch kurze Schrittdauer zu Stürzen geführt hätte, wäre die Schrittdauer länger gewesen.

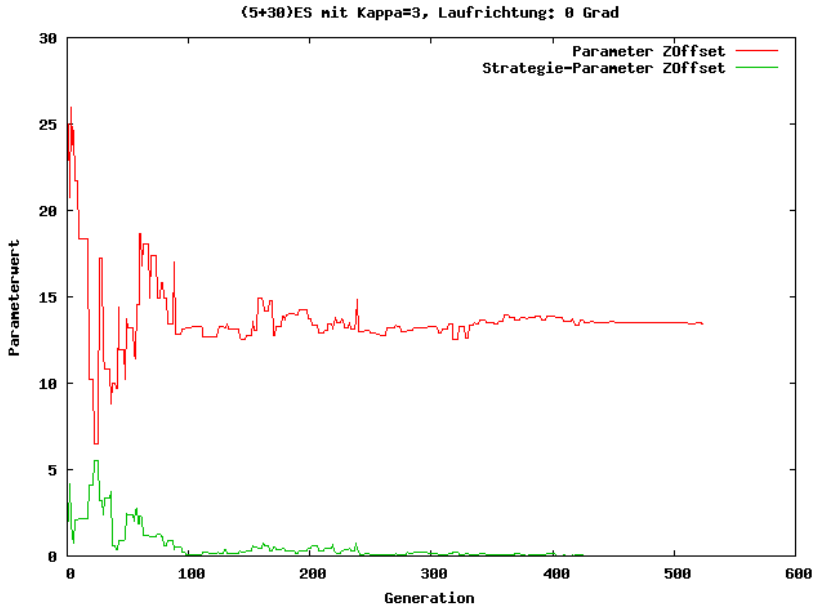


Abbildung 7.9: Darstellung der Verläufe des Parameters $ZOffset$ und dessen Mutationsschrittweite aus der Optimierung eines Geradeauslaufens im Simulator mit der (5+30)-ES mit $\kappa = 3$.

So konnte hier nun gezeigt werden, dass bereits nach den nur gut 500 Generationen der Evolutionsstrategie, welche aufgrund der langen Laufzeit der Fitnessmessungen nicht weiter erhöht werden konnten, das Optimierverfahren ein lokales Optimum gefunden hatte. Die Parameterwerte konvergierten, die Mutationsschrittweiten ließen zunächst keine größere Variation mehr zu. Dass nun trotzdem die Fitnesswerte der zugehörigen Individuen so stark schwanken, ist ja einerseits mit den verrauschten Messwerten zu erklären, jedoch zeigt der Roboter auch eine Empfindlichkeit gegen leichte Parameteränderungen. So kann eine leichte Variation bestimmte Parameter (hier sei vor allem *BodyAmplitudeY* genannt) das Laufverhalten stark beeinflussen.

Der Parametersatz, welcher im Simulator die reproduzierbar schnellste Laufgeschwindigkeit von 170 mm/s bedingte, hatte die in Tabelle 7.2 angegebenen Parameterwerte. Es ist zu erkennen, dass sich die Parameter *StepHeight*, *ArmAmplitudeX*, *ArmAmplitudeY*, sowie *BodyAmplitudeX* kaum von den initialen Parameterwerten unterscheiden. Dagegen gibt es bei den anderen Parametern zum Teil große Unterschiede. Vor allem die Schrittgeschwindigkeit *StepDuration* sowie die Schrittlänge *StepLength* veränderten sich stark, was jedoch nicht weiter verwundert, da ja eine schnelle Schrittfolge nebst großen Schritten zwangsläufig zu schneller Geschwindigkeit führen muss, solange der Roboter nicht stürzt. Um dies zu verhindern, wird mit dem durch die Optimierung erhöhten Parameter *BodyAmplitudeY* das Gleichgewicht gehalten, indem sich der Roboter stark zur Seite lehnt.

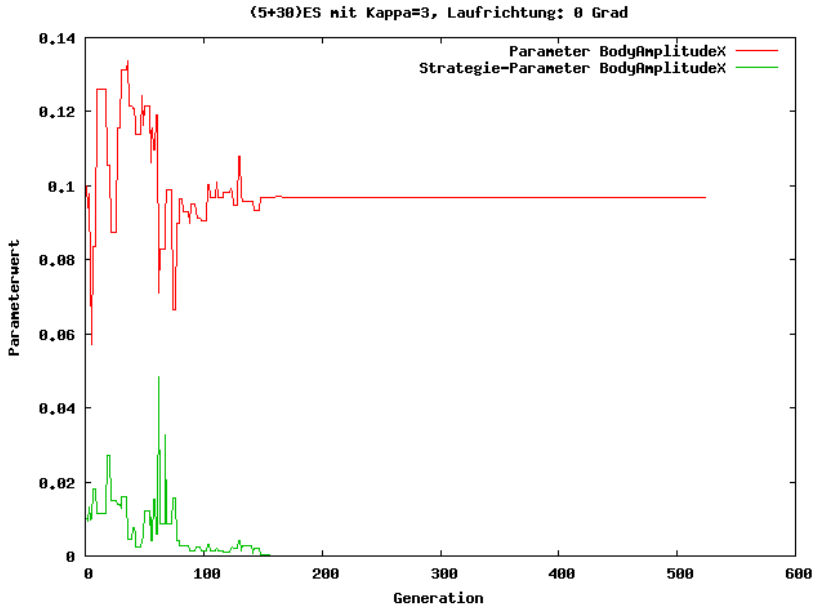


Abbildung 7.10: Darstellung der Verläufe des Parameters *BodyAmplitudeX* und dessen Mutationsschrittweite aus der Optimierung eines Geradeauslaufens im Simulator mit der (5+30)-ES mit $\kappa = 3$.

7.2.2 Parameter für verschiedene Laufrichtungen

Bisher wurde die Optimierung lediglich für ein Geradeauslaufen durchgeführt. Jedoch ist die hier vorgestellte WalkingEngine auch in der Lage, den Roboter translatorisch in beliebige Richtungen zu bewegen. So stellt sich nun die Frage, ob die bisher ermittelten Parameter auch für unterschiedliche Translationswinkel eine schnellstmögliche Bewegung erzeugen oder ob für jeden Translationswinkel α ein eigener optimierter Parametersatz vonnöten ist.

Zur Untersuchung, wie die Laufparameter für verschiedene translatorische Laufrichtungen zu wählen sind, wurde wieder eine Simulation durchgeführt. Dafür wurden mit der (5+30)-Evolutionsstrategie mit $\kappa = 3$ die Laufparameter für neun verschiedene Translations-Winkel hinsichtlich maximaler Geschwindigkeit optimiert, die Optimierung wurde pro Laufrichtung jeweils 500 Generationen lang durchgeführt. Die Mutationsschrittweiten lagen nach diesen 500 Generationen für alle Laufrichtungen nahe Null, so dass von einer Konvergenz bezüglich eines zumindest lokalen Optimums ausgegangen werden kann. Von der Optimierung war also keine weitere Verbesserung der Laufparameter zu erwarten.

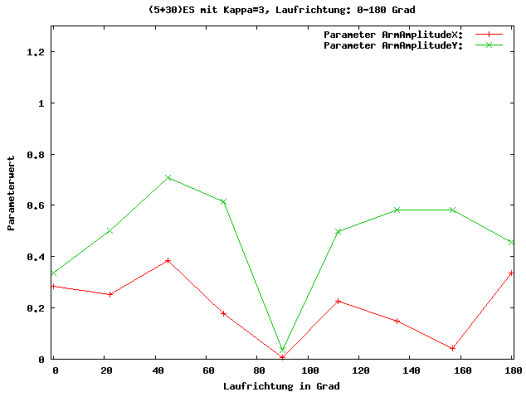
Die Laufrichtungen waren dabei: $0°$, $22,5°$, $45°$, $67,5°$, $90°$, $112,5°$, $135°$, $157,5°$ und $180°$. Dies sind die Richtungen von geradeaus nach vorne über ein schräges Laufen nach vorne rechts und ein Seitwärtslaufen nach rechts bis schlussendlich geradeaus nach hinten. Translationswinkel von $> 180°$ wurden nicht gewählt, da der Kondo KHR-1 spiegelsymmetrisch gebaut ist, und sich dadurch die Ergebnisse für ein Laufen nach rechts genauso auf ein Laufen nach links übertragen lassen.

Der Parameter *StepDuration* lag nach der Optimierung für alle Laufrichtungen außer für ein

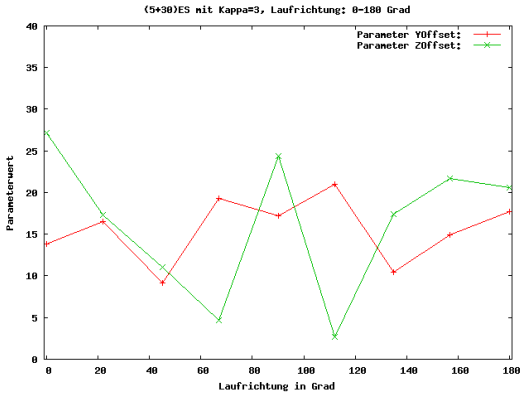
Parametername	Wert
StepDuration	715
StepHeight	16,9
StepLength	75
armAmplitudeX	0,22
armAmplitudeY	0,38
BodyAmplitudeX	0,096
BodyAmplitudeY	0,31
YOffset	8,4
ZOffset	13,4
FootTiltX	0,1
FootTiltY	0

Tabelle 7.2: Die im Simulator optimierten Parameter-Werte.

