TUHH
Technische Universität Hamburg-Harburg

Institut
für
Zuverlässiges Rechnen

Implementierung und Analyse verschiedener Regelstrategien für dynamische Laufbewegungen humanoider Roboter auf Basis des NAO-Robotiksystems

Diplomarbeit

vorgelegt von
Stefan Kaufmann

Hamburg, den 31. Oktober 2011

Erstprüfer : Prof. Dr.-Ing. Sven-Ole Voigt
Zweitprüfer: Prof. Dr. Herbert Werner

Erklärung
Hiermit versichere ich, Stefan Kaufmann, diese Arbeit im Rahmen der an der Technischen Universität Hamburg-Harburg üblichen Betreuung selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel eingesetzt zu haben.

Stefan Kaufmann

Hamburg, den 31. Oktober 2011

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Der Traum, einen menschenähnlichen Roboter zu erschaffen, lässt sich in der Geschichte weit zurückverfolgen. So soll, der griechischen Mythologie nach, der Schmiedegott Hephaistos im Jahre 3500 v. Chr. einen bronzenen Sklaven erschaffen haben. Jüngere Entwürfe über humanoide Roboter sind unter anderem von Leonardo da Vinci aus dem 15. Jahrhundert bekannt. Zu der damaligen Zeit ließen sich die Entwürfe allerdings noch nicht realisieren. Der große Durchbruch in diesem Forschungsbereich gelang erst in der zweiten Hälfte des 20. Jahrhunderts. Eine immernoch aktuelle Herausforderung im Gebiet der humanoiden Robotik ist die Implementierung von menschenähnlichen Laufalgorithmen. In der heutigen Zeit findet auf diesem Gebiet viel Forschungsarbeit statt. Der erste Roboter, der nicht nur gehen, sondern auch rennen kann, wurde eigenen Angaben zufolge von der japanischen Firma *Sony* im Jahre 2003 entwickelt und trägt den Namen *QRIO*. Der bekannteste humanoide Roboter, der seit 2004 ebenfalls rennen kann, ist wohl der von der japanischen Firma *Honda* entwickelte *Asimo*.
Neben der Laufimplementierung stellt die Autonomie der Roboter einen bedeutenden Forschungsschwerpunkt dar. So können einige Roboter Gegenstände und Personen erkennen sowie auf bestimmte Situationen reagieren. Ein Feld, in dem die Autonomie stark zur Geltung kommt, sind die beim *RoboCup* ausgetragenen Fußballturniere, bei denen Roboterteams gegeneinander antreten. Die Roboter dürfen hierbei nicht ferngesteuert werden, sondern müssen alle Entscheidungen selbstständig treffen. Die Wettkämpfe werden in unterschiedlichen Ligen ausgetragen, wobei auch in einigen Ligen humanoide Roboter gegeneinander antreten. Die Vision der Initiatoren des *RoboCup* ist es, bis 2050 ein Team humanoider Roboter gegen den dann amtierenden Fußballweltmeister antreten zu lassen und zu gewinnen.

Das Institut für Zuverlässiges Rechnen der Technischen Universität Hamburg-Harburg hat Anfang 2011 fünf *Nao*-Robotiksysteme erworben. Diese Robotiksysteme wurden über Studiengebühren finanziert und sollen daher auch den Studierenden zur Nutzung bereitgestellt werden. Ziel ist es, mit diesen Robotiksystemen in naher Zukunft an internationalen Wettbewerben, wie z.B. dem *RoboCup* teilzunehmen.
Diese Arbeit widmet sich daher zu einem großen Teil der Einrichtung einer Simulations- und Testumgebung, um ein wissenschaftliches Arbeiten mit dem System zu ermöglichen. Darüber hinaus bietet sie einen Einstieg in das komplette System, worunter der Roboter selbst, aber auch diverse Softwareanwendungen und interne Prozesse zählen.
Des Weiteren wird ein Laufalgorithmus auf Basis der aktuellen Forschungsergebnisse in diesem Bereich ausgewählt, implementiert und analysiert. Hierbei wird detailliert auf die Kinematik des Systems eingegangen, die die Grundlage der Laufbewegungen

darstellt. Die abschließende Evaluierung zeigt, dass der zugrunde liegende Algorithmus die Fähigkeit besitzt, auf Störungen während des Laufens zu reagieren.

Im Folgenden wird die Struktur der Arbeit erläutert. Die Vorstellung des *Nao*-Robotiksystem findet in Kapitel 2 statt. Im einzelnen werden auf die, für diese Arbeit benötigten Hardwarekomponenten und die zur Verfügung gestellte Software eingegangen. Für die Software findet darüber hinaus eine Evaluierung statt, um zu testen, ob sie für diese Arbeit nutzbar ist. Im darauf folgenden Kapitel wird der Entwurf einer Software zur Messdatenaufzeichnung beschrieben und gezeigt, warum diese Entwicklung für ein wissenschaftliches Arbeiten mit dem Roboter unumgänglich gewesen ist. Das Kapitel 4 behandelt die Auswahl einer geeigneten Simulationsumgebung sowie die Einrichtung und Verknüpfung mit bereits bestehender Software. Der aktuelle Stand der Regelstrategien für zweibeinige Roboter wird in Kapitel 5 aufgezeigt. Im darauf folgenden Kapitel wird die Kinematik des Systems beschrieben, wobei Vorwärtskinematik und inverse Kinematik getrennt behandelt werden. Das Kapitel umfasst ebenfalls Lösungen des Problems der inversen Kinematik für Arme und Beine des Roboters. Kapitel 7 befasst sich mit der Implementierung einer Laufstrategie und den dazu notwendigen Softwaremodulen in den Gesamtprozess. Hierbei wird sowohl die Eingliederung in den Simulationprozess als auch in den Roboterprozess beschrieben. Im Kapitel 8 wird aus den vorgestellten Regelstrategien eine für den Roboter passende ausgewählt und anschließend die Implementierung beschrieben. Hierbei wird auf Besonderheiten eingegangen, die die Implementierung für das *Nao*-Robotiksystem betreffen, sowie eine Optimierung des Modelles vorgenommen. Anschließend wird das System durch Messungen evaluiert. Im letzten Abschnitt dieses Kapitels wird ein neu entwickelter Ansatz zur Modellaktualisierung vorgestellt. Das letzte Kapitel beinhaltet eine Zusammenfassung dieser Arbeit und gibt einen Ausblick für zukünftige Arbeiten.

2. Das Nao Robotiksystem

Die 2005 von Bruno Maisonnier gegründete Firma *Aldebaran Robotics* hat die Vision, in naher Zukunft menschliche Roboter für den allgemeinen Gebrauch herzustellen. Das erste Produkt der Firma ist der *Nao*-Roboter, der erstmals 2006 vorgestellt wurde. Mittlerweile sind verschiedene Versionen dieses Systems für Universitäten und Forschungseinrichtungen zu erhalten.
Der *Nao* wird allerdings nicht nur zu Forschungszwecken eingesetzt, sondern ist seit 2008 der Roboter, welcher in der *Standard Platform League* des *RoboCups* eingesetzt wird. Der *RoboCup* ist ein jährlich durchgeführter Wettbewerb, bei dem Roboterteams in verschiedenen Disziplinen und Ligen gegeneinander antreten. Das ursprüngliche Ziel der *RoboCup Federation* war es, bis zum Jahr 2050 ein Fußballspiel mit einem Team aus Robotern gegen den dann amtierenden Fußballweltmeister zu gewinnen.

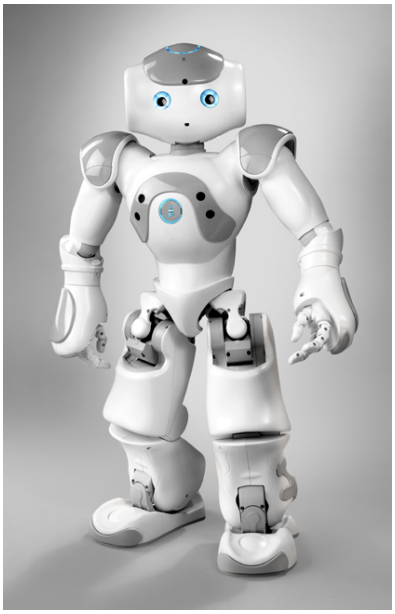


Abb. 2.1.: *Nao*-Roboter [Ald11]

In der *Standard Platform League* des *RoboCups* treten, anders als bei den anderen Ligen, keine eigenständig entwickelten Roboter, sondern stets baugleiche Roboter in einem Fußballspiel gegeneinander an. Vor 2008 wurde diese Liga noch als *Four Legged League* bezeichnet, da zu dieser Zeit der von der Firma *Sony* entwickelten *Aibo* eingesetzt wurde. Als Sony allerdings 2006 ankündigte, dass die Produktion des *Aibo* eingestellt wird, wurde nach einem passenden Nachfolger gesucht. Die Wahl fiel auf den *Nao* und die Liga wurde in *Standard Platform League* umbenannt.
Die Firma *Aldebaran Robotics* arbeitet derzeit an einer kommerziell erhältlichen Version des *Naos*, die 2012 erhältlich sein soll. Ein weiteres Projekt der Firma ist *Romeo*, ein 140 cm großer Roboter, welcher älteren oder pflegebedürftigen Menschen bei alltäglichen Aufgaben unterstützen soll.

In den nachfolgenden Abschnitten wird die Hard- und Software des *Nao*-Robotiksystems vorgestellt.

2.1. Hardware

Die Hardwarekomponenten des *Naos* unterscheiden sich hinsichtlich der Version des Roboters. In dieser Arbeit wird ausschließlich die Version Nao^{H25} 3.3 benutzt. Die

Zahl 25 gibt hierbei die Anzahl der Freiheitsgrade des Roboters an. Alle Angaben dieser Arbeit beziehen sich auf diese Version.
Im Anhang A sind alle übrigen Komponenten und Kenngrößen des Robotiksystems zu finden.
Im Folgenden werden die für diese Arbeit notwendigen Hardwarekomponenten kurz vorgestellt.

Größe
Der *Nao* hat stehend eine Größe von etwa 58 cm und kommt auf ein Gesamtgewicht von ca. 5 kg.

Netzwerk
Der *Nao* verfügt über eine Ethernetschnittstelle, die sich hinter einer Abdeckung auf der Rückseite seines Kopfes befindet. Des Weiteren ist er mit Wi-Fi (IEEE 802.11g) ausgestattet. Beim Start versucht der Roboter sich automatisch mit bekannten Netwerken zu verbinden.

Motherboard
Für die Rechenleistung des Roboters ist ein AMD Geode Mikroprozessor mit 500MHz im Kopf verbaut. Des Weiteren sind 256MB SDRAM sowie ein 2GB Flash verfügbar.

Inertiales Navigationssystem
Im Torso des Roboters befinden sich zwei einachsige Gyroskope und ein dreiachsiger Beschleunigungssensor. Die Gyroskope weisen eine relative Messgenauigkeit von 5% auf, der Beschleunigungssensor von 1%.

Kraftsensoren
In den Füßen des *Naos* befinden sich jeweils vier resistive Kraftsensoren mit einem Messbereich von 0-25 N und einer relativen Messgenauigkeit von 20%.

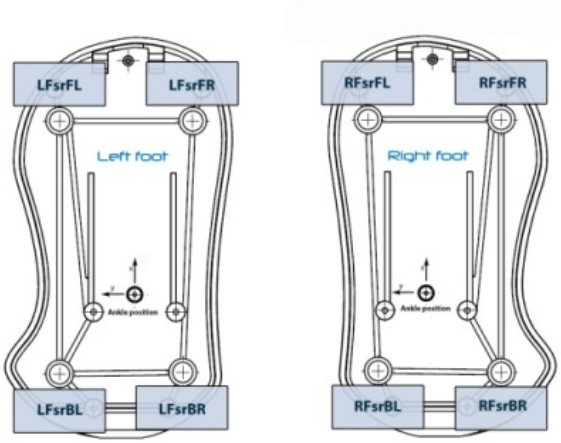


Abb. 2.2.: Resistive Kraftsensoren in den Füßen [Ald11]

Aktoren

Der *Nao* verfügt über insgesamt 25 Freiheitsgrade, die durch kernlose Gleichstrommotoren realisiert werden. Im Kopf des Roboters befinden sich zwei Freiheitsgrade. In den Armen sind jeweils vier Freiheitsgrade vorhanden und in den Beinen jeweils fünf. Die Hände des Roboters verfügen über jeweils zwei Freiheitsgrade und der verbleibende letzte Freiheitsgrad befindet sich in der Hüfte. Dieser Freiheitsgrad ist an beide Beine gekoppelt. Die Gelenke sind darüber hinaus mit Hallsensoren ausgestattet, die eine Positionsmessung ermöglichen. Insgesamt sind 36 dieser Sensoren verbaut, wovon je zwei Sensoren pro Freiheitsgrad in der unteren Körperhälfte (Hüfte und Beine) und je ein Sensor für die restlichen Freieheitsgrade verbaut sind. Die Sensoren sind mit 12 Bit aufgelöst, was einer Winkelauflösung von ca. 0,1° entspricht.

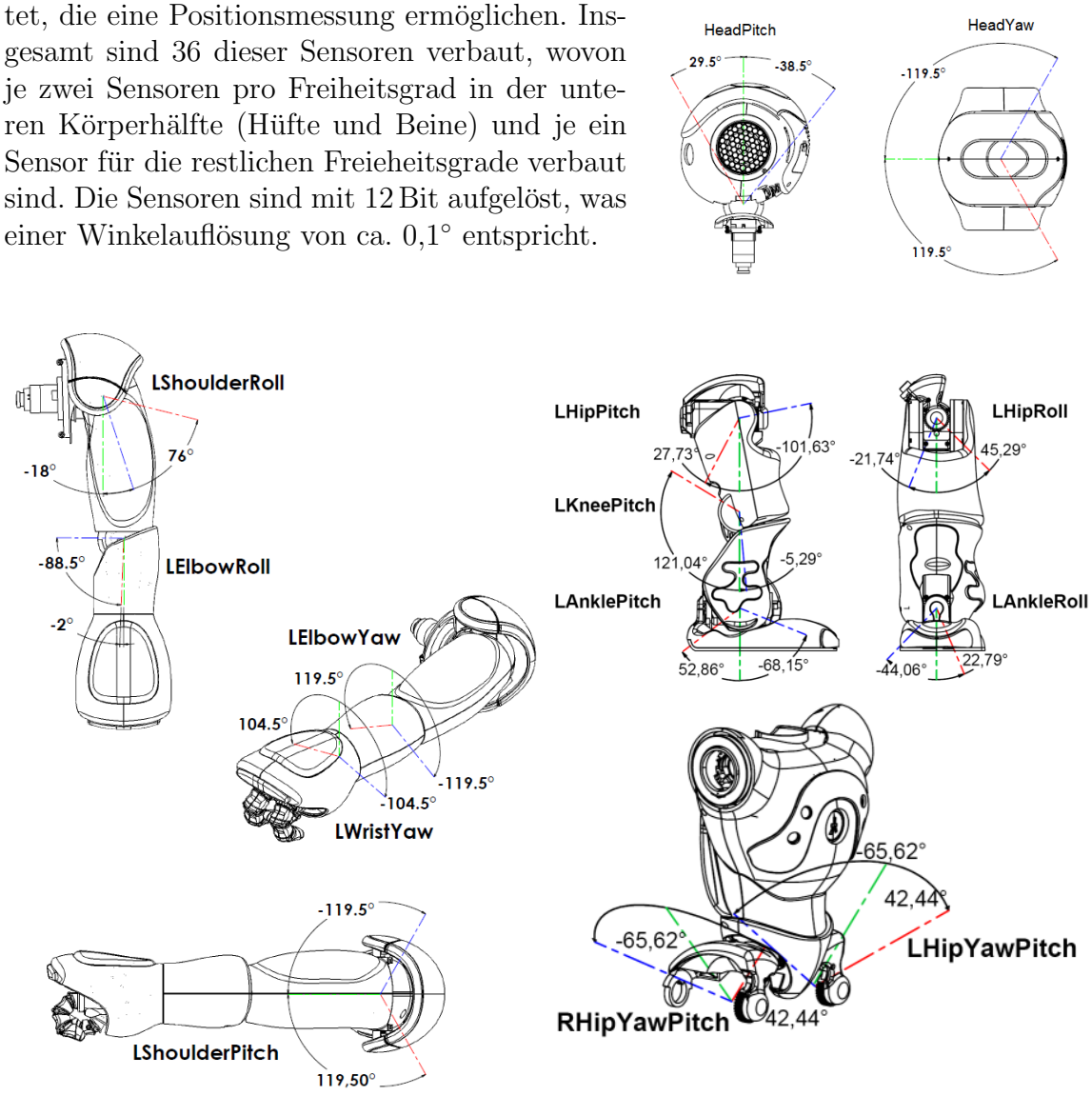


Abb. 2.3.: Übersicht über die Aktoren des Roboters [Ald11]

Anmerkung: In Abbildung 2.3 ist nur einer der beiden Handfreiheitsgrade, der *WristYaw*, eingezeichnet. Der zweite Handfreiheitsgrad dient zum Öffnen und Schließen der Hand. Hierbei werden die Finger über Seilzüge gesteuert. Des Weiteren sind die eingezeichneten Aktoren *LHipYawPitch* und *RHipYawPitch* gekoppelt. Sie werden gemeinsam durch einen Aktor gesteuert.