Laufen seitwärts nach rechts (90°) zwischen 720 und 705, also sehr nah bei dem niedrigsten erlaubten Wert für diesen Parameter und für alle Laufrichtungen annähernd gleich. Lediglich für den Winkel von 90° lag der Parameter bei 920, was zeigt, dass ein Seitwärtslaufen aufgrund der großen Trägheit des Roboters bei dieser wankenden Bewegung langsamer vonstatten gehen muss, als dies bei anderen Laufbewegungen der Fall ist. Die Parameter $FootTiltX$ und $FootTiltY$ regelten sich für alle Laufrichtungen auf annähernd gleiche Werte um 0,1 bzw. 0 ein, was erklärbar dadurch ist, dass diese Parameter nur die Gewichtsverteilung des Roboters regeln, also konstant gehalten werden können. Dass $FootTiltY$ bei 0 bleibt, ist durch die links-rechts-Symmetrie des Roboters erklärbar; der Wert von 0,1 für $FootTiltX$ erklärt sich durch das Gewicht der Controller-Platine auf dem Roboter-Rücken, von daher muss sich der Roboter leicht nach vorne beugen, um dieses Gewicht wieder auszugleichen.



(a) Verlauf der Parameter $ArmAmplitudeX$ und $ArmAmplitudeY$.



(b) Verlauf der Parameter $YOffset$ und $ZOffset$.

Abbildung 7.11: Der Verlauf der Parameter $ArmAmplitudeX$, $ArmAmplitudeY$, $YOffset$ und $ZOffset$ für verschiedene Laufrichtungen. Vergrößerungen dieser Abbildungen befinden sich im Anhang als Abbildung A.11 und Abbildung A.12.

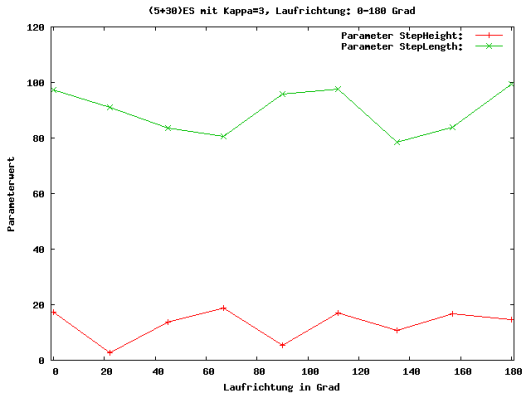
Die restlichen acht Parameter zeigten allerdings Unterschiede in Abhängigkeit von der Laufrichtung. In Abbildung 7.11(a) sind die Einstellungen der Parameter *ArmAmplitudeX* und *ArmAmplitudeY* für die jeweilige Laufrichtung dargestellt. Hier ist eine Spiegelsymmetrie bezüglich der Laufrichtung von 90° zu erkennen: bei einem Geradeauslaufen, egal ob nach vorne oder hinten, liegen beide Werte zwischen 0,35 und 0,45. Dagegen sinken sie beide bei einer Laufrichtung von 90°, entsprechend direkt seitwärts nach rechts, auf etwa null ab. Für ein Seitwärtslaufen scheinen beide Armbewegungen also eher hinderlich. Ansonsten ist generell ein Anstieg der Werte für ein diagonales Laufen (45° und 135°) zu erkennen. Es handelt sich hier nicht um perfekte Symmetrie, da die optimerten Werte ja Messungenauigkeiten und Toleranzen unterliegen, so dass keine perfekt gleichen Werte zu erwarten sind. Aber der Trend zur Abhängigkeit der Parameterwerte in Bezug auf die Laufrichtung ist klar erkennbar.

Ähnlich sieht es für die Parameter $YOffset$ und $ZOffset$ aus, wie in Abbildung 7.11(b) dargestellt. Auch hier ist eine Symmetrie um 90° erkennbar. Für ein direktes Seitwärtslaufen scheint eine größere Absenkung des Oberkörpers mittels $ZOffset$ sinnvoll zu sein, wogegen die Werte für diagonales Laufen wesentlich kleiner sein sollten. Für ein eher Geradeauslaufen wiederum steigen die Werte erneut an. Auch für $YOffset$ scheint der Werteverlauf symmetrisch zu sein, obwohl der Verlauf längst nicht so überzeugend scheint. Hier sollten weitere Untersuchungen und mehrfach wiederholte Optimierungen durchgeführt werden, um sicherzugehen, dass es sich nicht um unbedeutende Schwankungen handelt.

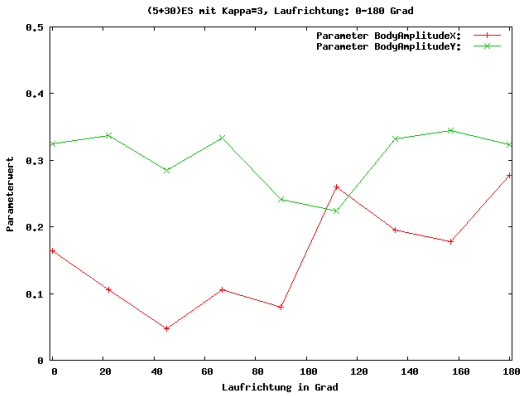
Der Verlauf der Parameter *StepHeight* und *StepLength* ist in Abbildung 7.12(a) dargestellt: hier sind die Änderungen in Bezug auf die Laufrichtung nicht so deutlich wie bei den bisher betrachteten Parametern. Jedoch ist auch hier sichtbar, dass für ein Seitwärtslaufen andere Werte benötigt werden, als für diagonales oder gerades Laufen. Leicht erklärbar ist das Absinken der Schritthöhe für Laufen in 90°-Richtung. Dort schaukelt sich der Roboter auf und führt so die seitwärtigen Schritte aus. Durch das Wippen kommen die Füße hoch genug vom Boden, so dass eine weitere Anhebung der Füße nicht erforderlich ist. Dies würde zusammen mit der weiten Auslenkung der Füße durch die dort gesteigerte Schrittweite auch zu einer zu starken Instabilität führen.

Abbildung 7.12(b) zeigt schließlich den Verlauf der Parameter *BodyAmplitudeX* und *BodyAmplitudeY*. Beide sind sehr unstetig, was sich nur für *BodyAmplitudeX* erklären lässt. Dieser Parameter wird ja in Abhängigkeit der Laufrichtung skaliert, so dass er für die Laufrichtungen 90° und 0° gleich Null ist. So ist der Verlauf von *BodyAmplitudeX* für diese beiden Laufrichtungen bei der Optimierung rein zufällig. Warum jedoch *BodyAmplitudeY* einen scheinbar willkürlichen Verlauf zeigt, kann hier nicht erklärt werden. Dazu würden sich weitere Untersuchungen anbieten.

So liegt nun die Idee nahe, dass für ein möglichst schnelles omnidirektionales Laufen die Parameter für einige Laufrichtungen auf dem realen Roboter optimiert werden. Für die Laufrichtungen, für die keine Parameterwerte explizit vorliegen, könnte dann interpoliert werden, wenn die genaue Abhängigkeit der Parameter von den Laufrichtungen bekannt ist. Die Abbildungen 7.11(a) bis 7.12(a) lassen ja den Schluss zu, dass der Verlauf der Parameter gewissen Gesetzmäßigkeiten folgt, weshalb auf deren Grundlage auch eine Interpolation möglich sein müsste. Für genauere Zusammenhänge und für das Problem mit den beiden Parametern *BodyAmplitudeX* und *BodyAmplitudeY* müssten jedoch noch mehr Untersuchungen



(a) Verlauf der Parameter *StepHeight* und *StepLength*.



(b) Verlauf der Parameter *BodyAmplitudeX* und *BodyAmplitudeY*.

Abbildung 7.12: Der Verlauf der Parameter *StepHeight*, *StepLength*, *BodyAmplitudeX* und *BodyAmplitudeY* für verschiedene Laufrichtungen. Vergrößerungen dieser Abbildungen befinden sich im Anhang als Abbildung A.13 und Abbildung A.14.

erfolgen, speziell müssten noch mehr unterschiedliche Laufrichtungen behandelt werden, zudem wäre zu empfehlen, diese Untersuchungen nicht im Simulator, sondern mit dem realen Roboter durchzuführen.

7.3 Die Optimierung im Laufgestell

Nachdem nun die (5+30)-Evolutionsstrategie mit $\kappa = 3$ als Algorithmus für die weitere Optimierung ausgesucht wurde, wurden die Laufparameter mit dem Kondo-KHR1 im Laufgestell optimiert. Dazu wurde der in Abschnitt 6.2 erläuterte Aufbau benutzt. Die initialen Parametersätze entsprachen denen aus Tabelle 7.1. Da der Aufwand für die Durchführung der Optimierung im Laufgestell wesentlich höher ist, als bei der Optimierung im Simulator, konnte hier in der zur Verfügung stehenden Zeit nicht so viele Generationen lang optimiert werden.

Im Durchschnitt dauerte die Messung eines Parametersatzes etwa eine Minute. Während der Optimierung kam es zu vielen Parametersätzen, die den Roboter umfallen ließen, worauf dieser erst wieder umständlich aufstehen und mit dem Standard-Parametersatz weiterlaufen musste. Dies wirkte sich stark auf die durchschnittliche Messzeit aus. Ferner waren jeweils nach wenigen Generationen der Optimierung die Schrauben an den Gelenken des Roboters nachzuziehen, da sich diese nach längerem Laufen lockern. Zudem mussten gelegentlich Servos aufgrund von Verschleiß gewechselt, sowie häufig eine Rekalibrierung der Servo-Motoren durchgeführt werden. So konnten in zwei Wochen der Optimierung mit dem Laufgestell insgesamt pro Laufrichtung 45 Generationen mit der (5+30)-Evolutionsstrategie evaluiert werden, was 1350 Funktionsauswertungen entspricht. Für beide Laufrichtungen zusammen handelt es sich also um die Summe von 2700 Funktionsauswertungen mit dem Roboter.

Bereits in den ersten Läufen wurde deutlich, dass die Variation der Parameter *FootTiltX* und *FootTiltY* nicht sinnvoll ist, da diese beiden Parameter unabhängig von den anderen

sind und daher konstant gesetzt werden konnten. Es wurde verzichtet auf eine automatische Einstellung durch das Optimierverfahren auf gute Werte für diese Parameter, da dies zu einer wesentlich längeren Optimierzeit geführt hätte. So wurden nach Experimenten für diese Parameter die Werte festgelegt als $FootTiltX = -0,0687$ und $FootTiltY = 0.019$. Dadurch wurde die Dimension des Optimierproblems von elf auf neun verringert, was zu einer schnelleren Konvergenz verhalf.

Während der Optimierung stellte sich heraus, dass sowohl der Drahtbügel am Kopf des Roboters als auch das Kabel für die Stromversorgung und die serielle Ansteuerung den Roboter beim Laufen zum Teil deutlich beeinflussen. Der Drahtbügel musste stets genau senkrecht nach oben ausgerichtet sein, damit er den Roboter nicht zu einer Seite herüberdrückte und so zu einem häufigeren Stürzen führte. Das Kabel zog den Roboter aufgrund seines Gewichtes vor allem dann zur Seite, wenn der Roboter weiter von dem Punkt entfernt war, an dem das Kabel senkrecht nach unten hing. Um dieses Problem zu umgehen, wurde das Kabel ständig von Hand so geführt, dass es sich stets senkrecht über dem Roboter befand. Ferner wurde der Roboter erneut hinter die Start-Lichtschranke gesetzt, wenn er durch eine nicht genaue Ausrichtung im Start schräg durch das Laufgestell lief und so die Holz-Bande des Laufgestells berührte.

Auffällig bei der Optimierung war, dass sehr viele Parametersätze den Roboter zum Umfallen brachten. Besonders traf dies für das Rückwärtslaufen zu, dort führten mehr als die Hälfte der Nachkommen einer Generation zu Stürzen. Warum gerade das Rückwärtslaufen dermaßen empfindlich ist, kann hier nicht geklärt werden. Zudem lag die maximale Geschwindigkeit für das Rückwärtslaufen nur etwas über der halben Geschwindigkeit für das Vorwärtslaufen. Es kann nur vermutet werden, dass dafür die unterschiedliche Gewichtsverteilung im Roboter durch das Einknicken der Knie verantwortlich ist. Ausserdem könnte das Gewicht der Platine auf dem Rücken des Roboters eine Rolle spielen.

Der beste gefundene Parametersatz nach den 45 Generationen der Optimierung führte für das Geradeauslaufen zu einer Laufgeschwindigkeit von reproduzierbar 220 mm/s, dieser Parametersatz wurde bereits nach 40 Generationen ermittelt. Eine Übersicht über den Verlauf der

Parametername	**Wert**	**Mutationsschrittweite σ_i**
StepDuration	726,255	329,772
StepHeight	22,477	2,871
StepLength	57,302	0,753
armAmplitudeX	0,682	0,023
armAmplitudeY	0,391	0,149
BodyAmplitudeX	0,074	0,026
BodyAmplitudeY	0,107	0,022
YOffset	14,959	2,247
ZOffset	18,536	2,434

Tabelle 7.3: Die Parameterwerte für den realen Roboter im Laufgestell nach der Optimierung über 45 Generationen für ein Vorwärtslaufen. Diese Parameter resultierten in einer Laufgeschwindigkeit von 220 mm/s.

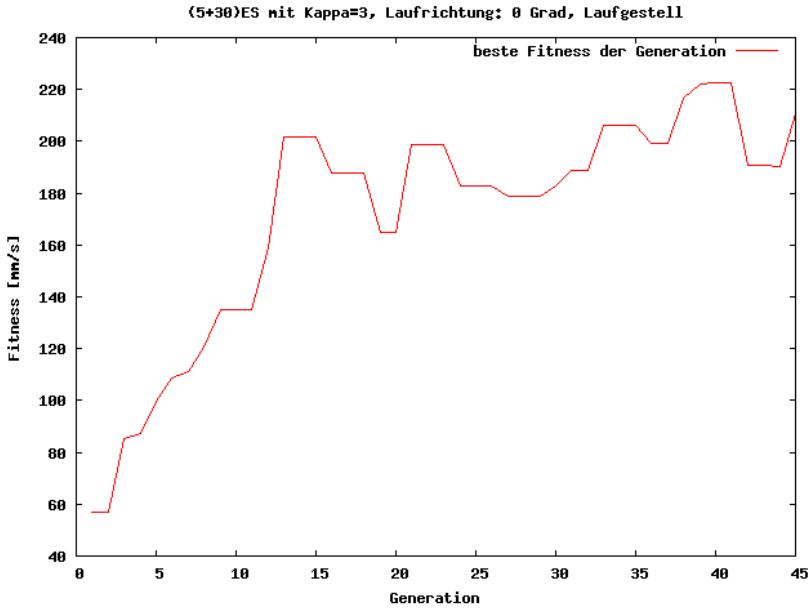


Abbildung 7.13: Verlauf der Fitnesswerte für die Optimierung im Laufgestell mit dem realen Roboter und der (5+30)-ES mit $\kappa = 3$. Es wurde mit 0° gerade nach vorne gelaufen, die schnellste ermittelte Geschwindigkeit betrug 220 mm/s.

Fitnesswerte, jeweils bezogen auf das fitteste Individuum der Generation, wird in Abbildung 7.13 gegeben. In Tabelle 7.3 sind die Werte der einzelnen, der Optimierung unterzogenen Parameter aufgelistet. Zudem werden auch die aktuellen Mutationsschrittweiten angegeben.

An den Werten der Mutationsschrittweiten kann auch schon erahnt werden, dass die Evolutionsstrategie noch längst nicht ein (zumindest lokales) Optimum erreicht hat, da dort die Mutationsschrittweiten wesentlich niedriger ausfallen müssten. Bei den Tests mit dem Simulator in Abschnitt 7.2.1 wurde ja schon deutlich, dass die meisten Mutationsschrittweiten nach etwa 100-200 Generationen nahe null waren. So kann hier davon ausgegangen werden, dass die Optimierung im Laufgestell noch weiter fortgeführt werden müsste, um die Mutationsschrittweiten absinken zu lassen. Es ist auch wahrscheinlich, dass die Fitnesswerte zumindest noch leicht ansteigen. Der Aufwand von 2-4 Wochen weiterer Optimierung wäre also gerechtfertigt. Trotzdem handelt es sich hier bereits um eine Laufgeschwindigkeit, welche für ein Laufen auf dem weichen Teppichboden über den Erwartungen lag.

Es handelt sich bei dieser Laufbewegung um ein sehr ruhiges und gleichmäßiges Laufen, vor allem zeichnet sich diese Parameterwahl neben der großen Geschwindigkeit dadurch aus, dass der Roboter sehr geradlinig läuft. Bei anderen Laufparametern, die während der Optimierung ausprobiert wurden, drehte sich der Roboter leicht bei einigen Schritten, so dass er oftmals in Kontakt mit der Holz-Bande des Laufgestells kam. Hier zeigt sich also, dass die Messung der Laufgeschwindigkeit tatsächlich als Fitnessfunktion ausreicht, da Parametereinstellungen, welche zu einem nicht geradlinigen Laufen führen, auch eine niedrigere Geschwindigkeit erzeugen.

Verblüffend ist bei diesem Parametersatz aber vor allem, dass aufgrund der Schrittdauer von

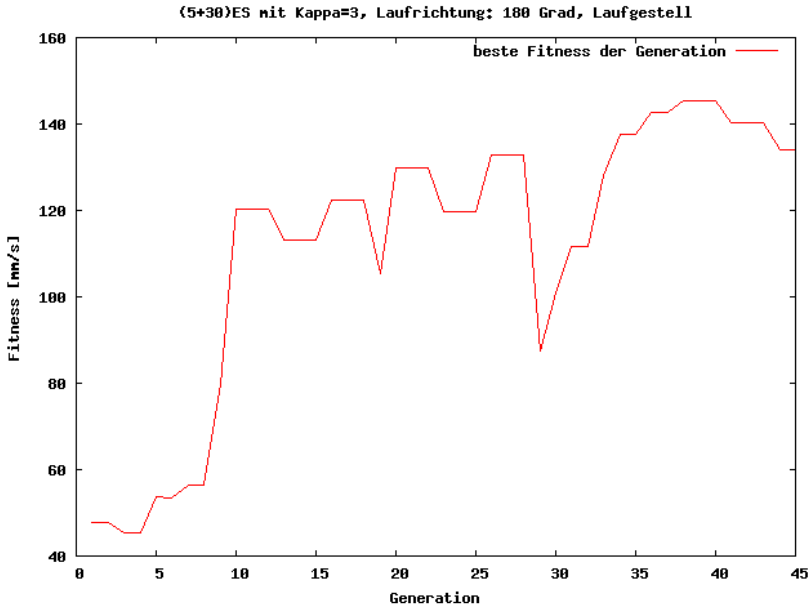


Abbildung 7.14: Verlauf der Fitnesswerte für die Optimierung im Laufgestell mit dem realen Roboter und der (5+30)-ES mit $\kappa = 3$. Dabei handelt es sich hier um die Laufrichtung 180°, also ein Rückwärtslaufen. Die daraus resultierende Laufgeschwindigkeit beträgt 145 mm/s.