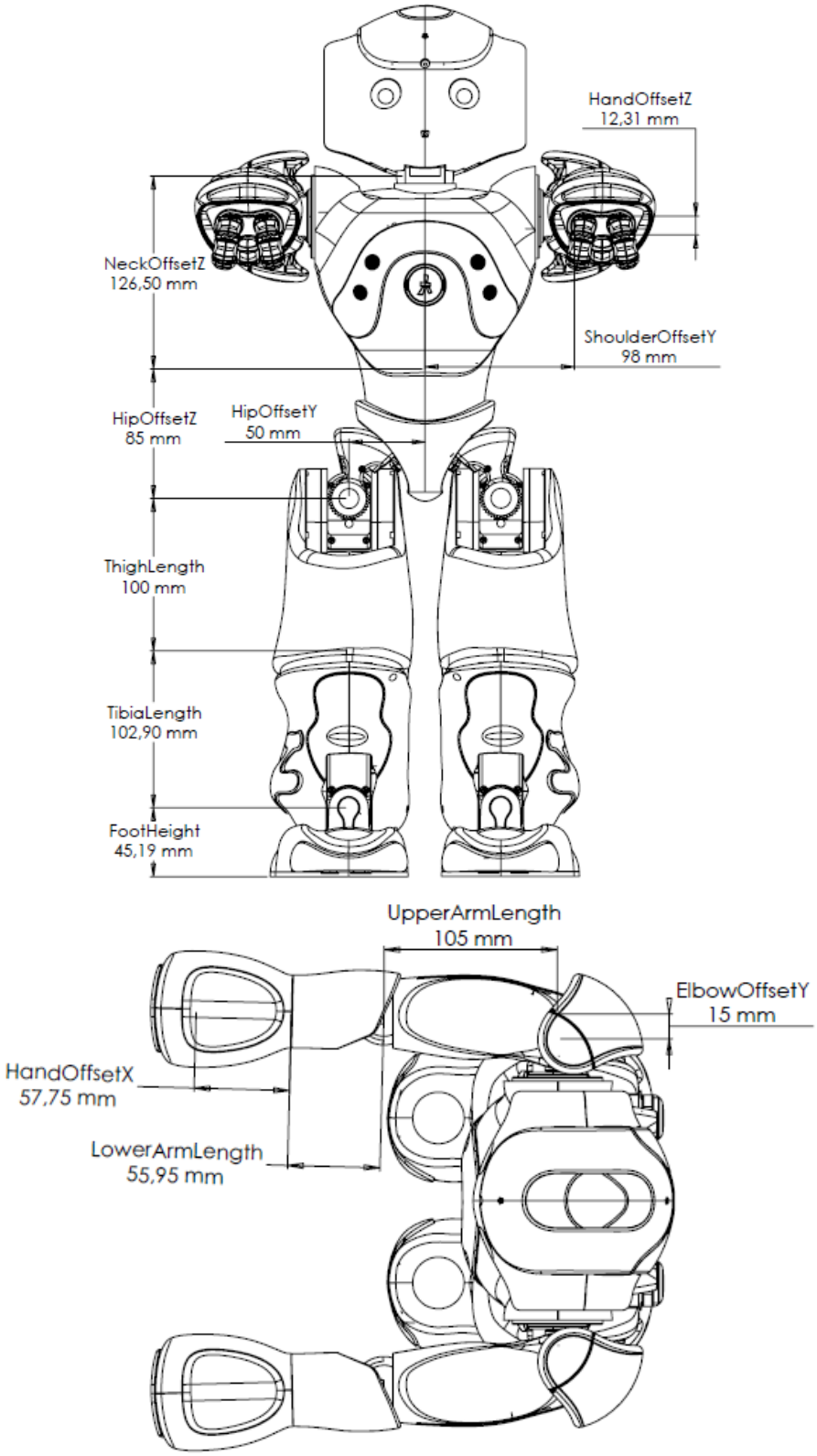


Abb. 2.4.: Körperabmessungen des *Naos* [Ald11]

2.2. Software

Das *Nao*-Robotiksystem kommt mit einer Vielzahl an Software, welche in diesem Abschnitt genauer vorgestellt werden soll. Auf dem Roboter selbst läuft als Betriebssystem ein 32-Bit Embedded Linux.
Im Folgenden werden die vom Hersteller mitgelieferten Anwendungen *Choregraphe*, *Telepathe* und *NaoSim* vorgestellt. Darüber hinaus wird ein SDK (Software Development Kit) zur Entwicklung eigener Module in einer Hochsprache bereitgestellt. Eigene Module kommunizieren mit dem Middlelayer *NaoQi*, welches ebenfalls erläutert wird.

2.2.1. Choregraphe

Die Software *Choregraphe* dient zur grafischen Programmierung des Roboters. Sie ist besonders zum schnellen Einstieg in das System sowie für Benutzer mit eingeschränkten Programmierkenntnissen geeignet. In Abbildung 2.5 ist die grafische Oberfläche des Programmes illustriert. Im linken Teil der Abbildung (Bereich 1) ist

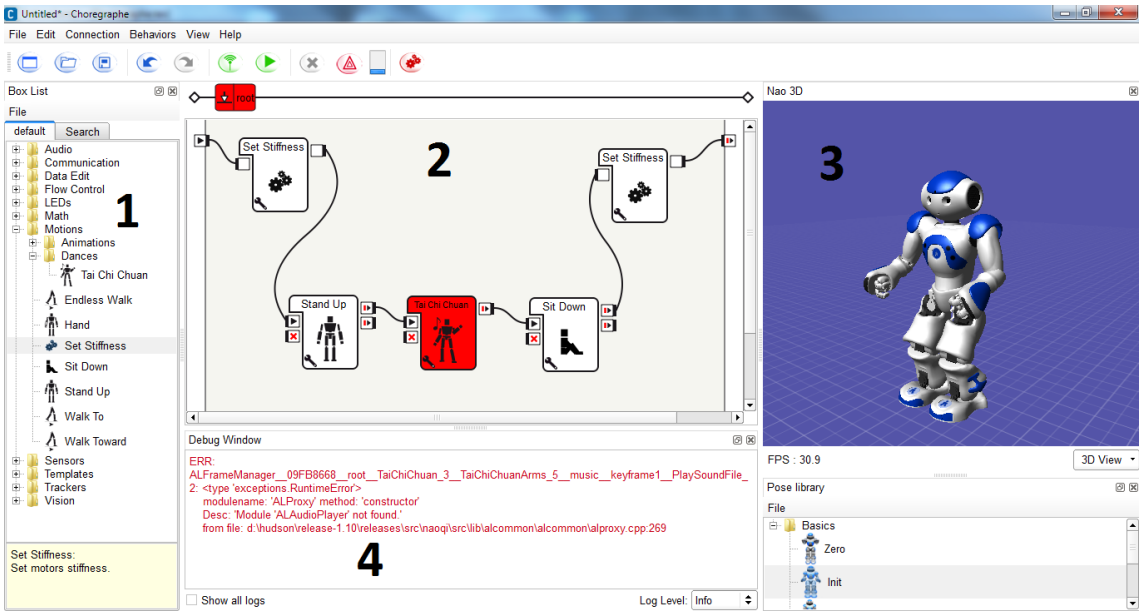


Abb. 2.5.: Choregraphe Benutzeroberfläche

eine BoxList dargestellt. In dieser Liste sind vordefinierte Blöcke in verschiedene Kategorien unterteilt. Es gibt Blöcke um Sensordaten abzufragen, Bewegungen auszuführen, LEDs zu erleuchten und vieles mehr. Der Bereich 2 ist das Flussdiagramm und dient zur Programmierung. In diesen Bereich können per drag&drop Blöcke aus der Boxlist gezogen und dann verbunden werden. Die Reihenfolge der Verbindung entspricht dann der Reihenfolge, in der die Blöcke ausgeführt werden. In der Abbildung ist ein Programm zu sehen, welches den Roboter aufstehen, tanzen und sich wieder hinsetzen lässt. Die Blöcke *setStiffness* dienen dazu die Steifigkeit der Motoren zu setzen. Hierauf wird später noch genauer eingegangen.
Im Bereich 3 ist ein 3D-Modell des Roboters zu sehen. Dieses Modell bewegt sich bei

Ausführung eines Programmes entsprechend. Allerdings bleibt der Torso des Roboters immer aufrecht und an der gleichen Position, sodass durch dieses Modell nicht festgestellt werden kann, ob der Roboter umfallen würde oder nicht.
Der Bereich 4 stellt eine einfache Ausgabekonsole dar, welche zur Ausgabe von Fehlern und Informationen dient.

Choregraphe bietet die Möglichkeit, einen *Nao*-Roboter mit dem Programm zu verknüpfen. Ist ein Roboter verknüpft, so wird beim Ausführen eines Programmes, dieses zunächst auf den Roboter übertragen und anschließend direkt ausgeführt. Darüber hinaus wird während der Ausführung das 3D-Modell ständig aktualisiert.
Es ist ebenfalls möglich einzelne Gelenke des Roboters direkt zu steuern, ohne dafür ein Programm schreiben zu müssen. Hierzu genügt ein Klick auf einen Körperbereich des Roboters im 3D-Modell. Hierdurch wird ein zugehöriges Dialogfenster geöffnet. Abbildung 2.6 zeigt das Dialogfenster. Durch Veränderung der Schieberegler oder

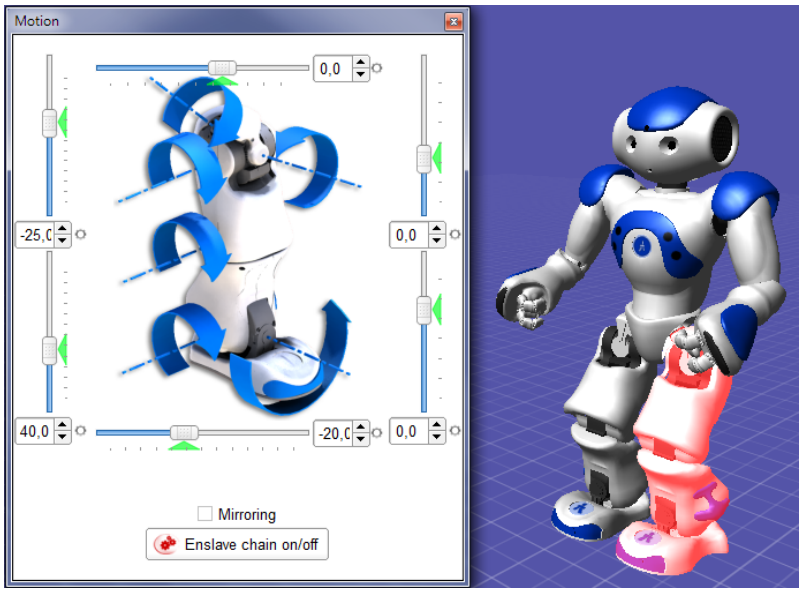


Abb. 2.6.: Dialogfenster zur Änderung der Gelenkstellung

numerische Eingabe in die entsprechenden Felder, können die Stellungen der Gelenke verändert werden. Die Zahlen in den Feldern entsprechenden den Gelenkstellungen in Grad. Unten im Dialogfeld ist ein Optionsfeld *Mirroring* zu sehen. Ist diese aktiviert, so wird die Bewegung gespiegelt. Wird z.B. das linke Kniegelenk verändert, so wird das recht Kniegelenk ebenso verändert, dass eine spiegelsymmetrische Bewegung entsteht. Ist die Option *Enslave chain on/off* aktiviert, so wirken sich Veränderungen der Gelenkstellungen nicht nur auf das 3D-Modell, sondern ebenfalls auf den verknüpften Roboter aus.

Die Programmierung des Roboters mit vordefinierten Blöcken reicht oft nicht aus, um ein Programm zu entwickeln, welches den eigenen Wünschen entspricht. Hierzu ist es notwendig eigene Blöcke zu entwerfen. Eigene Blöcke können aus Kombination bereits vorhandener Blöcke entstehen oder komplett neu entwickelt werden. Eine

Neuentwicklung ist meist für spezielle Bewegungsabläufe notwendig. Hierzu bietet Choregraphe verschiedene Möglichkeiten an.
Alle Arten der Neuerstellung beginnen damit, den Block *Animation* in das Flussdiagramm zu ziehen. Dieser Block stellt einen leeren Bewegungsblock dar. Durch Doppelklick auf diesen Block wird eine neue Ansicht geöffnet, in der eine leere Zeitleiste zu sehen ist. In diese Zeitleiste können nun verschiedene Positionen eingetragen werden, die der Roboter ansteuern soll. Abbildung 2.7 zeigt die Zeitleiste. Durch Kli-

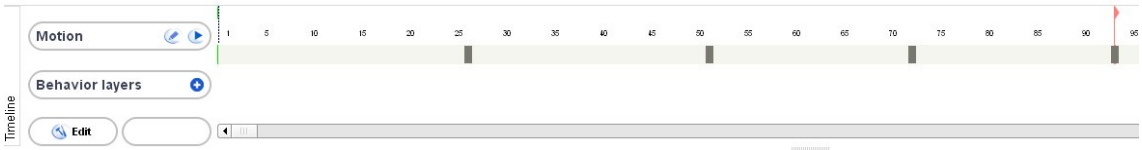


Abb. 2.7.: Zeitleiste [Ald11]

cken in die Zeitleiste wird ein Frame ausgewählt. Anschließend können die Gelenkstellungen, welche in diesem Frame erreicht werden sollen, über das in Abbildung 2.6 gezeigte Dialogfeld eingestellt werden. Dieser Vorgang kann für verschiedene Frames wiederholt werden, sodass am Ende ein bestimmter Bewegungsablauf resultiert.
Das Einstellen der Gelenkwinkel über das Dialogfeld kann allerdings mühselig werden, da die Gelenkwinkel einzeln geändert werden müssen. Will man z.B. die Hand vor den Mund führen und stellt zunächst die Schulterwinkel ein und danach die Ellenbogenwinkel, so kann es passieren, dass die zuvor eingestellten Winkel für das Schultergelenk nicht passen. Eine passende Position muss dann meist iterativ durch ständige Anpassungen der Winkel erfolgen, bis das Ergebnis zufriedenstellend ist. Eine einfachere Möglichkeit bietet sich, wenn ein Roboter mit Choregraphe verknüpft ist. In diesem Fall kann der Arm des verknüpften Roboters einfach in die gewünschte Position gebracht werden. Anschließend können die Winkel im Dialogfeld eingelesen und gespeichert werden. Noch schneller geht es über einen Rechtsklick in die Timeline. In dem sich öffnenden Menü kann die Option *Store joints in Keyframe* für den entsprechenden Körperbereich gewählt werden. Somit sind die gerade eingestellten Winkel des verknüpften Roboters in einem Frame gespeichert.

Ist eine Bewegung mit mehreren Frames erstellt, so kann über den *Timeline Editor*, die Interpolation der Bewegung zwischen den einzelnen Frames eingestellt werden. Zudem sind Änderungen der Winkel und Zeiten über diesen Editor ebenfalls möglich. Für jedes Gelenk ist hierzu eine eigene Trajektorie vorhanden. Abbildung 2.8 zeigt ein Beispiel für die Trajektorien einer Bewegung.
Ist die Bewegung fertig erstellt, kann abschließend noch eingestellt werden, wie viele Frames in einer Sekunde abgearbeitet werden sollen. Somit lässt sich die Geschwindigkeit der Bewegung beeinflussen.

Hinter einigen Blöcken verstecken sich allerdings keine Zeitleisten mit Frames oder Trajektorien, sondern Skripte, die andere Module aufrufen. Diese Module sind meist in C++ programmiert. Es ist also auch möglich, mit Choregraphe eigene Module, die in Hochsprachen entwickelt wurden, als einfache Blöcke in Choregraphe zu imple-

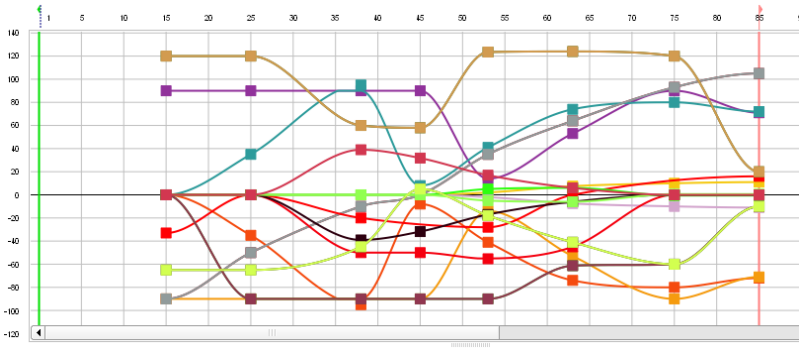


Abb. 2.8.: Timeline Editor mit Trajektorien für verschiedene Gelenke [Ald11]

mentieren und somit anderen Benutzern zugänglich zu machen.

Choregraphe bietet über die hier vorgestellten Möglichkeiten noch viele weitere Funktionen und Optionen, auf die hier allerdings nicht weiter eingegangen werden soll.

Die Vielfältigkeit und Benutzerfreundlichkeit von Choregraphe machen dieses Programm zu einem nützlichen Werkzeug für die Arbeit mit dem *Nao*-Robotiksystem, sowohl für Einsteiger, als auch für Experten.

2.2.2. Telepathe

Telepathe ist ein kleines Programm, welches zwei Funktionen bietet. Zum einen lässt sich das Kamerabild des *Naos* auf einem Monitor darstellen, zum anderen dient es zur Überwachung von Variablen, die im Speicher abgelegt sind. Zu diesen Variablen zählen unter anderem alle Sensorwerte.

Nach Öffnen des Programmes muss zunächst zwischen den beiden Funktionen entschieden werden. Die Anzeige des Kamerabildes funktioniert und ist eine nützliche Funktion, um den Roboter zu überwachen, wenn er aus dem eigenen Sichtfeld verschwunden ist. Allerdings ist die gleiche Funktion auch in *Choregraphe* implementiert.

Die Überwachung der Speicherinhalte ist einigen Einschränkungen unterworfen. Zum einen können die Inhalte numerisch dargestellt werden, zum anderen grafisch über der Zeit. Die zweite Option wäre optimal geeignet, um Messkurven aufzuzeichnen, wäre da nicht das Problem, dass ein Abspeichern der angezeigten Kurven nicht möglich ist. Ein weiteres Problem besteht in der Abtastrate, die auf maximal 10 Hz beschränkt ist. Die meisten Prozesse auf dem Roboter laufen allerdings mit höheren Wiederholraten (bis zu 100 Hz) ab, sodass keine präzisen Messreihen aufgezeichnet werden können. Ein Ablesen der numerischen Werte aus der Kurve ist ebenfalls schwierig, da die Achsenskalierung nur durch den minimal sowie maximal aufgezeichneten Werten und der Null besteht. Ein Zoomen in die Kurven oder ein Auswählen

eines Messpunktes ist ebenfalls nicht möglich, und der dargestellte numerische Wert der Variablen, entspricht immer nur dem aktuellen Wert. Eine Historie der Werte ist nicht vorhanden.

Die Funktionen von *Telepathe* sind insgesamt sehr eingeschränkt und teilweise ebenfalls in anderen Programmen verfügbar. Ein wissenschaftliches Arbeiten mit diesem Programm ist nicht möglich, weshalb es in dieser Arbeit nicht zum Einsatz kommen wird.

2.2.3. NaoSim

NaoSim ist eine Simulationsumgebung, die mit *Choregraphe* und *Telepathe* verknüpft werden kann. Der Funktionsumfang von *NaoSim* ist sehr eingeschränkt, allerdings ist der Simulator auch nicht für den professionellen Einsatz gedacht.

Durch Verknüpfen von *NaoSim* mit *Choregraphe* können die erstellten Programme zunächst im Simulator getestet werden, bevor die Übertragung auf den richtigen Roboter erfolgt. Hierdurch kann überprüft werden, ob bestimmte Bewegungen einen Sturz des Roboters herbeiführen würden oder nicht. Zur Erinnerung: Das 3D-Modell in Choregraphe berücksichtigt die Schwerkraft nicht, sodass dort keine Stürze erkannt werden können.
Des Weiteren kann überprüft werden, ob das Programm auf bestimmte Sensorik entsprechend reagiert oder nicht. Hierzu können Objekte im Simulator platziert werden, die als Hindernisse dienen können und z.B. per Ultraschall oder Kamera erkannt werden sollen.

NaoSim ist ein kleiner Simulator mit geringer Funktionsvielfalt. Für das schnelle Testen von Programmen, die mit *Choregraphe* erstellt wurden, ist er aber durchaus geeignet.

2.2.4. NaoQi Middlelayer Framework

In den vorigen Abschnitten wurden Programme vorgestellt, die die grafische Programmierung des Roboters ermöglichen bzw. unterstützen. Da grafische Programme mit steigender Komplexität allerdings schnell sehr unübersichtlich und fehleranfällig werden, ist bei größeren Programmen die Nutzung einer Hochsprache zu empfehlen. Mögliche Hochsprachen zur Programmierung sind C++ Urbi, Python und .Net. Hierzu stellt Aldebaran Robotics ein SDK bereit.

Das SDK beinhaltet die meisten Module des Herstellers sowie Funktionen, die zur Ansteuerung derselbigen dienen. Die Kommunikation zwischen den Modulen geschieht über das Middlelayer *NaoQi*. *NaoQi* wird beim Start des Roboters automatisch geladen, kann aber auch am Rechner simuliert werden. Die Simulation kann ebenfalls mit den bereits vorgestellten Programmen genutzt werden, falls kein *Nao*-Roboter zur Verfügung steht.
Da nicht alle Module des Herstellers in dem SDK vorhanden sind, können ebenfalls

nicht alle Module in der simulierten *NaoQi*-Version genutzt werden. Dies betrifft Module, welche direkt auf Komponenten des Roboters zugreifen.

Im Nachfolgenden wird die Struktur des *NaoQi* Frameworks erklärt.

Modularer Aufbau

Wie bereits erwähnt, stehen einige Herstellermodule zur Verfügung. Jedes Modul ist mit einem Broker verknüpft. Ein Broker ist ein ausführbares Programm, das eindeutig durch seine IP-Adresse und den Port bestimmt ist. Abbildung 2.9 zeigt drei Broker mit angebundenen Modulen. An den dargestellten *mainBroker*, der stan-

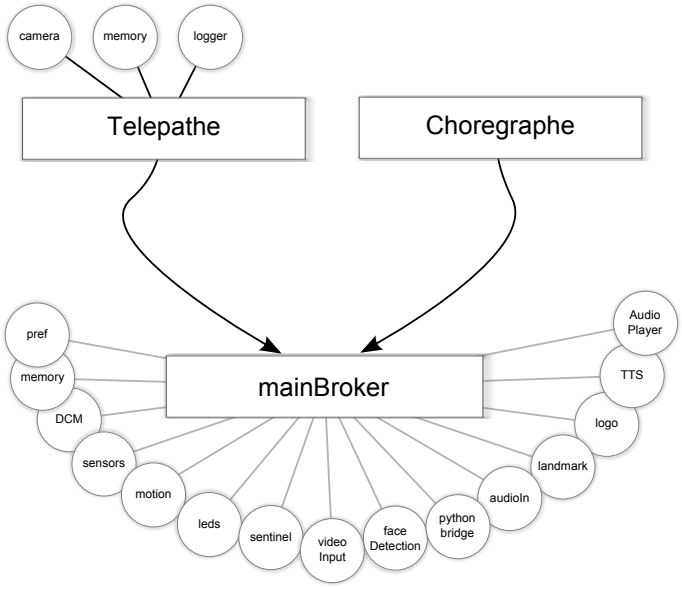


Abb. 2.9.: Verschiedene Broker mit angebundenen Modulen [Ald11]

dardmäßig von *NaoQi* bereitgestellt wird, sind die Module angebunden, die auf dem *Nao* verfügbar sind. Hierzu zählen unter anderem das Modul *memory* und das Modul *DCM*, die für diese Arbeit hauptsächlich genutzt werden. Ein Modul stellt im Allgemeinen verschiedene Funktionen bereit. So verfügt das Modul *memory* z.B. über Funktionen zum Einfügen und Auslesen von Daten im Speicher. Um aus einem anderen Modul heraus eine Funktion des *memory*-Moduls aufrufen zu können, ist die Erstellung eines Proxys notwendig. Ein Proxy stellt die Verknüpfung zwischen Modulen her und ist für das Aufrufen von Funktionen notwendig. Die Funktionen, welche über einen Proxy aufgerufen werden können, müssen allerdings als solche Funktionen kenntlich gemacht werden. Hierzu stellt der Hersteller eine BIND-Funktion bereit.

Beispiel: Das Modul *motion* verfügt über verschiedene Funktionen, die zur Bewegungssteuerung des Roboters dienen. Eine der Funktionen lässt den Roboter einen Schritt in eine bestimmte Richtung gehen. Soll nun vor Ausführung dieser Funktion überprüft werden, ob der Roboter mit den Füßen auf dem Boden steht, so können

die im Modul *memory* abgelegten Sensorwerte der Drucksensoren im Fuß abgefragt werden. Im Modul *motion* wird hierzu ein Proxy für das Modul *memory* erzeugt. Da beide Module mit dem selben Broker verknüpft sind, kennt der Broker beide Module und kann den Proxy erstellen. Anschließend kann der Proxy dazu verwendet werden, die Funktion zum Auslesen der Sensorwerte, die vom Modul *memory* bereitgestellt wird, aufzurufen. Sind die beiden Module, die über einen Proxy verknüpft werden, wie in diesem Beispiel, mit dem selben Broker verknüpft, so wird ein Funktionsaufruf zwischen den Modulen als „lokaler“ Aufruf bezeichnet.

Die Broker *Telepathe* und *Choregraphe* sind die bereits vorgestellten Programme, welche ebenfalls mit Modulen verknüpft sein können. Die Pfeile zum *mainBroker* symbolisieren, dass auch diese Broker mit den Modulen des *mainBrokers* verknüpft werden können. Hierzu wird beim Erstellen des Proxys die IP-Adresse und der Port des *mainBrokers* übergeben. Da die Programme *Telepathe* und *Choregraphe* nicht auf dem Roboter, sondern auf einem Rechner laufen, kennen diese Broker die Module des *mainBrokers* nicht. Daher ist die Angabe der IP-Adresse und des Ports des *mainBrokers* notwendig. Sollen nun Funktionen der Module des *mainBrokers* vom Rechner aufgerufen werden, ist die gerade beschriebene Verknüpfung notwendig. Ein Aufruf von einer Funktion eines Moduls, das, wie in diesem Beispiel, mit einem anderen Broker verknüpft ist, wird als „remote“-Aufruf bezeichnet.

Aufrufe von Modulfunktionen sind unabhängig von der Programmiersprache. So können z.B. Module, deren Funktionen in C++ entwickelt wurden, von einem Python-Modul aufgerufen werden und umgekehrt.

Lokale und Remote-Module

Im vorigen Abschnitt wurde der Unterschied zwischen lokalen- und Remote-Aufrufen zwischen verschiedenen Modulfunktionen erklärt. Ein Modul, das Aufrufe per remote vornimmt, wird im Folgenden als Remote-Modul bezeichnet.
Der Hersteller schlägt vor, alle selbst entwickelten Module als Remote-Module zu erstellen. Die Entscheidung, ob ein Modul ein lokales oder ein Remote-Modul ist, kann beim Erstellen des Moduls getroffen werden. Der Vorteil eines Remote-Moduls ist, dass bei fehlerhaftem Verhalten nur der eigene Broker abstürzen kann. Der *mainBroker*, mit dem die vom Hersteller entwickelten Module verknüpft sind, läuft weiter. Hierdurch können Stürze und somit Beschädigungen am Roboter vermieden werden. Abbildung 2.10 zeigt nochmals die Remote-Architektur.

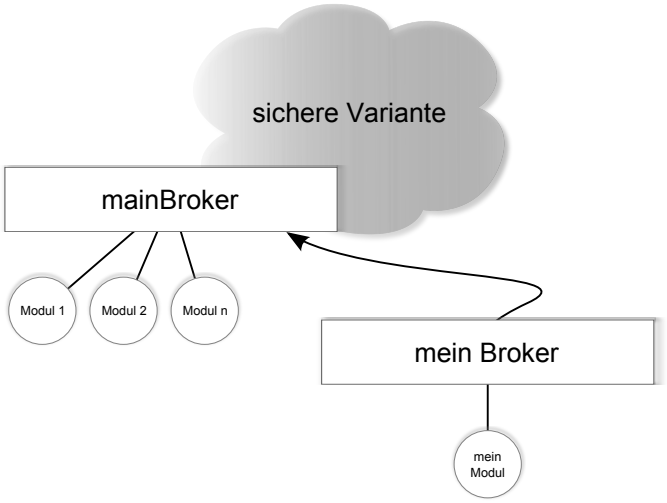


Abb. 2.10.: Sichere Remote-Architektur [Ald11]

Der Nachteil von Remote-Modulen ist, dass sie nicht echtzeitfähig sind und die Kommunikation zwischen zwei Remote-Modulen mit einer Zyklusdauer von 100 ms sehr langsam ist. Des Weiteren sind manche Funktionen einzelner Module nicht per remote verfügbar, wie z.B. das Auslesen der Adresspointer des *memory* Moduls. Die Adresspointer garantieren einen schnellen Zugriff auf den Speicher, was für Echtzeitanwendungen wichtig ist.

Da lokale Module keinen dieser Nachteile besitzen, sind dies auch gleichzeitig die Vorteile lokaler Module. Der Nachteil ist natürlich, dass diese Module direkt mit dem *mainBroker* verknüpft sind und bei fehlerhaftem Verhalten einen Systemabsturz des Roboters verursachen, der wiederum zu Stürzen und Beschädigungen führen kann. Die Architektur eines lokalen Moduls ist in Abbildung 2.11 dargestellt.

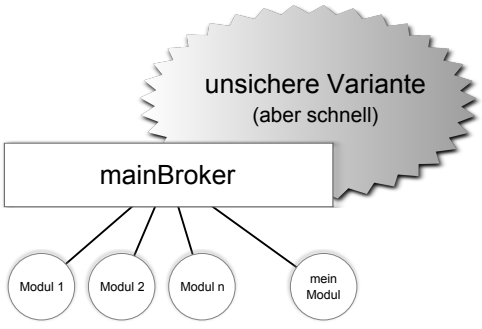


Abb. 2.11.: Unsichere aber schnelle lokale Architektur [Ald11]

Aus den Vor- und Nachteilen der beiden Module muss eine Entscheidung für das eigene Modul getroffen werden. Für einige Anwendungen sind Remote-Module sehr gut geeignet. Gerade dann, wenn sehr viel mit den Herstellermodulen gearbeitet wird. In dieser Arbeit wurden, je nach Anforderung, beide Modultypen verwendet.