726 ms und der Schrittlänge von 57 mm der Roboter theoretisch nur maximal 157 mm/s schnell laufen dürfte. Trotzdem kann die Geschwindigkeit von 220 mm/s wiederholt erreicht werden. Erklärbar ist diese höhere Geschwindigkeit nur dadurch, dass der Roboter seine Beine aufgrund der schwachen Servos und der hohen Kräfte durch sein Eigengewicht zum Teil weiter auseinanderauseinanderspreizt, als beabsichtigt. Dadurch wird die Schrittlänge also vergrößert. Zudem ist anzunehmen, dass die Füße nicht unerheblich über den Boden rutschen und sich so auch während der Bodenberührung fortbewegen.

Der Verlauf der Fitness für das Rückwärtslaufen, welches aufgrund der Art des Laufgestells ja parallel mit dem Vorwärtslaufen optimiert wird, ist in Abbildung 7.14 dargestellt. Hier ist sofort ersichtlich, dass die Geschwindigkeiten wesentlich niedriger sind als beim Vorwärtslaufen. Die bisher höchste erreichte Geschwindigkeit liegt bei 145 mm/s. Die zugehörigen Parameter sind in Tabelle 7.4 aufgelistet. Auch hier sind die Mutationsschrittweiten noch nicht so weit abgesenkt, dass keine weitere Verbesserung erwartet werden könnte. So ist auch hier zu erwarten, dass eine weitere Optimierung bessere Resultate bringt.

Jedoch sollte vor allem beim Rückwärtslaufen eine andere Fitnessfunktion benutzt werden. So würde es sich gerade hier anbieten, eine Beurteilung der Laufbewegung mit zu benutzen, so dass die Form der Bewegung ruhiger wird. Die entwickelten Parametereinstellungen für das Rückwärtslaufen führten nämlich allesamt zu einem sehr unruhigen und unstabilen Laufverhalten, ganz im Gegensatz zu den Einstellungen für das Vorwärtslaufen. Auch wäre es sogar denkbar, für das Rückwärtslaufen nicht eine maximale Geschwindigkeit zu fordern, sondern eher eine mittlere Geschwindigkeit mit allerdings maximaler Stabilität als Beurteilungskriteri-

um zu benutzen. Das Rückwärtslaufen ist beim Roboterfußball, dem Hintergrund für die hier herrschenden Bestrebungen nach maximaler Laufgeschwindigkeit, längst nicht so wichtig wie das Vorwärtslaufen. Von daher kann dort die Stabilität einen höheren Stellenwert einnehmen.

Parametername	**Wert**	**Mutationsschrittweite σ_i**
StepDuration	1188,773	40,473
StepHeight	13,801	0,369
StepLength	93,740	4,088
armAmplitudeX	0,234	0,030
armAmplitudeY	0,065	0,216
BodyAmplitudeX	0,005	0,004
BodyAmplitudeY	0,307	0,044
YOffset	16,201	4,082
ZOffset	28,074	7,500

Tabelle 7.4: Die Parameterwerte für den realen Roboter im Laufgestell nach der Optimierung über 45 Generationen für ein Rückwärtslaufen.

Es ist generell bei beiden Laufrichtungen zu beobachten, dass bei langsamen Bewegungen vor allem die Armbewegung in der Y-Z-Ebene einen großen Anteil an der Stabilität des Roboters hat, wogegen bei höheren Geschwindigkeit die Bewegung in der X-Z-Ebene wichtiger wird. Dies lässt sich damit erklären, dass bei langsameren Geschwindigkeiten der Roboter-Körper viel eher die Bestrebung hat, seitlich wegzukippen. Von daher ist es notwendig, durch Bewegungen der Arme zur Seite diesem Verhalten entgegen zu wirken. Wird mit höheren Geschwindigkeiten gelaufen, so wird auch die Trajektorie, welche des Roboter-Oberkörper seitlich hin- und herbewegt, schneller abgefahren. Dies führt dazu, dass der Roboter aufgrund der Massenträgheit nicht mehr so weit zur Seite ausgelenkt wird, also die seitliche Armbewegung an Wichtigkeit verliert. Die Bewegung der Arme nach vorne und hinten scheint nun den seitlichen Drehungen um die Z-Achse des Roboters während des Laufens entgegen zu wirken. So erklärt sich auch das sehr geradelinige Laufen mit höheren Geschwindigkeiten.

Es hat sich somit gezeigt, dass das Laufgestell als Experimental-Anordnung zur Optimierung der Laufparameter geeignet und empfehlenswert ist. Die Laufgeschwindigkeit des Roboters konnte stark gesteigert werden, bei weiterer Fortführung der Optimierung ist zu erwarten, dass sich die mögliche Geschwindigkeit noch etwas erhöht. Allerdings sei hier auch betont, dass die Anordnung des Laufgestells mit der Kabelführung und dem Drahtbügel den Roboter beeinflusst. So verhindert der Drahtbügel, dass der Roboter bei einer zu starken Bewegung seitwärts (hervorgerufen vor allem durch die Oberkörper-Trajektorie entlang der Y-Achse) umkippt. So verfälscht der Drahtbügel die Messung. Allerdings führt das Abbremsen des Bügels dazu, dass der Roboter nicht mehr synchron zu seiner Pendel-Bewegung die Füße bewegt, was in sehr langsamen Laufgeschwindigkeiten resultiert. Somit führen Parametereinstellungen, welche den Roboter zu stark aufschaukeln lassen, zwar nicht zu einem Umfallen, was sie mit Fitness null bestrafen würde, jedoch wird der Roboter trotzdem so langsam, dass diese Parametereinstellungen keine große Chance haben, lange in der aktuellen Population zu bleiben.

Auf jeden Fall sollten die durch die Optimierung im Laufgestell gewonnenen Parameter für

eine weitere Verwendung außerhalb des Laufgestells noch einigen Generationen der Optimierung mit dem völlig autonomen Roboter durchlaufen, um sicherzustellen, dass die Einflüsse der Kabelführung und des Drahtbügels nicht mehr die Laufparameter beeinflussen. Dafür steht ja durch Wireless LAN die Verwendung des Pocket PC offen, auf den sich die Parameter sowieso noch einstellen müssen.

8 Zusammenfassung

In dieser Arbeit wurde ein Laufmuster für den zweibeinigen Roboter Kondo KHR-1 entwickelt. Zunächst wurde dafür eine parametrisierbare Laufbewegung erstellt, basierend auf Trajektorien, welche die Bewegungen einzelner Roboter-Bestandteile vorgeben. Diese Parameter wurden mittels Evolutionsstrategien hinsichtlich maximaler Laufgeschwindigkeit des Roboters optimiert. Dazu wurden zunächst anhand eines physikalischen Simulators Tests durchgeführt, um verschiedene Typen von Evolutionsstrategien für das vorliegende Optimier-Problem zu bewerten. Nach Auswahl der (5 + 30)-Evolutionsstrategie mit $\kappa = 3$ wurde die Optimierung der Laufparameter schlussendlich in einem eigens dafür entwickelten Laufgestell durchgeführt.

Dabei stellte sich heraus, dass das entwickelte Laufmuster den Roboter befähigt, stabil aufrecht zu gehen, dies zudem in verschiedenen Geschwindigkeiten (in Abhängigkeit der gewählten Parameter) und in verschiedenen Richtungen. Die entwickelten Optimier-Behaviors ermöglichten automatisierte Tests im Simulator, das entwickelte Laufgestell ermöglichte eine kontrollierte Optimierung der Laufparameter mit dem realen Roboter. Es wurden Erkenntnisse über die Eignung verschiedener Evolutionsstrategien für diese Problemstellung gewonnen und für die endgültige Optimierung der Laufparameter eingesetzt, ferner wurden Untersuchungen über die Variation der Laufparameter für verschiedene Laufrichungen angestellt und die Varianz der Laufgeschwindigkeits-Messung ermittelt.

Schließlich wurden Laufparameter ermittelt, die es dem Kondo KHR-1 ermöglichen, auf Teppichboden mit einer Geschwindigkeit von 220 mm/s stabil geradeaus zu laufen.

8.1 Ausblick

Die bisherigen in dieser Arbeit erzielten Ergebnisse lassen nun einige Fragen offen. Ein besonderes Augenmerk sollte bei zukünftigen Entwicklungen auf der Lauf-Stabilität liegen. So ist zu überlegen, ob der hier gewählte Ansatz, eine feste Laufbewegung ungeachtet der tatsächlichen Bewegung des Roboters vorzugeben, für ein zuverlässiges Laufen ausreicht. So wäre es denkbar, die Trajektorien der Laufbewegung abhängig vom Gleichgewichtszustand des Roboters zu variieren. Damit könnte der Roboter ein Umfallen im Vorfeld verhindern, indem er sein Gewicht passend verlagert, oder z.B. die Arme zur Gegenregulation einsetzt.

Bleibt es bei dem hier beschriebenen Ansatz der festen Laufbewegung mit einer Optimierung deren Parameter, so wäre eine zusammengesetzte Fitnessfunktion zu empfehlen, welche nicht nur die Laufgeschwindigkeit als einziges Kriterium betrachtet. So wäre es für die Stabilität und Ruhe der Laufbewegung von Vorteil, wenn zusätzlich ruhigere Laufbewegungen bevorzugt würden, bei denen z.B. anhand der Werte des Beschleunigungs-Sensors gemessen wird, wie stark der Roboter erschüttert wird.