Wichtige Standardmodule

In dieser Arbeit werden hauptsächlich zwei Standardmodule, welche vom Hersteller entwickelt wurden, eingesetzt. Zum einen ist dies das Modul *memory* (*ALMemory*), zum anderen das *DCM*-Modul (*DCM*). Diese Module bilden zudem die Grundlage für das *NaoQi* Framework, da fast alle anderen Module Funktionen dieser beiden Module nutzen. Aus diesen Gründen werden diese Module im Nachfolgenden genauer erklärt.

Das Modul *memory* stellt den Speicher des *Naos* dar. Es verfügt über Funktionen zum Schreiben und Lesen von Daten im Speicher. Jede abgelegte Speichervariable verfügt über einen eindeutigen Namen. Wird die Lesefunktion aufgerufen, so muss ihr der Name der auszulesenden Variable übergeben werden. Die Funktion sucht anschließend im Speicher nach der zugehörigen Variable und gibt diese zurück. Dieser Aufruf ist sowohl per remote als auch lokal verfügbar. Da für jedes Auslesen der Speicher nach der passenden Variablen durchsucht wird, ist diese Methode nicht sehr schnell. Um große Datenmengen auszulesen, können Listen von Variablen erstellt werden, die dann mit einem Aufruf auslesbar sind. Diese Art des Zugriffs nimmt weniger Zeit in Anspruch als dies beim einzelnen Auslesen jeder Variablen der Fall wäre. Die schnellste Möglichkeit Daten auszulesen, ist allerdings der direkte Zugriff auf den passenden Speicherbereich. Hierzu bietet das Modul eine Funktion an, die den passenden Adresspointer zu einer Variablen zurückgibt. Dieser Aufruf dauert zwar länger als ein normales Auslesen der Variablen, muss allerdings auch nur einmal getätigt werden. Anschließend kann der Speicherinhalt der im Pointer abgelegten Adresse direkt ausgelesen werden, ohne dass ein Durchsuchen des Speichers notwendig ist. Diese Funktion ist allerdings nur bei lokalen Modulen nutzbar. Die gerade beschriebene Vorgehensweise zum Auslesen von Daten aus dem Speicher lässt sich auf das Schreiben von Daten in den Speicher übertragen.

Das *DCM*-Modul (Device-Communication-Manager-Modul) stellt die Verbindung zwischen den elektronischen Komponenten des Roboters her. Es bildet die Schnittstelle von der höheren Softwareebene, zu denen die Module gehören, zu der tieferen Softwareebene, die zur Ansteuerung der einzelnen Komponenten dient. Eine Übersicht der Komponenten ist in Abbildung 2.12 dargestellt. Konkret ermöglicht der *DCM* die Kommunikation mit allen Sensoren und Aktoren des Roboters. Einige Operationen werden vom *DCM* automatisch und zyklisch ausgeführt. So liest das Modul alle 10 ms fast alle Sensorwerte und schreibt sie in das *memory*-Modul. Manche Sensoren, wie z.B. die Ultraschallsensoren, benötigen zur Aktualisierung einen Befehl an einen Aktor, bevor ein Auslesen und Schreiben in den Speicher stattfindet. Diese Sensoren werden nicht zyklisch vom *DCM* ausgelesen. Der 10 ms-Zyklus des Moduls wird zudem als separater Echtzeitthread zur Verfügung gestellt. An diesen können Funktionen, die in Echtzeit ablaufen müssen, angebunden werden.
Des Weiteren müssen alle Befehle an die Aktoren über den *DCM* geschickt werden. Zuvor wurde beschrieben, dass ein Lesen und Schreiben für Variablen des *memory*-Moduls möglich ist. Das Schreiben ist allerdings nur für bestimmte Variablen, wozu

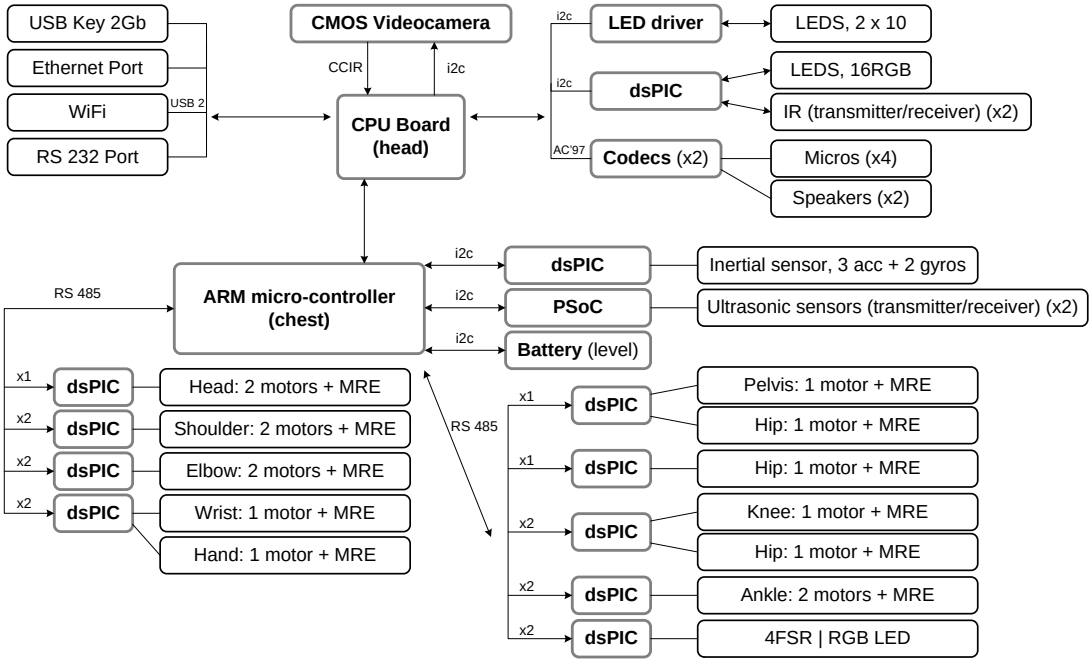


Abb. 2.12.: Übersicht der Roboterkomponenten [Ald11]

auch alle selbst definierten Variablen zählen, zulässig. Um einen Befehl an einen Aktor zu senden, muss der Befehl an das *DCM*-Modul gesendet werden, das den Befehl zum einen an den Aktor sendet und zum anderen im Modul *memory* hinterlegt. Dies geschieht allerdings nur, wenn der angesprochene Aktor den Befehl bestätigt[1]. Abbildung 2.13 verdeutlicht die Einbindung des *DCM* in das *NaoQi*-Framework. Beide hier vorgestellten Module sind bei Simulation von *NaoQi* am Rechner nicht im vollen Funktionsumfang verfügbar. Der *DCM* wird überhaupt nicht simuliert. Der Grund ist, dass in der Simulation sämtliche Hardware fehlt, die angesteuert werden kann. Da der *DCM* fehlt, ist auch ein Arbeiten mit dem *memory*-Modul in der Simulation nicht in vollem Umfang möglich. Es werden keine Daten vom *DCM* in das Modul geschrieben. Für einen Großteil der Daten wäre es möglich, diese durch ein eigenes Modul in den Speicher zu schreiben. So könnten z.B. auch simulierte Sensorwerte im *memory*-Modul abgelegt werden. Einige Variablen dürfen allerdings nur vom *DCM* geschrieben werden, sodass auch hier nur eine eingeschränkte Nutzung möglich ist.

[1]Diese Funktion befand sich während der Erstellung dieser Arbeit noch in der Entwicklung.

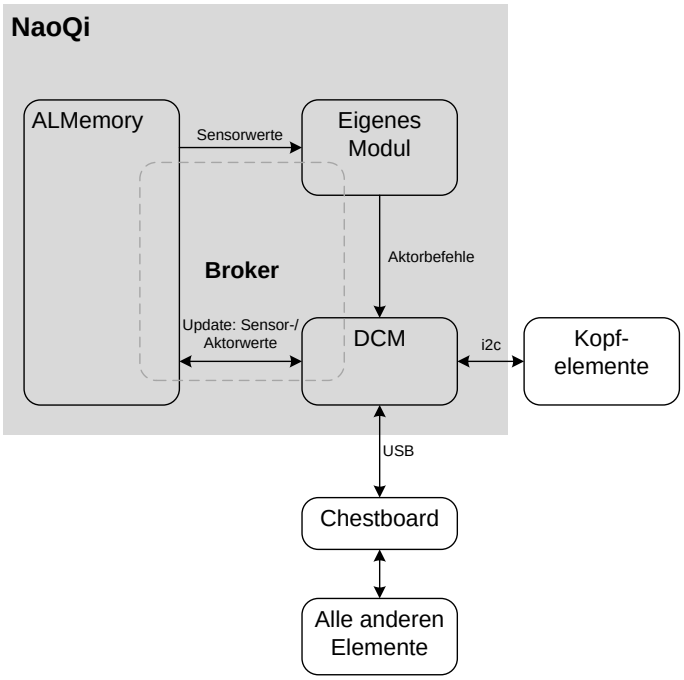


Abb. 2.13.: Einbettung des *DCM* in das *NaoQi*-Framework [Ald11]

3. Entwicklung einer Software zur Messdatenaufzeichnung

Einer der ersten Schritte dieser Arbeit ist die Entwicklung einer Software zur grafischen Messdatenaufzeichnung. Wie bereits in Abschnitt 2.2.2 erklärt, eignet sich das Programm *Telepathe* nicht dazu, Messdaten aufzuzeichnen und zu analysieren. Für ein wissenschaftliches Arbeiten mit dem Roboter ist es allerdings notwendig, Messkurven aufzeichnen und vor allem auch abspeichern zu können. Aus diesem Grund ist die Entwicklung eines solchen Programmes zwingend notwendig gewesen.

3.1. Anforderungen

Das neu entwickelte Programm soll keine der Schwachstellen von *Telepathe* aufweisen. Hierzu werden folgende Anforderungen an das Programm gestellt:

- Grafische Darstellung der Messkurven über der Zeit,
- einfache Auswahlmöglichkeit der aufzuzeichnenden Variablen,
- Möglichkeit Messdaten abzuspeichern, wobei das Format der Daten so wählbar sein soll, dass ein Importieren der Daten in andere Programme (z.B. *Matlab*) möglich ist,
- Abtastrate für die Aufzeichnung soll bei 100 Hz liegen (da dies die höchste Frequenz ist, mit der die Sensorwerte auf dem Roboter ausgelesen und im Speicher abgelegt werden),
- Zoomfunktion in die grafischen Messkurven zur schnellen Analyse der Daten,
- Auslesen der numerischen Werte von Datenpunkten in der Messkurve.

3.2. Umsetzung

Für die Umsetzung der gestellten Anforderungen ist es zunächst notwendig, Daten vom Roboter auf den PC zu übertragen. Zur Erinnerung: Die Daten der Sensoren, Aktoren sowie einige andere Variablen werden vom *DCM* in den Speicher des Roboters geschrieben. Dieser ist über das *memory*-Modul zugänglich und macht einen Zugriff auf dieses Modul erforderlich. Abbildung 2.9 zeigt, wie *Telepathe* diesen Zugriff realisiert. Es ist ein remote-Zugriff, welcher der Einschränkung einer Wiederholrate

von 10 Hz unterliegt. Für die gestellten Anforderungen ist eine remote-Übertragung also nicht möglich.
Benötigt wird ein schneller Zugriff auf Speicherinhalte des *memory*-Moduls, was nur durch ein lokal an den *mainBroker* angebundenes Modul erreicht werden konnte.
Dieses Modul wird *memoryLogger* genannt. Es bietet nun die Möglichkeit, Daten schnell aus dem Speicher auslesen zu können. Allerdings sind die Daten dann lokal in diesem Modul gespeichert, das auf dem Roboter läuft. Es bleibt das Problem, die Daten vom Roboter auf den PC zu übertragen. Da eine remote-Übertragung zu langsam ist, muss eine eigene Verbindung erstellt werden. Hierzu musste auf dem Rechner ein Server und auf dem Roboter ein Client erstellt werden. Als Protokoll der Übertragung wird TCP gewählt.
Das Programm auf dem PC, das den Server der Verbindung aufbaut, wird *NaoObserve* genannt. Es ist zugleich das Programm, das später zur Aufzeichnung der Messdaten dient.
Der Aufbau einer TCP-Verbindung erfordert allerdings, dass der Server die IP-Adresse des Clients kennt, und umgekehrt. Um dem Roboter mitzuteilen, welche IP-Adresse der PC hat, auf dem NaoObserve läuft, wird die Remoteverbindung zur Übertragung gewählt. Abbildung 3.1 veranschaulicht dies.

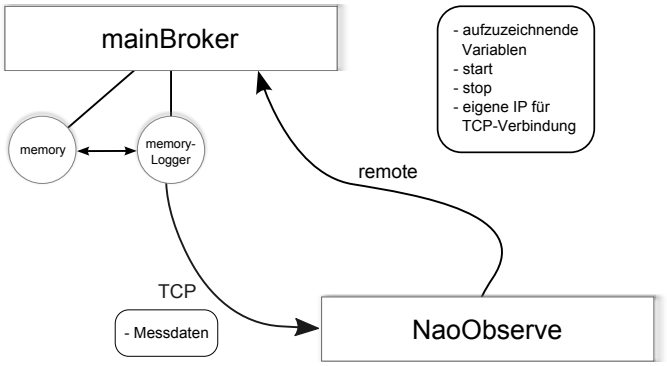


Abb. 3.1.: Kommunikationsarchitektur zwischen NaoObserve und dem *memoryLogger*-Modul

Die Remoteverbindung wird des Weiteren zur Übertragung von Daten genutzt, die die Programmaufzeichnung steuern. Hierzu gehört die Übertragung der Variablennamen, die aufgezeichnet werden sollen sowie die Information zum Starten und Stoppen der Aufzeichnung.

Die weiteren Anforderungen des Programmes sind mit dem Entwurf der grafischen Benutzeroberfläche verbunden. Die Oberfläche wurde in C++ unter Nutzung des Pakets *Qt* (engl. Aussprache: *cute*) erstellt. *Qt* wurde von dem norwegischen Unternehmen *Trolltech* entwickelt und 2008 von *Nokia* aufgekauft.
Das Paket bietet eine große Vielfalt an vordefinierten Klassen, die zur Erstellung von grafischen Benutzeroberflächen dienen.

Die Anforderungen, eine grafische Darstellung der Messdaten zu ermöglichen und

eine einfache Auswahlmöglichkeit für aufzuzeichnende Daten zu gewährleisten, konnten mit Hilfe von einigen *Qt*-Klassen erreicht werden. Das Ergebnis ist in Abbildung 3.2 zu sehen. Im Bereich 1 ist ein Variablenbaum zu erkennen. In dieser Baumstruk-

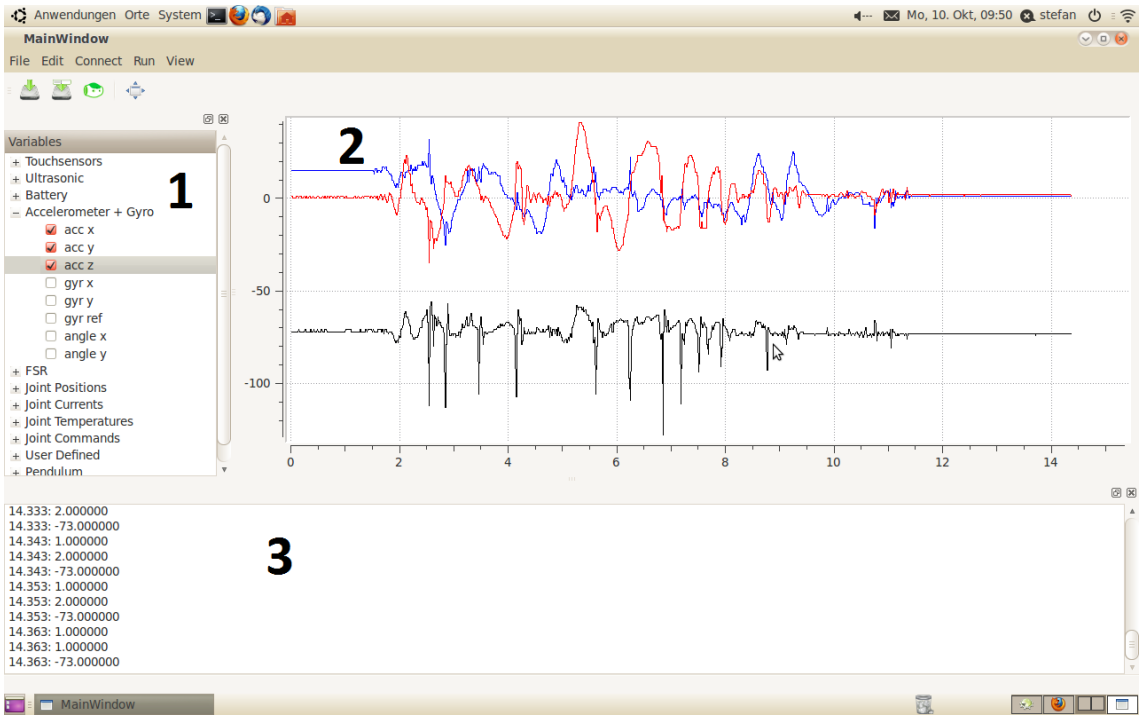


Abb. 3.2.: Grafische Benutzeroberfläche von NaoObserve

tur sind einige Variablen, die im *memory*-Modul des Roboters liegen, aufgeführt. Dies betrifft unter anderem alle Sensorwerte. Variablen können in dieser Struktur durch einfaches Anklicken zur Aufzeichnung ausgewählt werden. In der Abbildung sind die Sensorwerte der Beschleunigungssensoren zur Aufzeichnung selektiert.
Der Bereich 2 dient zur grafischen Darstellung der Messwerte über der Zeit. In diesem Bereich ist sowohl das Zoomen als auch das Auslesen der numerischen Werte durch Klicken auf einen Messpunkt möglich. Der Bereich 3 dient als Ausgabekonsole des Programmes. Hier werden Statusinformationen und Messwerte ausgegeben.

Um eine Aufzeichnung starten zu können, muss *NaoObserve* zunächst eine remote-Verbindung zum *memoryLogger*-Modul aufbauen. Hierzu gibt es in *NaoObserve* eine Funktion, die ein Dialogfeld öffnet, in dem IP-Adresse und Port des *mainBrokers* angegeben werden müssen. Die IP-Adresse ist die des Roboters. Der Port des *mainBrokers* ist standardmäßig 9559 und ist bereits im Dialogfeld voreingestellt. Durch Erstellen der remote-Verbindung wird eine weitere Funktion aufgerufen, die dem *memoryLogger*-Modul die IP-Adresse des PCs mitteilt und den Aufbau der TCP-Verbindung veranlasst.
Vor dem Start der Aufzeichnung müssen zunächst die aufzuzeichnenden Variablen ausgewählt werden, danach erst kann die Aufzeichnung gestartet werden. Durch Starten einer Aufzeichnung werden verschiedene Befehle nacheinander ausgeführt. Zu Beginn werden die aufzuzeichnenden Variablen an das *memoryLogger*-Modul

übertragen. Das *memoryLogger*-Modul liest anschließend die Adresspointer der Variablen vom *memory*-Modul aus. Ist dies erfolgt, wird ein Prozess gestartet, der mit einer Wiederholrate von 100 Hz die entsprechenden Variableninhalte liest, verpackt und über die TCP-Verbindung verschickt. *NaoObserve* überprüft ebenfalls mit einer Wiederholrate von 100 Hz, ob Daten am Port der TCP-Verbindung anliegen. Ist dies der Fall, werden die Daten ausgelesen, verarbeitet und grafisch dargestellt. Während einer Messdatenaufzeichnung basiert die Kommunikation zwischen *NaoObserve* und dem *memoryLogger*-Modul allein auf der TCP-Verbindung.
Zum Beenden der Aufzeichnung wird allerdings die remote-Verbindung genutzt, um dem *memoryLogger*-Modul mitzuteilen, dass keine Daten mehr gesendet werden sollen.

Ist eine Aufzeichnung beendet, so können die Messkurven zum einen direkt im Programm analysiert werden und sind zum anderen auch abspeicherbar, um mit anderen Programmen, die mehr Funktionen zur Messdatenaufbereitung bieten, weiter analysiert werden zu können. Die Messdaten werden hierzu in eine einfache Textdatei gespeichert. In der Kopfzeile der Datei stehen die Variablennamen, danach folgen die durch „;“ getrennten Messdaten. Eine Datei in diesem Format kann von vielen anderen Programmen, wie z.B. *Matlab* importiert werden.

NaoObserve erfüllt die gestellten Anforderungen und umgeht die Probleme von Telepathe durch ein zusätzliches Modul und zusätzliche Funktionen, allerdings gibt es auch bei *NaoObserve* einige Dinge zu beachten.

Das *memoryLogger*-Modul liest und sendet Daten in einem 10 ms-Zyklus (100 Hz). Dieser Zyklus ist an den *DCM* angebunden. Wird eine Vielzahl von Daten aufgezeichnet, so kann es vorkommen, dass die 10 ms nicht ausreichen. In diesem Fall wird der *DCM*-Zyklus verändert. Diese Veränderung kann Auswirkungen auf andere Module haben, die ebenfalls den 10 ms-Zyklus nutzen. Wird die Zykluszeit zu stark verändert, so ruft *NaoQi* eine Sicherheitsroutine auf, die die Steifheit der Motoren langsam reduziert. Steht der Roboter in diesem Moment, so kann er hierdurch umfallen. Experimentell wurde ermittelt, dass eine Aufzeichnung bis zu 10 Variablen keine Probleme bereitet. Lediglich am Anfang der Aufzeichnung kann es durch die Initialisierung der Adresspointer zu Verzögerungen kommen. Bei mehr als 10 Variablen kam es während der Aufzeichnung in einigen Fällen zu Aussetzern des *DCM*.
Eine Möglichkeit dies zu umgehen, wäre, die Messdaten nicht alle 10 ms zu übertragen, sondern lokal auf dem Roboter zu sammeln. Eine Übertragung an *NaoObserve* während der Aufzeichnung könnte durch eine Anfrage des Servers stattfinden. Somit könnten Daten nur dann übertragen werden, wenn beide Seiten genügend Ressourcen hierfür bereit stellen können. Somit wären zwar während der Aufzeichnung nicht alle Daten vorhanden, nach Beenden der Aufzeichnung könnte die lokal gespeicherte Datei aber abgerufen werden. Somit könnten auch die zuvor nicht übertragenen Messwerte nachträglich offline analysiert werden.
Eine weitere unvorteilhafte Eigenschaft von *NaoObserve* ist, dass die grafische Darstellung der Messwerte zeitverzögert abläuft. Dies ist zum einen durch die Übertragungsdauer verursacht, zum anderen aber dadurch, dass die Daten nicht in Realzeit

verarbeitet werden. Hierdurch entsteht über die Dauer der Aufzeichnung ein immer größerer Zeitverzug. Nach dem Beenden der Aufzeichnung werden die restlichen Daten nachträglich verarbeitet, sodass am Ende alle Daten bereit stehen. Diese Eigenschaft wurde in Kauf genommen, um vollständige Messreihen mit festen Abtastzeiten zu bekommen. Daher wurde auch die verbindungsorientierte TCP-Verbindung gewählt, die im Gegensatz zu UDP überprüft, ob gesendete Pakete ankommen und ebenso auch die Reihenfolge der Pakete sicherstellt.

Wie bereits angedeutet, lassen sich diese Probleme durch Umstrukturierungen im Programm aber beheben. Für diese Arbeit sind die bisherigen Funktionen allerdings ausreichend, weshalb die Weiterentwicklung von *NaoObserve* im Rahmen dieser Arbeit nicht weiter vorangetrieben wurde.

4. Simulationsumgebung

Zur Verwirklichung eines Laufalgorithmus für den Roboter ist die Arbeit mit einer passenden Simulationsumgebung von großer Bedeutung. Zum einen können verschiedene Ansätze getestet und somit Fehlverhalten beseitigt werden, bevor das Programm auf dem Roboter getestet wird. Hierdurch können Beschädigungen des Systems vermieden werden. Zum anderen stellt ein Simulator die Möglichkeit bereit, das Programm in einem Debug-Modus Schritt für Schritt auszuführen und Fehler systematisch zu analysieren. Aus diesen Gründen ist die Nutzung einer Simulationsumgebung in der Regel unverzichtbar.

4.1. Mögliche Simulatoren

Für diese Arbeit wird ein Simulator gesucht, der bereits über ein Modell des *Naos* verfügt. Hierdurch wird die Anzahl möglicher Simulatoren schon stark eingeschränkt. Zur Auswahl stehen neben dem bereits vorgestellten Simulator *NaoSim*, die Simulatoren *Webots* und *SimRobot*.

Wie bereits erwähnt ist der Funktionsumfang von *NaoSim* äußerst gering ist. Das Spielfeld für den *RoboCup* ist nicht integriert und die Steuerung des Roboters im Simulator kann nur durch *Choregraphe* und das simulierte *NaoQi* erfolgen. Ein *DCM*, der die Speicherinhalte aktualisiert, steht nicht zur Verfügung. Daher können auch keine direkten Befehle über den *DCM* an Aktoren gesendet werden. Hierzu müsste das herstellereigene *motion*-Modul benutzt werden. Dieses Modul soll allerdings später bei der Entwicklung des Laufalgorithmus nicht genutzt werden, da es langsamer agiert, als der *DCM*. Eine Verwendung des gleichen Quellcodes für den Simulator und den richtigen Roboter ist nicht ohne Weiteres möglich, was einen großen Nachteil darstellt, da das eigentliche Programm nicht getestet werden kann. Des Weiteren ist auch während der Simulation eine Überwachung einiger Variablen gewünscht. Hierzu wird eine plot-Funktion benötigt. Diese müsste erst entwickelt und durch ein eigenes Modul integriert werden.
Insgesamt genügt der Simulator den gestellten Ansprüchen nicht, da viel Arbeit investiert werden müsste und dennoch unterschiedlicher Quellcode für Simulator und Roboter benötigt werden würde.
SimRobot wurde in Zusammenarbeit der Universität Bremen und dem Deutschen Forschungszentrum für Künstliche Intelligenz als Open Source Projekt entwickelt und wird vom Bremer *RoboCup* Team *B-Human* genutzt.
Ein Modell des Roboters sowie das für den *RoboCup* verwendete Fußballfeld sind bereits als Objekte in dem Simulator vorhanden.
Das Problem bei diesem Simulator ist, dass es keine ausführliche Hilfe oder Tu-

torials zum Einstieg für die Arbeit mit diesem Simulator gibt. Eine Einarbeitung benötigt dementsprechend viel Zeit. Des Weiteren ist es ein Wunsch bei Erreichen des langfristig gesetzten Ziels, ein eigenes *RoboCup* Team aufzubauen, nicht auf Weiterentwicklungen anderer Teams angewiesen zu sein.

Webots ist eine professionelle Simulationsumgebung, die 1996 am Federal Institute of Technology in Lausanne (Schweiz) entwickelt und mittlerweile von der Firma Cyberbotics weiterentwickelt und vertrieben wird.
Der Simulator liefert neben einem Modell des *Naos* ebenfalls mit dem für den *RoboCup* genutzten Fußballfeld als integrierte Objekte. Des Weiteren ist der Simulator durch verschiedene Programmiersprachen steuerbar. Hierzu zählen C, C++, Matlab, Java, Python und URBI. Eine ausführliche Hilfe und Tutorials für den Einstieg in die Arbeit mit diesem Simulator stehen bereit.
Ein großer Vorteil von *Webots* ist die Möglichkeit, die Steuerung des Simulators aus *Matlab* durchzuführen. *Matlab* bietet neben der einfachen Möglichkeit, Daten grafisch darzustellen, auch eine Vielzahl an Toolboxen, die auf diese Weise genutzt werden können.
Im Gegensatz zu *NaoSim*, der bereits im Lieferumfang vorhanden ist und *SimRobot*, der kostenlos zur Verfügung steht, ist eine Lizenz in der Educational Version für *Webots* derzeit für 320CHF (ca. 260€) erhältlich.
Aufgrund der Vorteile von *Webots* gegenüber den anderen Simulatoren wird dieser Simulator aber für diese Arbeit genutzt und soll auch weiterhin als Simulationsplattform dauerhaft eingesetzt werden.

4.2. Webots

Im vorigen Abschnitt wurden bereits einige Eigenschaften des Simulators genannt. In diesem Abschnitt soll der Simulator nun genauer vorgestellt werden.
Die simulierte Dynamik des Roboters in *Webots* basiert auf der ODE (Open Dynamics Engine). Diese dient zur Detektion von Kollisionen und zur Simulation der Starrkörperdynamik. Weiterhin ermöglicht die ODE die Simulation vieler physikalischer Eigenschaften wie z.B. Reibung, Geschwindigkeit oder Trägheit.

Neben dem bereits implementierten Modell des *Naos* und des Fußballfeldes sind weitere Robotersysteme bereits integriert. Des Weiteren verfügt Webots über eine Vielzahl simulierter Sensoren und Aktoren und über viele Objekte wie z.B. Tische, Stühle, Wände und vieles mehr.

Die Steuerung des Simulators erfolgt durch einen Controller. Dieser Controller kann in verschiedenen Programmiersprachen erstellt werden, worunter auch C++ und *Matlab* fallen.
Die Geschwindigkeit, mit der eine Simulation abläuft, hängt stark von der Komplexität des Controllers, der Aktualisierungsfrequenz und den Hardwarekomponenten des Rechners ab. Um ein Gefühl für den simulierten Vorgang in Echtzeit zu bekommen, ist es möglich, ein Video der Simulation zu erstellen, welches anschließend in

Echtzeit abläuft.

Verschiedene Simulationen können als „Welten“ abgespeichert werden. Eine Welt kann verschiedene Objekte, worunter auch Roboter zählen, enthalten. Des Weiteren werden die zugehörigen Controller sowie benutzerdefinierte Einstellungen gespeichert.