Da insbesondere beim Roboterfußball, dem Einsatzgebiete des hier behandelten Roboters,

nicht ständig die maximalen Geschwindigkeiten gefordert sind, sondern auch mittelschnelle oder langsame, könnte die hier vorgestellte Art der Optimierung auch für fest vorgegebene Geschwindigkeiten durchgeführt werden. So wäre eine Fitnessfunktion denkbar, welche die Abweichung von der vorgegebenen Geschwindigkeit bestraft. Zusammen mit einer Messung der Laufruhe könnte so für jede geforderte Geschwindigkeit eine Lauf-Parametrisierung gefunden werden, welche den Roboter bestmöglichst zum Laufen bringt. Für die in Abschnitt 4.6 beschriebene Ansteuerung der WalkingEngine würde dies bedeuten, dass für die geforderte Geschwindigkeit ein passender Parametersatz ausgewählt würde. Sinnvoll wäre es in diesem Zusammenhang, nur eine bestimmte Anzahl von Parametern für einige Geschwindigkeits-Stufen und Laufrichtungen zu optimieren und Zwischenwerte ggf. zu interpolieren.

Schließlich wäre es von Vorteil, wenn eine verbesserte Version des Laufgestells entwickelt würde. Es sollte vorrangig das Problem gelöst werden, dass die Kabel-Führung den Roboter beeinflusst. Durch die Benutzung eines PocketPCs ist dies ja bereits jetzt möglich, jedoch mit dem Nachteil, dass die Batteriekapazitäten nur für einige Minuten ausreichen. Ferner wäre eine Absicherung von Vorteil, die den Roboter nicht so stark beeinflusst, wie es momentan der Drahtbügel tut. Alternativ könnte ggf. der Roboter komplett ohne Laufgestell in einer freien Umgebung benutzt werden, wobei dann die Geschwindigkeitsmessung z.B. durch den Roboter selber über eine eingebaute Kamera oder über einen Distanzsensor erfolgen könnte. Jedoch auch hier müsste der Roboter per Batterie betrieben werden, ferner wären Vorkehrungen zu treffen (z.B. durch eine Polsterung), damit der Roboter bei einem Umfallen nicht beschädigt wird.

A Vergrößerte Abbildungen

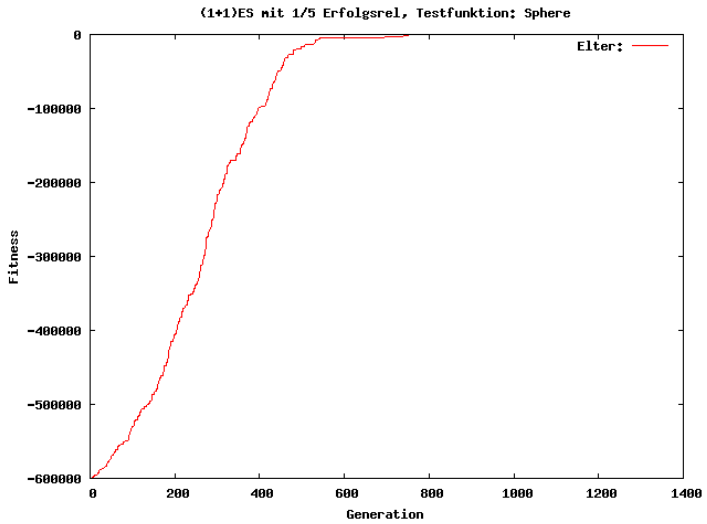


Abbildung A.1: Fitnessverlauf der $(1+1)$-ES auf SPHERE.

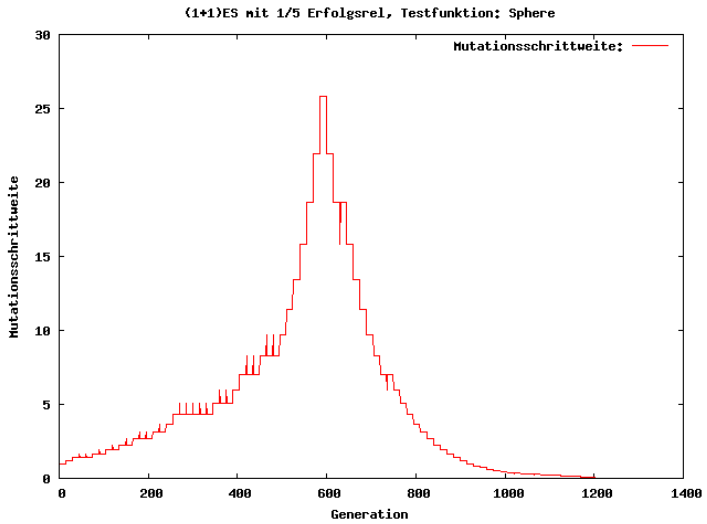


Abbildung A.2: Verlauf der Mutationsschrittweiten vom jeweiligen Elter der Population der $(1+1)$-ES auf SPHERE.

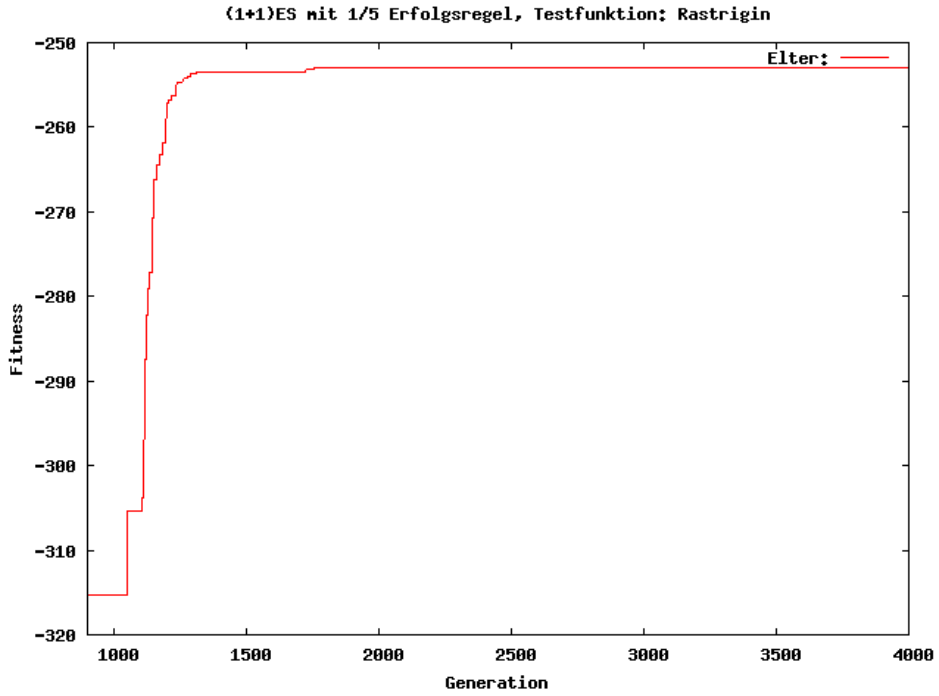


Abbildung A.3: Fitnessverlauf der $(1+1)$-ES auf RASTRIGIN, vergrößert ab Generation 800.

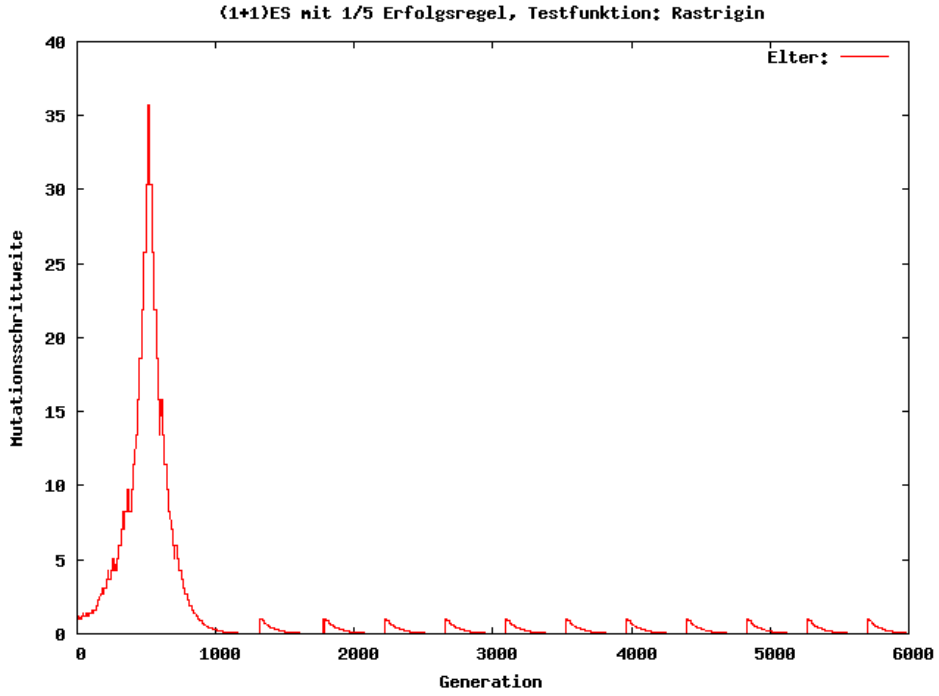


Abbildung A.4: Der Verlauf der Mutationsschrittweiten vom jeweiligen Elter der $(1+1)$-ES auf RASTRIGIN.