Simulationsfenster

Das Simulationsfenster in *Webots* kann benutzerdefiniert angepasst werden. In Ab-

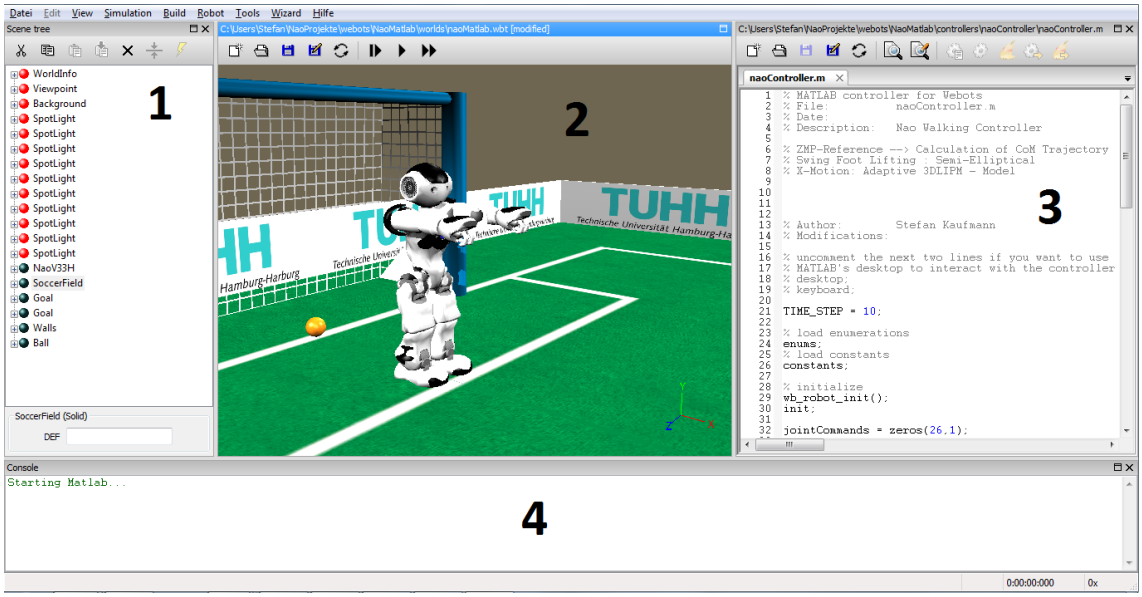


Abb. 4.1.: Webots Simulationsfenster mit geöffneten Menüs

bildung 4.1 sind die wichtigsten Menüs aufgeführt. Im Bereich 1 ist der *Scene Tree* zu sehen. In diesem sind alle Objekte aufgelistet, die zur Welt gehören. Durch Klick auf ein Objekt können einige Eigenschaften wie z.B. die Position ausgewählt und verändert werden.
Im Bereich 2 ist das eigentliche Simulationsfenster zu sehen. Hier wird die Welt dargestellt. Die Ansicht der Welt kann verschoben und gedreht werden. Des Weiteren ist es möglich, Objekte in der Welt mit Hilfe der Maus in diesem Fenster zu verschieben und zu drehen.
Im Bereich 3 ist ein Texteditor zu erkennen. In diesem können Veränderungen am Controller vorgenommen werden. Es ist somit möglich, einen *Matlab*-Controller direkt in *Webots* zu programmieren, ohne hierfür *Matlab* öffnen zu müssen. Allerdings bietet der Texteditor nicht alle Funktionen, die *Matlab* bietet wie z.B. die Autovervollständigung.
Im Bereich 4 des Bildes ist eine Ausgabekonsole zu erkennen. Über diese Konsole werden Statusinformationen wie z.B. Fehlermeldungen ausgegeben. Die Ausgabekonsole kann allerdings auch genutzt werden, um selbst Daten auszugeben und zu überwachen.

Simulationssteuerung

Wie in den vorigen Abschnitten bereits erwähnt, übernimmt ein Controller die Steuerung des simulierten Roboters. Für jeden Roboter in einer Welt muss ein Controller existieren. Neben der Steuerung des Roboters kommuniziert der Controller mit der Welt. Verfügt ein Roboter über Sensoren, so muss der Controller die Sensorwerte auslesen und bereit stellen. Auf dem tatsächlichen Roboter übernimmt der *DCM* diese Aufgabe. In *Webots* stellt der Simulator die simulierten Sensorwerte bereit. Diese müssen nun vom Controller verarbeitet werden. Gleiches gilt für die simulierten Aktoren des Roboters. Die Befehle an diese Aktoren müssen ebenfalls vom Controller gesendet werden. Die Operationen, die zwischen dem Auslesen der Sensoren und dem Senden der Aktorbefehle durchgeführt werden, sind im Idealfall für den Simulator und den echten Roboter die gleichen, sodass der gleiche Quellcode für Simulation und Roboter genutzt werden kann.

Da für den Controller verschiedene Hochsprachen zur Programmierung bereit stehen, muss eine passende Sprache gewählt werden. Da die selbst entwickelten Module für den Roboter in C++ programmiert werden, liegt es zunächst nahe, auch den Controller in C++ zu programmieren. Dies ermöglicht zudem die Nutzung des simulierten *NaoQi*. Das bereits bekannte Problem des simulierten *NaoQi* ist allerdings, dass das *DCM*-Modul nicht simuliert wird. Des Weiteren liegen keine Messwerte im *memory*-Modul, und schnelle Speicherzugriffe per Adresspointer sind in der Simulation ebenfalls nicht verfügbar. Eine Nutzung des simulierten *NaoQi* bringt also keine Vorteile. Der einzige Vorteil eines C++-Controllers ist die Möglichkeit, dass auf die eigenen Module direkt zugegriffen werden kann.

Der ausschlaggebende Grund, warum dennoch kein C++-Controller, sondern ein *Matlab*-Controller verwendet wird, ist die Vielfältigkeit, die Matlab bietet. Mit Matlab ist es mit wenigen Zeilen Code möglich, den zeitlichen Verlauf verschiedener Variablen grafisch darzustellen und somit während der Simulation zu überwachen. Für C++ müsste hierfür ein eigenes Modul entwickelt werden. Zudem bietet *Matlab* eine Vielzahl an Toolboxen, die zur Datenverarbeitung genutzt werden können. Somit können neue Konzepte schnell und einfach umgesetzt und getestet werden.

Die Nutzung von Matlab bringt allerdings auch den Nachteil mit sich, dass die bestehenden C++-Module nicht ohne Weiteres angesprochen werden können. Die Lösung dieses Problems wird im folgenden Abschnitt erläutert.

4.3. Verknüpfung von Simulator - Matlab - C++

Die Verknüpfung zwischen *Matlab* und dem Simulator ist durch *Webots* bereits vorhanden. Anders sieht dies für die Verknüpfung von Matlab zu C++ aus. Allerdings bietet Matlab die Möglichkeit C- und C++-Funktionen über sogenannte *mex*-Funktionen zu nutzen.
Eine *mex*-Funktion ist eine C- oder C++-Funktion, die von Matlab aus aufgerufen

werden kann. Diese Funktion kann wiederum andere C- oder C++-Funktionen aufrufen. Der Vorgang für C und C++ ist der Gleiche. Wird im Folgenden von C++ im Zusammenhang mit *mex*-Funktionen gesprochen, so gilt dies ebenso für C-Funktionen. Für Funktionen, die keine Daten zurückgeben und keine Parameter benötigen, ist der Aufruf aus Matlab sehr einfach möglich. Bei anderen Funktionen müssen Konvertierungen vorgenommen werden.
Parameter, die den Funktionen übergeben werden sollen, müssen ebenso in *Matlab* diese Parameter übergeben werden. Da *Matlab* alle Zahlen als Matrizen abspeichert, C++ mit diesem Format allerdings nicht arbeiten kann, muss eine Konvertierung erfolgen. Hierzu stellt die *mex*-Bibliothek Funktionen bereit.
Ebenso muss ein Rückgabewert der Funktion aus dem C++-Format in eine Matrix konvertiert werden. Im Folgenden ist ein Beispiel für die Nutzung einer C++-Funktion dargestellt, die zwei Zahlen vom Typ `float` multipliziert. Die Funktionen der *mex*-Bibliothek beginnen alle mit dem Präfix `mx`.

```
#include "mex.h"
#include "multiplier.h"
void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[])
{
    // Auslesen der übergebenen Parameter und Konvertierung nach float
    float f1 = (float) mxGetScalar(prhs[0]);
    float f2 = (float) mxGetScalar(prhs[1]);

    // Aufruf der C++ Funktion, Speicherung des Ergebnisses in prod
    float prod = Multiplier::multiply(f1,f2);

    // Konvertierung des Ausgabewertes in eine 1x1-Matrix
    plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
    double *outPtr = mxGetPr(plhs[0]);
    outPtr[0] = prod;
}
```

Ist die *mex*-Funktion erstellt, muss sie noch gespeichert und kompiliert werden. Das Kompilieren kann in *Matlab* mit dem Befehl *mex* durchgeführt werden. Diesem Befehl muss der Dateiname übergeben werden. Der Funktionsname entspricht ebenfalls dem Dateinamen. Wurde die Datei z.B. als *multiplikation.cpp* abgespeichert, so kann sie in Matlab unter diesem Namen aufgerufen werden.

```
% Aufruf in Matlab
produkt = multiplikation(5.5, 47.93);
```

Die Zugriffsmöglichkeit auf C++-Funktionen aus *Matlab* heraus ermöglicht somit die Verknüpfung von *Webots* über einen *Matlab*-Controller mit den selbst entwickelten C++-Modulen. Für alle Funktionen, die aus *Matlab* heraus nutzbar sein müssen, ist jedoch eine separate *mex*-Funktion erforderlich. Da aber viele Operationen innerhalb der Module durchgeführt werden, begrenzt sich die Anzahl der Funktionen, für die dies der Fall ist.

4.4. Quellcodekompatibiliät

Um den Simulator optimal zur Entwicklung für Module, die später auf dem Roboter laufen sollen, nutzen zu können, ist es von großem Vorteil, wenn für den Simulator und den Roboter der gleiche Quellcode genutzt werden kann. Wie bereits mehrfach erwähnt, fehlt der *DCM* in der Simulation und die Nutzung des *memory*-Moduls ist auch nur eingeschränkt möglich.

Device Communication Manager (DCM)

Die Aufgabe des *DCM* ist die Kommunikation zwischen den einzelnen Hardwarekomponenten und die Bereitstellung von Daten im Speicher. Des Weiteren müssen Befehle an die Aktoren über den *DCM* gesendet werden.
Im Simulator können die meisten Aufgaben des *DCM* über den Controller realisiert werden, sodass der Controller eine Teilsimulation des *DCM* implementiert. Das Senden von Befehlen erfolgt ebenfalls über den Controller im Simulator, allerdings bietet der *DCM* die Möglichkeit, einen Zeitpunkt für einen Aktorbefehl vorzudefinieren. Der DCM interpoliert daraufhin die Befehle, die in jedem Zyklus angelegt werden müssen. Diese Funktion ist im Simulator nicht vorhanden und muss nachgebildet werden.

Um die Nachbildung der Interpolation durchführen zu können, ist ein Verständnis der Vorgehensweise des DCM notwendig. Diese wird im Folgenden erläutert.

Ein Befehl für einen Aktor besteht immer aus dem Namen des Aktors, für den der Befehl gilt, einem Wert, den der Aktor ansteuern soll, und einer Zeit, nach welcher der Zielwert zu erreichen ist sowie weiteren Optionen. Soll z.B. die Hand innerhalb von einer Sekunde von der aktuellen Position komplett geöffnet werden, so wird ein Befehl mit diesen Werten an den *DCM* gesendet. Einer kompletten Handöffnung entspricht der Wert eins. Der *DCM* speichert zunächst diesen Befehl. Anschließend wird eine lineare Interpolation durchgeführt, deren Startwert die aktuelle Position der Hand und deren Endwert der Zielwert entspricht. Anschließend sendet der *DCM* den aktuell interpolierten Wert an den Handmotor. Dieser Vorgang wird in jedem Zyklus wiederholt durchgeführt. Dies geschieht solange, bis der Befehl abgearbeitet wurde, in diesem Fall also bis eine Sekunde vergangen ist. Anschließend liegt der Endwert des Befehls am Aktor an. Abbildung 4.2 stellt dieses Vorgehen dar. In diesem Fall wurden gleich mehrere Befehle an den Aktor gesendet. Die Interpolation wird dann jeweils zum zeitlich nächsten Befehl durchgeführt. Weitere Optionen, die beim Senden von Befehlen angegeben werden müssen, ist die Art der Befehlskombination. Wird ein Befehl an einen Aktor gesendet, an dem bereits Befehle anliegen, so werden vier Möglichkeiten im Umgang zwischen alten und neuen Befehlen zur Verfügung gestellt. Die erste Option (*ClearAll*, Abbildung 4.3(a)) besteht darin, alle alten anliegenden Befehle zu löschen und nur die neuen Befehle zu übernehmen. Die zweite Option (*Merge*, Abbildung 4.3(b)) kombiniert alte und neue Befehle. Liegen z.B. zwei Befehle an einem Aktor an, wobei einer nach einer Sekunde und der ande-

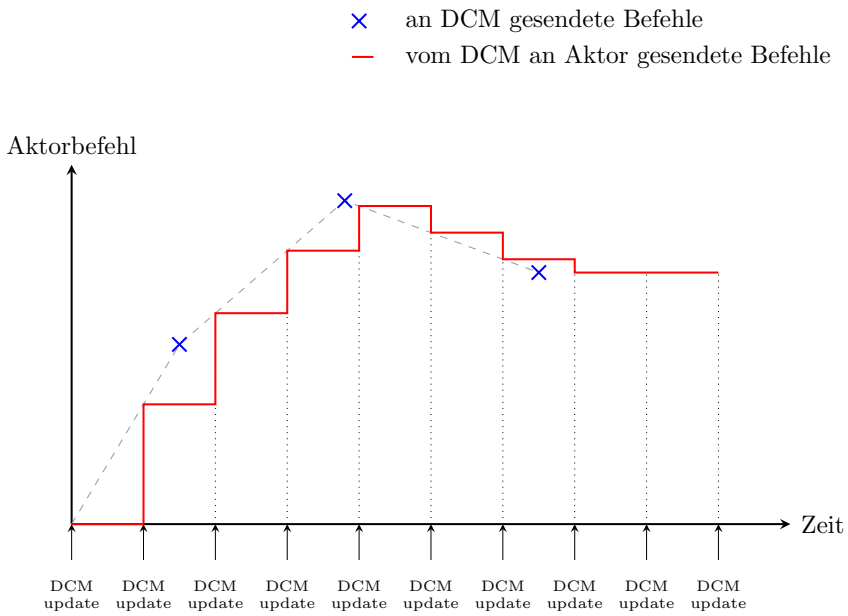
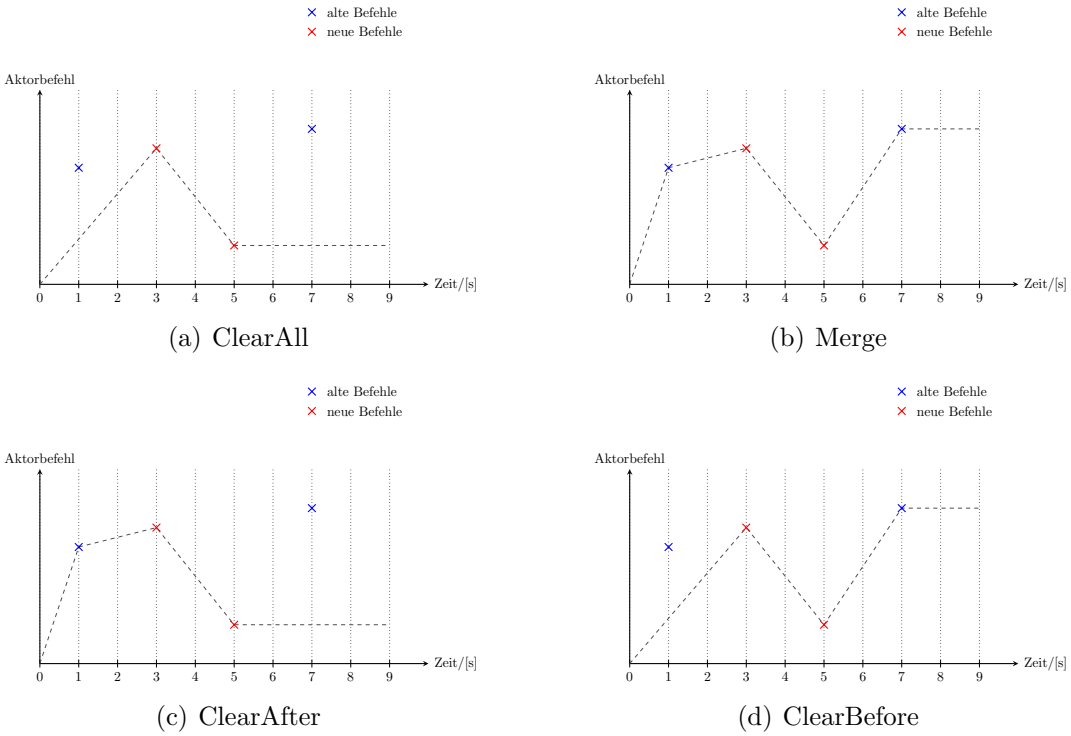


Abb. 4.2.: Interpolation von *DCM*-Befehlen [Ald11]

re nach sieben Sekunden erfüllt werden sollen und es werden nun zwei neue Befehle gesendet, die nach drei und fünf Sekunden ihren Sollwert erreichen sollen, so ist eine Kombination der Befehle möglich. Die neuen Befehl werden einfach in die vorige Befehlskette integriert. Der Hersteller warnt allerdings vor Gebrauch dieser Funktion, da es möglich ist, zwei verschiedene Sollwerte für die gleiche Zeit an einen Aktor zu senden. Die dritte Option (*ClearAfter*, Abbildung 4.3(c)) besteht in der Löschung aller alten Befehle, die nach dem ersten neuen Befehl anliegen würden. In dem zuvor genannten Beispiel würden die neuen Befehle nun nicht zwischen die alten Befehle eingeschoben, sondern der letzte alte Befehl, der nach sieben Sekunden aktiv wäre, würde gelöscht. Die letzte Option (*ClearBefore*, Abbildung 4.3(d)) besteht darin, alle alten Befehle, welche vor dem letzten neuen Befehl anliegen, zu löschen. In dem Beispiel würde somit der alte Befehl, der nach sieben Sekunden erfüllt sein soll, erhalten bleiben.

Für die meisten Bewegungen ist es notwendig, Befehle an mehrere Aktoren zu senden. Es ist zwar möglich, für jeden Aktor einen Befehl oder eine Befehlskette an den *DCM* zu übermitteln, allerdings bietet der *DCM* eine benutzerfreundlichere Alternative. Es ist nämlich möglich, sogenannte Aliases zu erzeugen. Ein Alias ist ein Synonym für eine bestimmte Liste von Aktoren. Diese Liste kann auch aus nur einem Eintrag bestehen. Somit kann z.B. ein Alias für den Aktor *HeadYaw*, der die seitliche Schwenkbewegung des Kopfes ermöglicht (siehe S. 7), mit Namen *KopfDrehung* erstellt werden. Wird nun ein Befehl an *KopfDrehung* gesendet, so weiß der *DCM*, dass *HeadYaw* gemeint ist. Diese Nutzung der Aliasfunktion verschafft allerdings keinen Vorteil beim Ansprechen der Aktoren. Anders ist dies, wenn für den Kopf ein Alias namens *Head* erzeugt wird, der die Aktoren *HeadYaw* und *HeadPitch* beinhaltet. Wichtig ist hierbei die Reihenfolge bei der Initialisierung des Alias, da der erste Befehl, der an *Head* gesendet wird, vom *DCM* an den Aktor *HeadYaw*



(a) ClearAll (b) Merge

(c) ClearAfter (d) ClearBefore

Abb. 4.3.: Optionen zur Kombination aus alten und neuen Befehlen an den *DCM*

weitergeleitet wird und der zweite Befehl an *HeadPitch*. Weitere Aliases können z.B. für die Arme und Beine erstellt werden.
Ein Befehl, der an einen Alias gesendet wird, muss immer genauso viele Befehle enthalten wie Einträge in der Alias-Liste vorhanden sind. Soll ein Aktor seine Position unverändert halten, so kann für dessen Befehl der Wert `NaN` (Not a Number) verwendet werden.

Die Nachbildung dieser Funktionen erfolgt in einem neuen C++-Modul *DCMEngine*. Das Modul bildet die für andere Module aufrufbaren Funktionen des *DCM* nach und übernimmt deren Aufgaben. Allerdings kann das Modul die interpolierten Werte nicht selbstständig an die simulierten Aktoren senden. Daher verfügt das *DCMEngine*-Modul über zwei zusätzliche Funktionen. Eine der Funktionen (*setTime*) bietet die Möglichkeit, die Zeit zu setzen. Das richtige *DCM*-Modul stellt die Zeit normalerweise selbst zur Verfügung, weshalb sie nicht gesetzt werden muss. Das simulierte Modul muss allerdings die Zeit des Simulators übernehmen, da dieser die aktuelle Simulationszeit vorgibt.
Die andere Funktion *updateCommands* dient zum Auslesen der interpolierten Aktorbefehle. Das richtige *DCM*-Modul sendet die interpolierten Werte direkt an die Aktoren. Im Simulator übernimmt dies der Controller. Dieser liest die Werte per *updateCommands* ein und schickt sie an die simulierten Aktoren.

Bei der Entwicklung der *DCMEngine* wurden allerdings nicht alle der Optionen für Befehlskombinationen (siehe Abb. 4.3) implementiert. Die Optionen *Merge* und

ClearBefore sind in der Simulation nicht verfügbar. Von der Implementierung wurde aus Zeitgründen abgesehen. Des Weiteren werden sie für diese Arbeit nicht benötigt. Eine nachträgliche Implementation in das *DCMEngine*-Modul ist bei Bedarf aber möglich.

Um in den einzelnen Modulen nicht unterscheiden zu müssen, an welches Modul (*DCM* oder *DCMEngine*) die Befehle zu senden sind, war es notwendig, ein Zwischenmodul *DCMConnector* zu entwickeln. Dieses Modul entscheidet bei der Kompilierung zwischen Simulator und Roboter und leitet die gesendeten Befehle dementsprechend weiter (siehe Abbildung 4.4). Ein weiterer Vorteil dieser Methode ist, dass

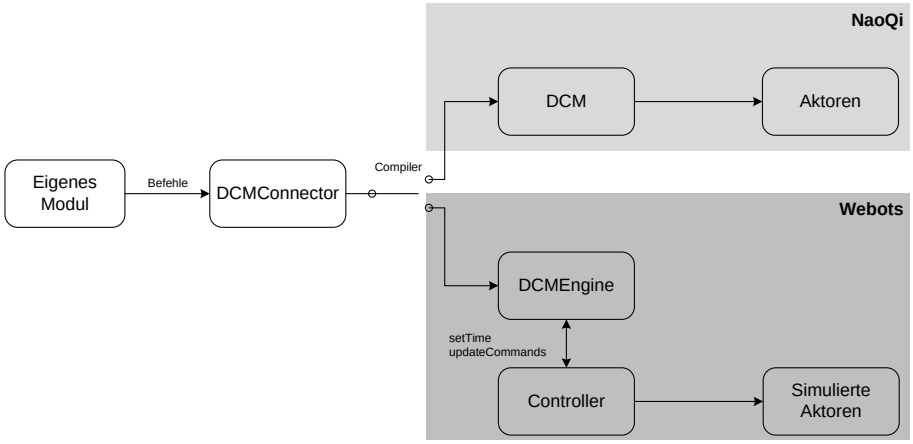


Abb. 4.4.: Unterscheidung zwischen *NaoQi* und *Webots* durch den Compiler

von der unvorteilhaften Befehlsstruktur, die das *DCM*-Modul benötigt, für den Simulator abgesehen werden kann. Der *DCM* benötigt nämlich einen Befehl vom Typ `ALValue`. Dieser Datentyp wurde vom Hersteller entwickelt und ist in der Lage verschiedene Standarddatentypen darzustellen. Des Weiteren kann eine Variable vom Typ `ALValue` zu einer Liste von Einträgen erweitert werden. Jeder Eintrag dieser Liste ist wiederum vom Typ `ALValue` und kann ebenfalls erweitert werden. Somit kann ein verschachteltes Konstrukt erschaffen werden. Genau ein solches Konstrukt ist für einen *DCM*-Befehl erforderlich und hat einen Nachteil, der im Folgenden genauer erläutert werden soll.
Zum besseren Verständnis wird hier als Beispiel ein Befehl gewählt, der beide Kopfmotoren bewegen soll. Beide Motoren sollen nach einer Sekunde die Position 1 erreichen und nach zwei Sekunden die Position 0. Alle alten anliegenden Befehle sollen gelöscht werden. Die Struktur eines Befehls vom Typ `ALValue` ist in Abbildung 4.5 dargestellt. Es ist zu erkennen, dass der Befehl aus einer Liste mit sechs Einträgen besteht, wovon die ersten vier Einträge einzelne Felder sind. Die Optionen Zeitmodus und Wichtigkeit wurden bisher nicht erläutert. Der Zeitmodus gibt an, ob es eine gemeinsame oder getrennte Zeitliste für die Aktorbefehle gibt. In diesem Beispiel wird eine gemeinsame Zeitliste genutzt. Andernfalls müsste die Option auf *time-mixed* gesetzt werden. Die Befehlsstruktur sähe dann ebenfalls anders aus. Die Option Wichtigkeit gibt an wie streng das Erreichen dieses Befehls verfolgt werden soll. Zur Zeit befindet sich diese Option noch in der Entwicklung und hat keine

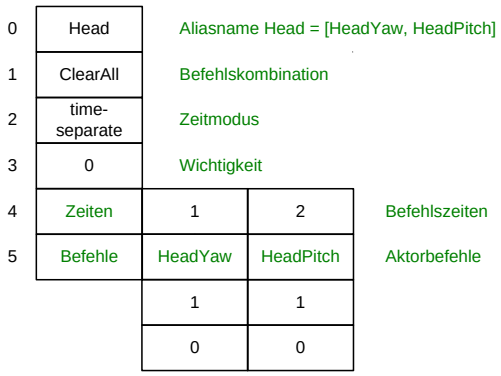


Abb. 4.5.: *DCM*-Befehl

Auswirkungen. Der Eintrag mit den Zeiten besteht wiederum aus eine Liste. Der Eintrag Befehle enthält eine Liste mit den Aktoren, die wiederum aus Listen mit den Befehlen bestehen.
Die Möglichkeit, verschiedene Datentypen in einem neuen Datentyp darstellen zu können, der die Möglichkeit besitzt, dynamisch erweitert zu werden, hat allerdings ihren Preis. Der Befehl in diesem Beispiel belegt mindestens 616 Byte Speicher. Der tatsächliche Wert kann größer sein, da häufig mehr Speicher reserviert wird, als tatsächlich benötigt würde. Ein leerer Container vom Typ `ALValue` benötigt 44 Byte Speicher. Wird dieser dazu genutzt eine ganze Zahl vom Typ `Integer` (4 Byte) zu speichern, wie das für die Zeiten der Fall ist, wird also 11-mal so viel Speicher benötigt. Hinzu kommt, dass die dynamische Erweiterung des `ALValue` Datentyps sehr rechenaufwendig ist.

Aus diesen Gründen wurde für die Simulation ein eigener Datentyp `Command` entwickelt, der feste Felder mit den entsprechenden Datentypen bereitstellt. Die Zeiten und Befehle werden in Vektoren bzw. verschachtelten Vektoren gespeichert. Somit entsteht die gleiche Struktur wie in Abbildung 4.5. Der Befehl im obigen Beispiel belegt als `Command` allerdings nur noch 200 Byte Speicher.

memory-Modul

Neben dem *DCM* steht auch das *memory*-Modul nicht in vollem Umfang in der Simulation zur Verfügung. Normalerweise schreibt der *DCM* neben den Sensorwerten auch alle Aktorbefehle in den Speicher. Da diese nur über den *DCM* aktualisierbar sind, ist eine Nutzung dieses Moduls in der Simulation nicht möglich. Die Entwicklung eines eigenen Speichermoduls liegt daher nahe. Dieses Modul wird als *Blackboard* bezeichnet und beinhaltet Speicherplätze für alle Variablen, die von den Modulen benötigt werden.
Dieses Modul ist allerdings nicht nur für die Simulation von Nutzen. Auch auf dem Roboter ist es sinnvoll, einmal pro Zyklus alle Variablen global zu speichern. Wie bereits erwähnt, können Zugriffe auf Daten im *memory*-Modul viel Zeit in Anspruch nehmen. Es besteht also hiermit die Möglichkeit, die Adresspointer global zu speichern und den Modulen zur Verfügung zu stellen oder eben die Inhalte der Variablen

zu speichern und diese den Modulen bereitzustellen. Beide Möglichkeiten führen zu dem gleichen Ergebnis, nur dass die zweite Möglichkeit sowohl für die Simulation als auch auf dem Roboter umsetzbar ist.

Abbildung 4.6 zeigt die Einbindung des *Blackboards* in den Prozess. Der Zugriff auf

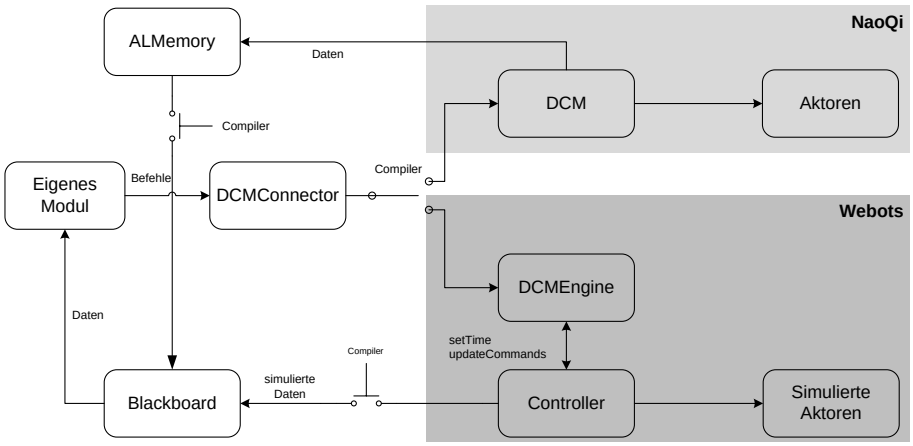


Abb. 4.6.: Einbindung des *Blackboard*-Moduls in Simulation und *NaoQi*

Daten durch ein eigenes Modul erfolgt dann durch Zugriff auf die Daten im Blackboard. Somit muss im eigenen Modul keine Unterscheidung getroffen werden, ob die Daten vom Simulator oder vom Roboter abgefragt werden müssen. Diese Unterscheidung übernimmt der Compiler.

Durch die entwickelten Module zur Simulation verschiedener Komponenten ist es nun möglich, eigene Module ohne Veränderungen im Quellcode sowohl im Simulator als auch auf dem Roboter zu verwenden.

5. Regelstrategien humanoider Laufbewegungen

In diesem Kapitel werden gängige Ansätze für Regelstrategien humanoider Laufbewegungen für Roboter vorgestellt. Es gibt verschiedene Ansätze für verschiedene Arten humanoider Roboter. So gibt es Roboter, die einen passiven Gang ausführen. Diese Roboter besitzen keine Aktoren, und eine Bewegung ist nur auf einer schiefen Ebene möglich. Des Weiteren existieren volldynamische Roboter, die ständig in Bewegung bleiben müssen, um nicht umzufallen. Auf die Regelungen solcher Roboter wird im Folgenden nicht eingegangen. Die hier vorgestellten Ansätze beziehen sich auf Roboter, die durch Aktoren getrieben werden und die zudem in ein statisches Gleichgewicht gebracht werden können. Bevor die Regelstrategien vorgestellt werden, wird im nächsten Abschnitt zunächst der Unterschied zwischen statischer und dynamische Balance erläutert.

5.1. Balancierter Gang

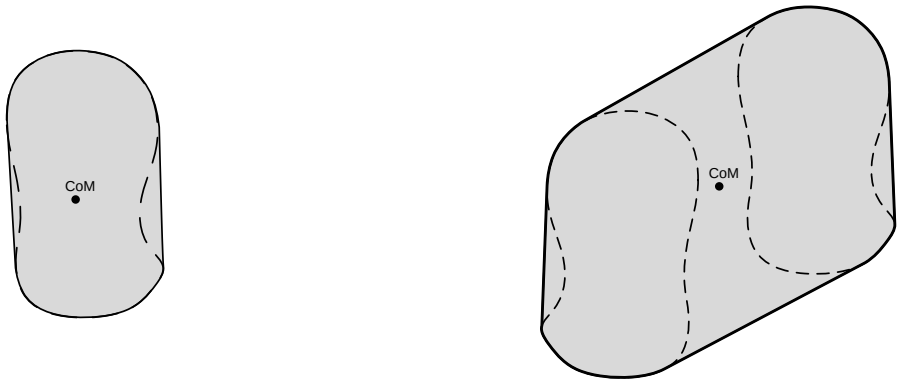
Für die Entwicklung eines Laufalgorithmus ist es zunächst notwendig, die Bedingungen zu erfassen, welche den Roboter im Gleichgewicht halten. Hierbei werden die Fälle der statischen Balance und die der dynamischen Balance[1] unterschieden. Des Weiteren wird der Gang in verschiedene Phasen unterteilt, und zwar in Einzel- und Doppelsupportphasen. Während der Einzelsupportphase hat der Roboter nur durch einen Fuß Bodenkontakt. Diese Phase dient dazu, den jeweils anderen Fuß zu versetzen. Während der Doppelsupportphasen verteilt sich das Gewicht des Roboters auf beide Füße. Diese Phasen werden genutzt, um den Schwerpunkt des Roboters zu verlagern.