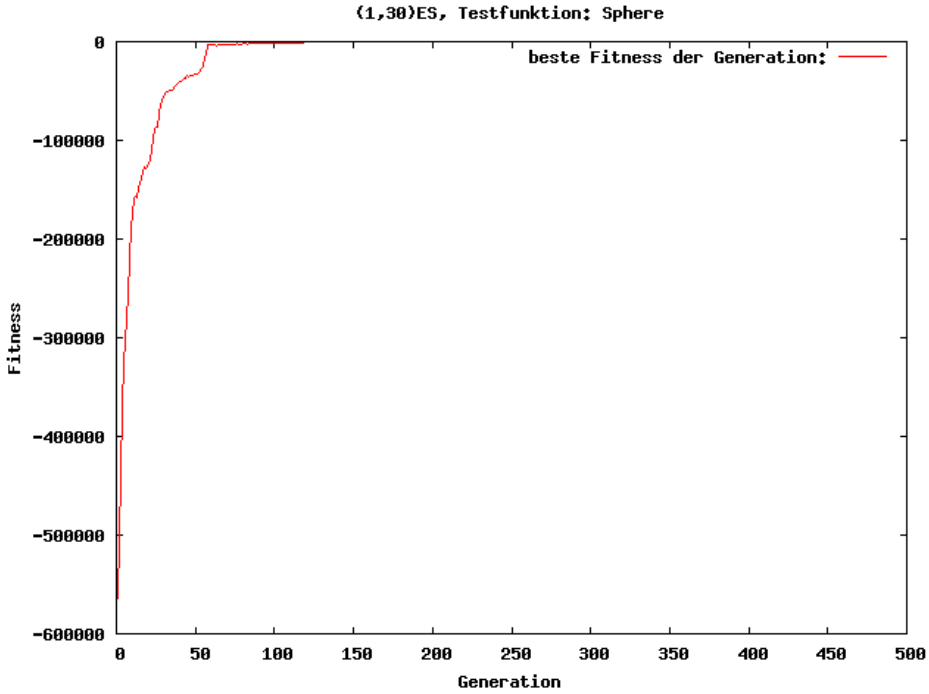


Abbildung A.5: Fitnessverlauf der $(1, \lambda)$-ES auf SPHERE.

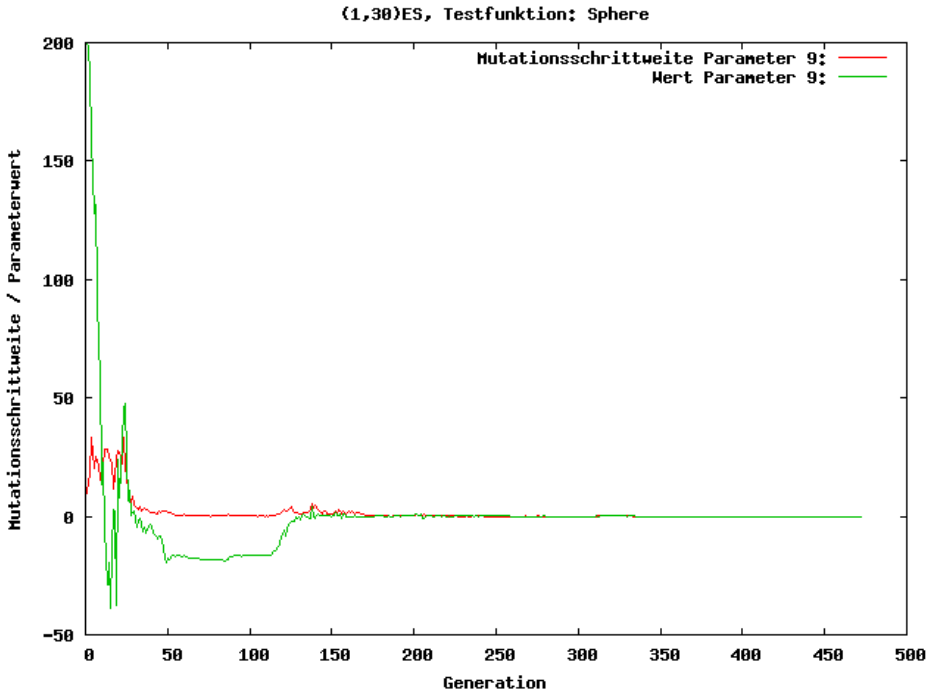


Abbildung A.6: Die Mutationsschrittweiten eines ausgewählten Parameters für das jeweils fitteste Individuum der Generation der $(1, \lambda)$-ES auf SPHERE.

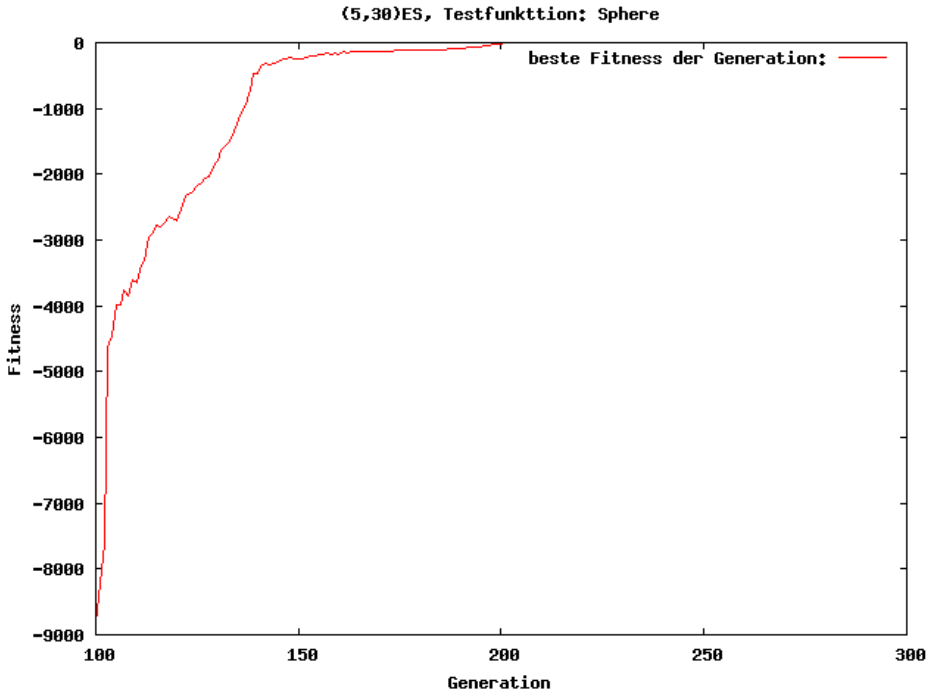


Abbildung A.7: Fitnessverlauf über die Generationen der (5,30)-ES auf SPHERE, vergrößert ab Generation 100.

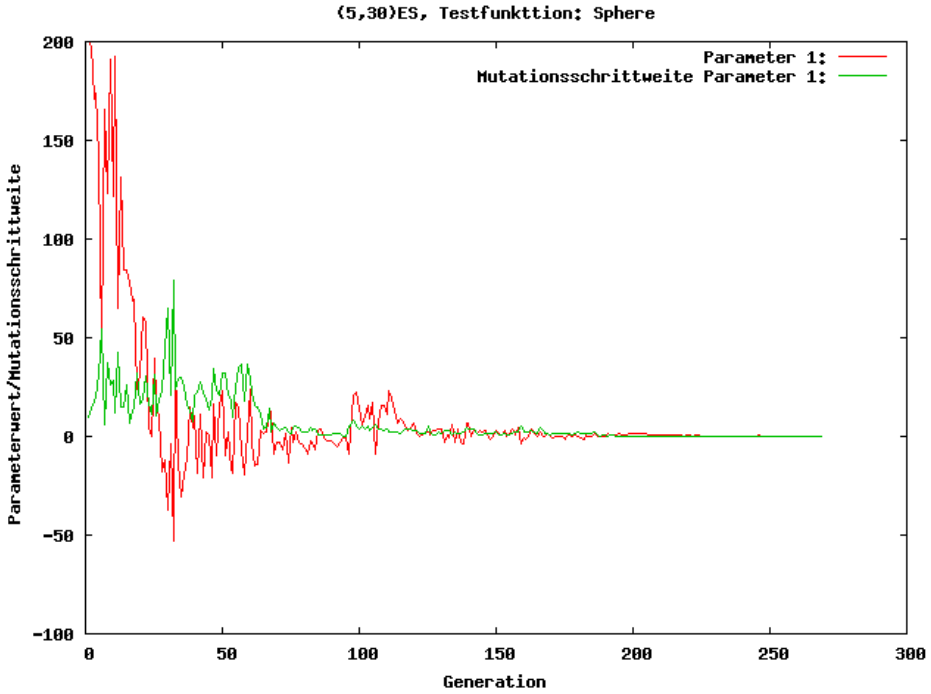


Abbildung A.8: Die Mutationsschrittweiten eines ausgewählten Parameters, sowie die Werte dieses Parameters für das jeweils fitteste Individuum der Generation der (5,30)-ES auf SPHERE.

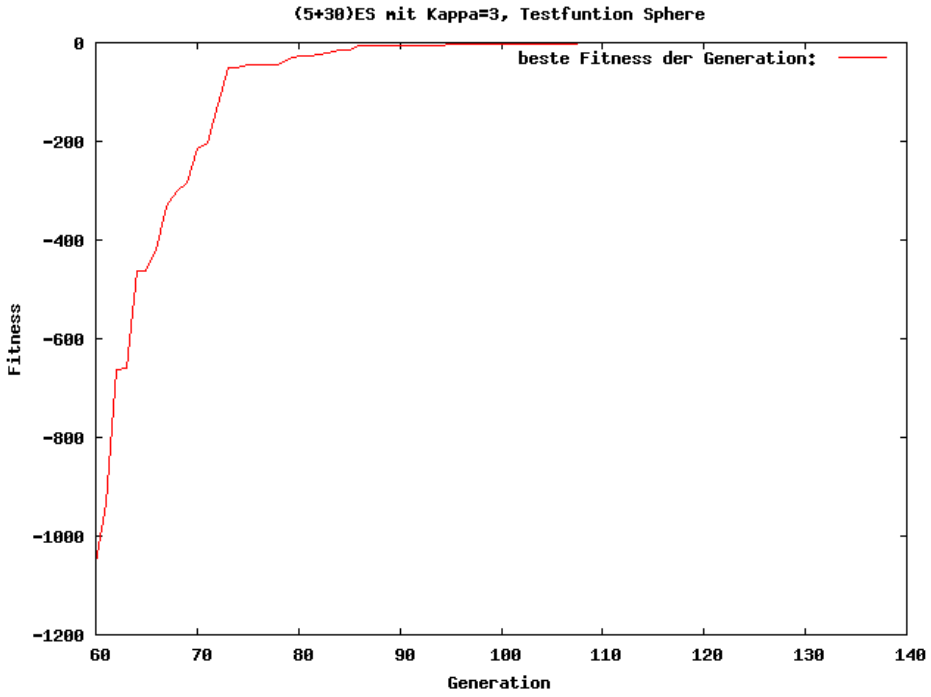


Abbildung A.9: Fitnessverlauf über die Generationen der (5+30)-ES auf SPHERE, vergrößert ab Generation 60.

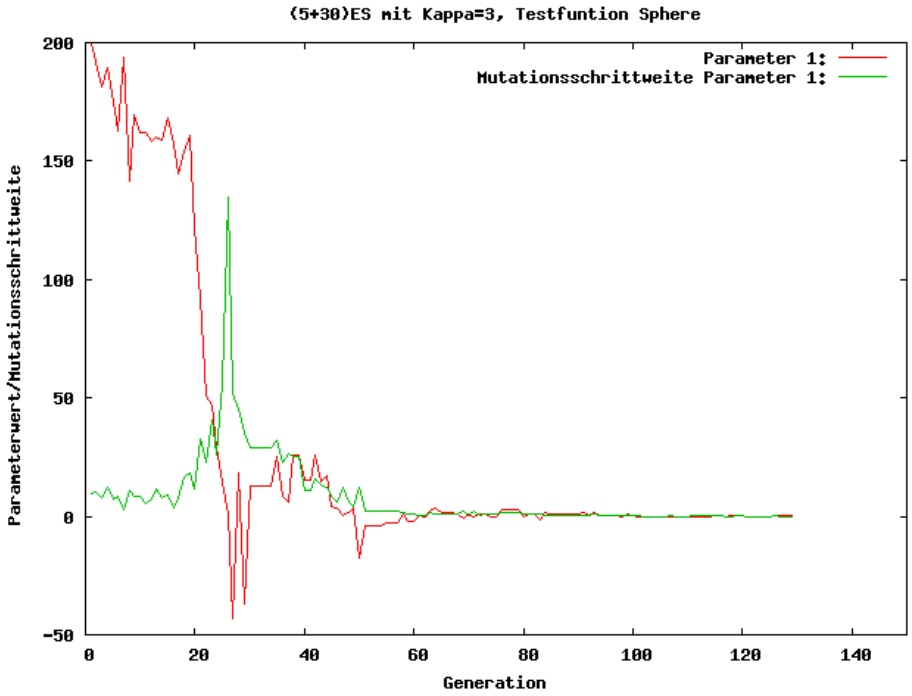


Abbildung A.10: Die Mutationsschrittweiten eines ausgewählten Parameters, sowie die Werte dieses Parameters der (5+30)-ES auf SPHERE.

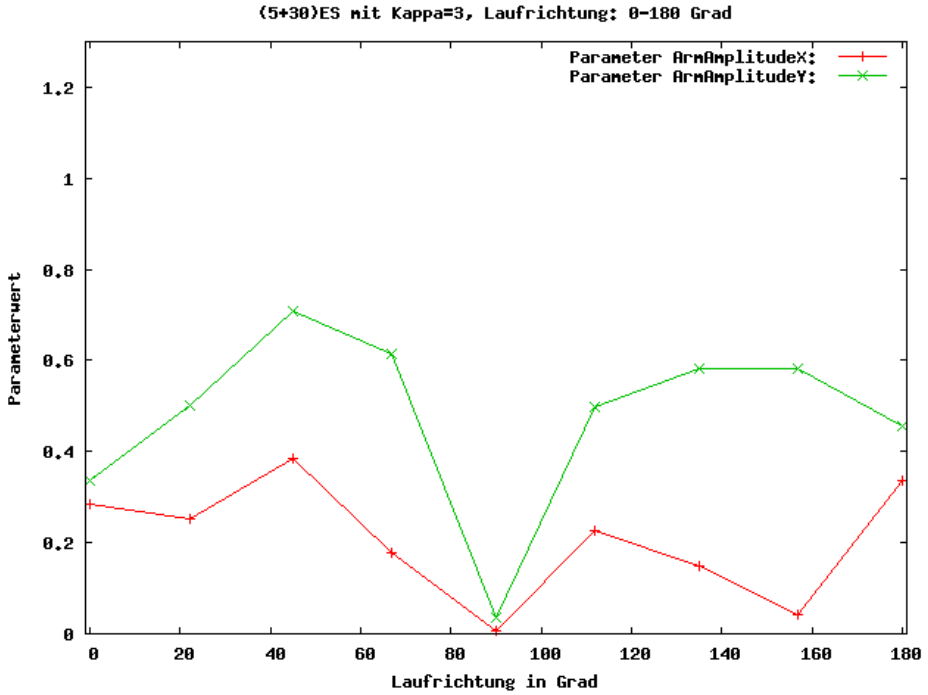


Abbildung A.11: Verlauf der Parameter $ArmAmplitudeX$ und $ArmAmplitudeY$ für verschiedene Laufrichtungen.

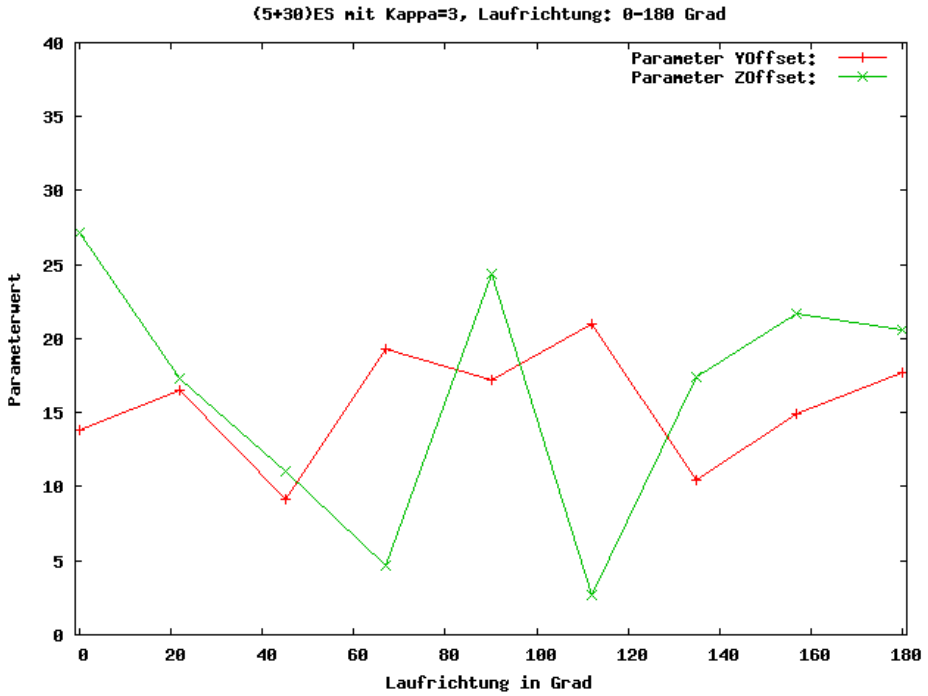


Abbildung A.12: Verlauf der Parameter $YOffset$ und $ZOffset$ für verschiedene Laufrichtungen.

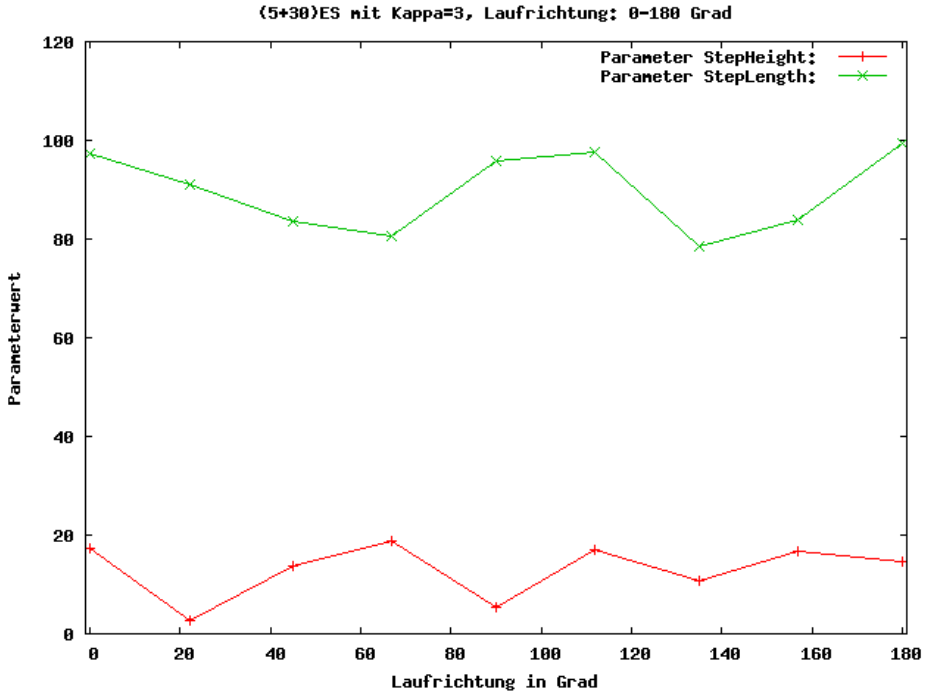


Abbildung A.13: Verlauf der Parameter *StepHeight* und *StepLength* für verschiedene Laufrichtungen.

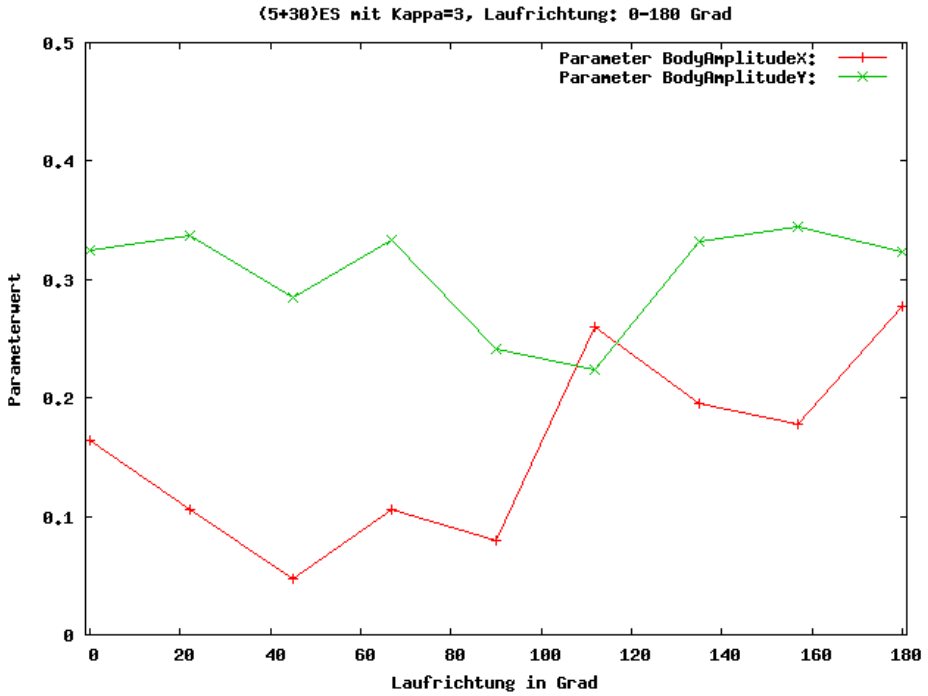


Abbildung A.14: Verlauf der Parameter *BodyAmplitudeX* und *BodyAmplitudeY* für verschiedene Laufrichtungen.

Literaturverzeichnis

[Alb01] Albert, A.: *Intelligente Bahnplanung und Regelung für einen autonomen, zweibeinigen Roboter*, Universität Hannover, Diss., 2001

[BMS06] Behnke, S. ; Müller, J. ; Schreiber, M.: Toni: A Soccer Playing Humanoid Robot. In: *RoboCup-2005: Robot Soccer World Cup IX* Bd. LNCS 4020, 2006 (Lecture Notes in Artificial Intelligence), S. 59–70

[BNKF02] Banzhaf, W. ; Nordin, P. ; Keller, R. E. ; Francone, F. D.: *Genetic Programming - An Introduction.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 2002

[Bre06] BreDoBrothers: *BreDoBrothers – Humanoid-League Team der Universitäten Dortmund und Bremen.* Stand: 15. September 2006. – `http://www.bredobrothers.de`

[Bri04] Brillowski, K.: *Einführung in die Robotik - Auslegung und Steuerung serieller Roboter -.* Shaker Verlag Aachen, 2004

[BS93] Bäck, T. ; Schwefel, H.-P.: An Overview of Evolutionary Algorithms for Parameter Optimization. In: *Evolutionary Computation* 1 (1993), Nr. 1, S. 1–23

[Düf04] Düffert, U.: *Vierbeiniges Laufen – Modellierung und Optimierung von Roboterbewegungen*, Humboldt-Universität Berlin, Diplomarbeit, 2004

[FGLK98] Frik, M. ; Guddat, M. ; Losch, D.C. ; Karatas, M.: Terrain Adaptive Control of the Walking Machine Tarry II. In: *Proc. European Mechanics Colloquium, Euromech 375 - Biology and Technology of Walking, Munich, 1998*, 1998, S. 108–115

[FKK+05] Friedmann, M. ; Kiener, J. ; Kratz, R. ; Ludwig, T. ; Petters, S. ; Stelzer, M. ; Stryk, O. von ; Thomas, D.: Darmstadt Dribblers 2005: Humanoid Robot. In: *RoboCup 2005: Robot Soccer World Cup IX*, 2005

[FOW66] Fogel, L. J. ; Owens, A. J. ; Walsh, M. J.: *Artificial Intelligence through Simulated Evolution.* New York, USA : John Wiley, 1966

[Gol89] Goldberg, D. E.: *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, 1989

[GTRLW+05] German Team: Röfer, T. ; Laue, T. ; Weber, M. ; Burkhard, H.-D. ; Jüngel, M. ; Göhring, D. ; Hoffmann, J. ; Altmeyer, B. ; Krause, T. ;

Spranger, M. ; Stryk, O. v. ; Brunn, R. ; Dassler, M. ; Kunz, M. ; Oberlies, T. ; Risler, M.: GermanTeam RoboCup 2005. 2005. – Forschungsbericht. – 247 Seiten, http://www.germanteam.org/GT2005.pdf

[HB92] Hoffmeister, F. ; Bäck, T.: Genetic Algorithms and Evolution Strategies: Similarities and Differences / Universität Dortmund, Lehrstuhl für Systemanalyse. 1992. – Forschungsbericht

[HNF07] Hebbel, M. ; Nistico, W. ; Fisseler, D.: Learning in a High Dimensional Space: Fast Omnidirectional Quadrupedal Locomotion. In: *RoboCup 2006: Robot Soccer World Cup X*, Springer, 2007 (Lecture Notes in Artificial Intelligence). – to appear

[Hol75] Holland, J.H.: *Adaption in Natural and Artificial Systems.* Ann Arbor, Michigan : University of Michigan Press, 1975

[Hum06] Humanoid: *Offizielle Webseite der RoboCup Humanoid League.* Stand: 15. September 2006. – http://www.humanoidsoccer.org/

[Kin06] Kindler, T.: *Entwicklung von Hard- und Software für einen zweibeinigen Roboter*, Universität Dortmund - Institut für Roboterforschung, Diplomarbeit, 2006

[Koz92] Koza, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge (Mass.) : The MIT Press, 1992

[Lar06] *Laufroboter Lara mit künstlichen Muskeln.* http://www.lara-robot.de/. Version: Stand: 15. September 2006

[LGC01] Luk, B. L. ; Galt, S. ; Chen, S.: Using genetic algorithms to establish efficient walking gaits for an eight-legged robot. In: *International Journal of Systems Science* 32(6) (2001), S. 703–713

[LM68] Liston, R.A. ; Mosher, R.S.: A versatile walking truck. In: *Proceedings of the Transportation Engineering Conference* Institution of Civil Engineers, London, 1968

[MM04] Meredith, M. ; Maddock, S.: Real-Time Inverse Kinematics: The Return of the Jacobian / Department of Computer Science, University of Sheffield. 2004. (Research Memorandum CS-04-06). – Forschungsbericht

[Nim06] NimbRo: *Team NimbRo, Humanoide Roboter an der Universität Freiburg.* Stand: 15. September 2006. – http://www.nimbro.net/

[ODE06] *ODE Open Dynamics Engine.* Stand: 15. September 2006. – http://www.ode.org/

[Rai86] Raibert, M. H.: *Legged robots that balance.* MIT Press, Cambridge, MA, 1986

[Rec73] Rechenberg, I.: *Evolutionsstrategie.* Friedrich Fromm Verlag, Stuttgardt, 1973

[Rie92] Riesler, H.: *Roboterkinematik – Grundlagen, Invertierung und symbolische Berechnung.* Vieweg Verlag Wiesbaden, 1992

[Rob06] Robocup: *RoboCup Official Site.* Stand: 15. September 2006. – `http://www.robocup.org`

[Sch75] Schwefel, H.-P.: *Evolutionsstrategie und numerische Optimierung*, Technische Universität Berlin, Fachbereich Verfahrenstechnik, Dr.-Ing. Dissertation, 1975

[Sch87] Schwefel, H.-P.: Collective phenomena in evolutionary systems. In: *Preprints of the 31st Annual Meeting of the International Society for General System Research, Budapest* Bd. 2, 1987, S. 1025–1033

[Sch95] Schwefel, H.-P.: *Evolution and Optimum Seeking.* Wiley Interscience, 1995

[WN02] Wolff, K. ; Nordin, P.: Evolution Of Efficient Gait With An Autonomous Biped Robot Using Visual Feedback. In: *Proceedings of the Mechatronics 2002 Conference, Drebbel Institute for Mechatronics*, 2002

[WN03] Wolff, K. ; Nordin, P.: Learning Biped Locomotion from First Principles on a Simulated Humanoid Robot using Linear Genetic Programming. In: *In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2003 (pp.). Chicago*, Morgan Kaufmann, Juli 2003

[WP04] Wischmann, S. ; Pasemann, F.: From passive to active dynamic 3D bipedal walking - An evolutionary approach -. In: Armada, M. (Hrsg.) ; Santos, P. Gonzalez d. (Hrsg.): *Proc. of the 7th Int. Conference on Climbing and Walking Robots (CLAWAR 2004)*, Springer Verlag, 2004, S. 737–744

[Zie03] Ziegler, J.: *Evolution von Laufrobotersteuerungen mit Genetischer Programmierung*, Fachbereich Informatik, Universität Dortmund, Diss., Februar 2003

Abbildungsverzeichnis