Statische Balance

Für eine statische Balance muss die Bodenprojektion des Schwerpunktes innerhalb der konvexen Hülle liegen, die durch die Supportfüße aufgespannt werden. Als Supportfuß wird im Folgenden immer der Fuß bezeichnet, der den Kontakt zum Boden hält. Hierbei können auch beide Füße Supportfüße sein. Abbildung 5.1 zeigt den jeweils zulässigen Bereich der Schwerpunktsprojektion für die unterschiedlichen Fälle. Die Bedingung für eine statische Balance muss immer dann erfüllt sein, wenn der Roboter sich nicht bewegt und die Summe der auf den Roboter wirkenden Kräfte

[1] Häufig auch als statische bzw. dynamische Stabilität bezeichnet.

Null ist. Als Beispiel sei hier das Balancieren auf einem Bein genannt. Sofern keine externe Kraft auf den Roboter wirkt, die nicht kompensiert werden kann[2], wird der Roboter auf einem Bein stehen bleiben, wenn sich sein Schwerpunkt über dem Supportfuß befindet.



(a) Einzel-Support (b) Doppel-Support

Abb. 5.1.: konvexe Hüllen der Supportflächen für Einzel- und Doppelsupport (CoM = Center of Mass)

Im Allgemeinen ist der menschliche Gang allerdings nicht statisch balanciert, was bedeutet, dass sich der Schwerpunkt nicht zu jeder Zeit über der Supportfläche befindet. Stattdessen sind die Bedingungen für dynamische Balance erfüllt.

Dynamische Balance

Ein dynamisch ausbalanciertes System ist gegeben, wenn sich der Zero Moment Point[3], im Folgenden mit ZMP abgekürzt, zu jeder Zeit innerhalb der Supportfläche befindet. Der ZMP wurde erstmals 1968 von Vukobratović [VB04] eingeführt und beschreibt den Punkt am Boden, an dem eine Kraft wirken muss, die die horizontal wirkenden Komponenten aller aktiven Momente kompensiert. In Abbildung 5.2 ist diese Kraft F eingezeichnet.

[2] Kräfte, welche in Richtung der Gewichtskraft wirken, können durch den Support-Fuß kompensiert werden. Des Weiteren sind Kompensationen durch Reibung möglich.
[3] deutsch: Null-Moment-Punkt

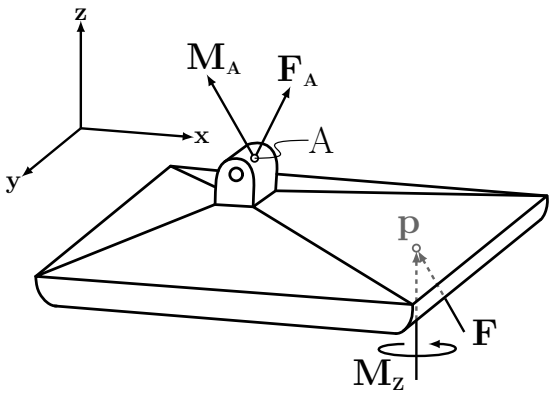


Abb. 5.2.: ZMP im Punkt P mit eingezeichneten Momenten und Kräften [VB04]

Die Kraft F wirkt im Punkt p und kann die Momente kompensieren, die um die x- und die y-Achse wirken. Momente, die um die z-Achse wirken, können hierdurch nicht kompensiert werden. Diese veranlassen den Roboter, sich um die Hochachse zu drehen, wodurch die Balance allerdings nicht gefährdet wird. Die Summe aller Momente des Körpers, sowie die Kraft des restlichen Körpers, wirken im Punkt A. Liegt der ZMP innerhalb der Supportfläche, so können alle Momente, die den Roboter aus der Balance bringen würden, kompensiert werden. Liegt der ZMP außerhalb, so wirkt ein Moment, welches den Roboter zu Fall bringen kann.
Tatsächlich ist der ZMP nur innerhalb der Supportfläche definiert. Es ist allerdings auch möglich, einen Punkt zu berechnen, an dem die Gegenkraft wirken müsste, wenn dieser nicht innerhalb der Supportfläche liegt. Dieser Punkt wird dann als FZMP (Fictitous ZMP) oder FRI (Foot Rotation Indicator) bezeichnet.
Eine Besonderheit des ZMP ist also, dass er nur im Falle eines dynamisch ausbalancierten Systems existiert. Ist dies der Fall, so ist er mit dem CoP[4] (Center of Pressure) koinzident. Der Druck zwischen Fuß und Boden kann immer durch eine einzelne Krafteinwirkung im CoP ersetzt werden. Abbildung 5.3 verdeutlicht das Verhältnis zwischen ZMP, FZMP und CoP.

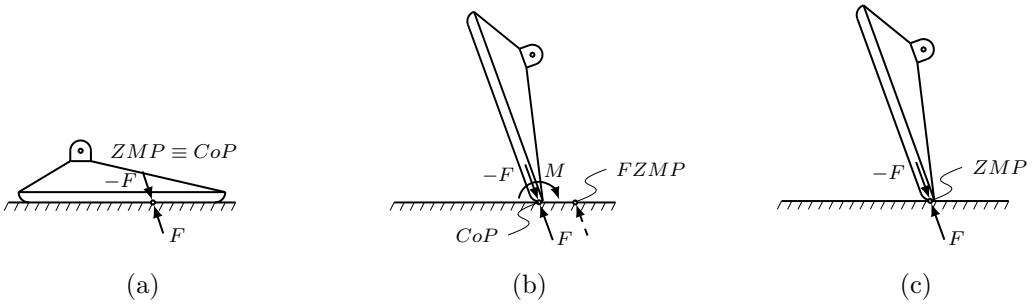


Abb. 5.3.: Beziehung zwischen ZMP, FZMP und CoP [VB04]

[4] deutsch: Druckzentrum

5.2. Statisch stabiles Laufen

Im vorigen Abschnitt wurde die Definition für statische Balance erklärt. Tatsächlich waren die ersten implementierten Laufalgorithmen für humanoide Roboter statischer Natur. Bei der Implementierung solcher Algorithmen wird von kleinen Geschwindigkeiten und Beschleunigungen für die Bewegungen des Roboters ausgegangen. Des Weiteren wird davon ausgegangen, dass die Kontaktpunkte des Roboters zum Boden während eines Schrittes keiner Bewegung unterliegen. Dies bedeutet, dass für einen

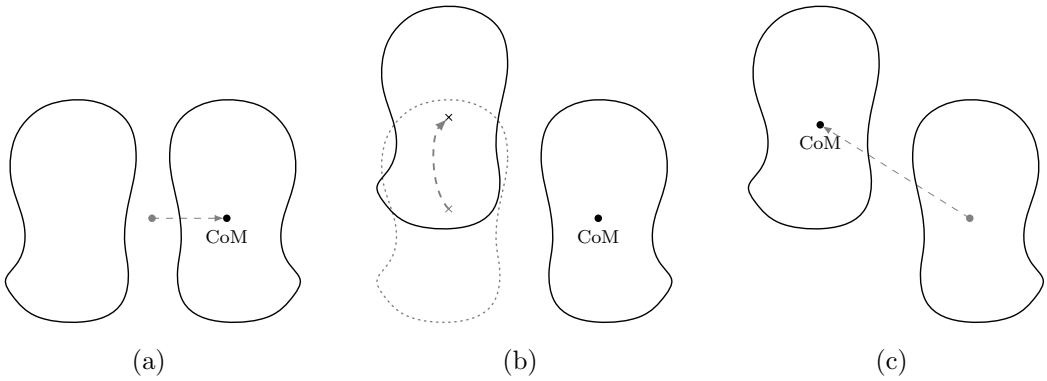


Abb. 5.4.: Schwerpunktbewegung für statisches stabiles Gleichgewicht. **(a)** Verschiebung des Schwerpunkts auf einen Fuß, **(b)** Versetzen des anderen Fußes, **(c)** Verlagerung des Schwerpunktes auf gerade versetzten Fuß

Schritt, die Schwerpunktsprojektion zunächst innerhalb der konvexen Hülle eines Fußes gebracht werden muss. Dieser Fuß ist dann der Supportfuß für den nächsten Schritt. Während eines Schrittes wird das zum Supportfuß gehörige Bein allerdings nicht bewegt. Es wird lediglich der andere Fuß versetzt. In der darauf folgenden Doppelsupportphase wird die Schwerpunktsprojektion anschließend in die Supportfläche des versetzten Fußes gebracht. Abbildung 5.4 verdeutlicht diesen Vorgang. Eine Implementierung und Regelung von statisch ausbalancierten Laufalgorithmen ist aufgrund der Reduzierung auf kleine Beschleunigungen sowie der guten Erfassbarkeit des Schwerpunktes meist relativ schnell und einfach umsetzbar, allerdings können keine hohen Geschwindigkeiten beim Gehen erreicht werden.

5.3. Dynamisch stabiles Laufen

Abgesehen von den bereits erwähnten passiven- sowie volldynamischen Robotern, liegt die Regelstrategie für dynamische Laufstabilität darin, den ZMP innerhalb der Supportfläche zu halten. Diese Strategie ist auch bei vielen bekannten humanoiden Robotern wie z.B. dem *Asimo* von *Honda* implementiert.
Lediglich die Modellansätze, die zur Regelung verwendet werden, unterscheiden sich. Einige Strategien basieren auf dem *Cart-Table-Modell*, andere auf dem *3DLIPM* und wieder andere nutzen ein vollständiges dynamisches Modell des Roboters. Für alle Modelle sind Beispiele für erfolgreiche Implementierungen vorhanden. Im Folgenden werden die einzelnen Modelle genauer erläutert.

5.3.1. Cart-Table-Modell

Das *Cart-Table-Modell* besteht aus einem masselosen Tisch, auf dem sich ein Fahrzeug der Masse M bewegen kann. Eine Skizze des Modells ist in Abbildung 5.5 dargestellt.

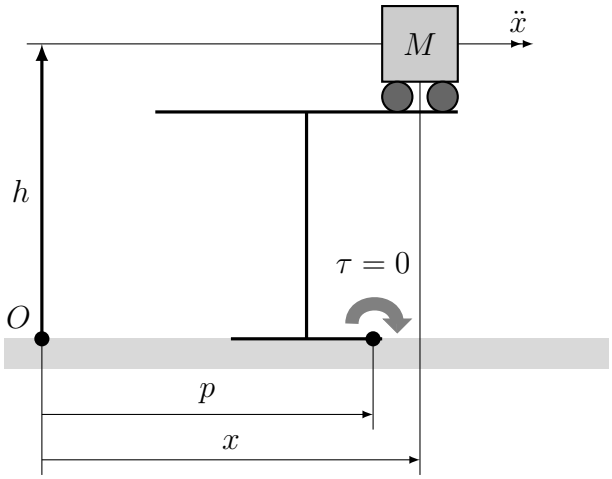


Abb. 5.5.: Cart-Table-Modell [SK08]

Für dieses Modell kann die Position des ZMP $p = [p_x\ p_y]^T$ durch folgende Gleichung beschrieben werden (Herleitung [SK08]):

$$p = x - \frac{h}{g}\ddot{x} \tag{5.1}$$

Hierbei ist g die Erdbeschleunigung und $x = [x_x\ x_y]^T$ und h geben die Position des Schwerpunktes der Anordnung an. Die Gleichung ist allerdings nur gültig, sofern der Fuß des Tisches mit seiner gesamten Fläche auf dem Boden aufliegt. Für die Entwicklung eines Laufalgorithmus auf Basis dieses Modells muss davon ausgegangen werden, dass der Supportfuß immer vollständigen Kontakt zum Boden hat, d.h. er wird also als mit dem Boden verklebt betrachtet.

Für einen stabilen Gang ist die Bedingung zu erfüllen, den ZMP innerhalb der Supportfläche zu halten. Aus diesem Grund wird eine Referenztrajektorie für den ZMP vorgegeben. Anschließend muss der Roboter so bewegt werden, dass der ZMP dieser Trajektorie folgt. Die Trajektorie wird zunächst allerdings in ihre x- und y-Komponenten zerlegt. Das Koordinatensystem des Roboters ist in Abbildung 5.6 eingezeichnet. Ein Beispiel für die Trajektorien für eine Bewegung in x-Richtung ist in Abbildung 5.7 dargestellt. Der Verlauf der Trajektorie in x-Richtung entsteht durch die Bedingung, dass der ZMP während eine Einzelsupportphase seine Position nicht ändern soll. Er verharrt so lange an der gleichen Position, bis ein Supportwechsel ansteht. In diesem Fall wird der ZMP in die Supportfläche des anderen Fußes geführt. Während des Wechsels des ZMP vom einen auf den anderen Fuß muss er dennoch innerhalb der Supportfläche bleiben. Der Wechsel findet also in einer Doppelsupportphase statt. Diese kann, wie in der Abbildung ersichtlich, sehr kurz

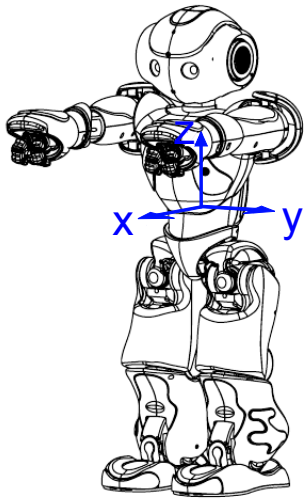


Abb. 5.6.: Koordinatensystem des Roboters [Ald11]

angesetzt werden.
Die Bewegung in y-Richtung ergibt sich aus der Verlagerung des ZMP von der Supportfläche des einen Fußes in die Supportfläche des anderen Fußes. Für den *Nao* ist es sinnvoll, den ZMP zwischen ±50mm alternieren zu lassen. Dies ist genau der Abstand vom Torso zur Hüfte (siehe Abb. 2.4). Sofern keine Rotation des Beines um die x-Achse vorhanden ist, sind ebenfalls die Füße in diesem Abstand positioniert. Es wird also eine Referenztrajektorie gewählt, die auf die körperlichen Abmessungen des Roboters abgestimmt ist.

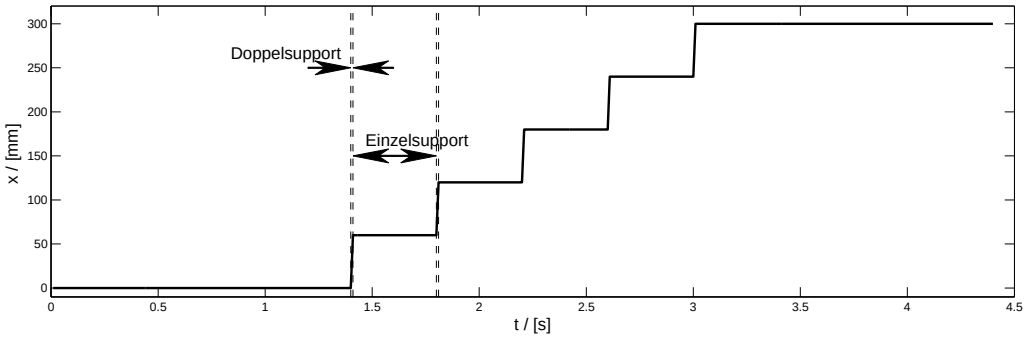
Sind die Trajektorien für den ZMP definiert, muss nachfolgend eine Trajektorie für den Schwerpunkt berechnet werden, aus welcher die ZMP-Trajektorie hervorgeht. Diese Berechnung kann auf verschiedene Arten erfolgen. Die am häufigsten referenzierte und implementierte Lösung des Problems besteht allerdings in der Nutzung eines prädiktiven Reglers [KKK+03]. Dieser Lösungsansatz wird im Folgenden erläutert.

Zunächst wird die Gleichung (5.1) in ein Zustandsraummodell überführt. Als Stellgröße des Systems wird der Ruck $\dddot{x}$ des Schwerpunkts genutzt.

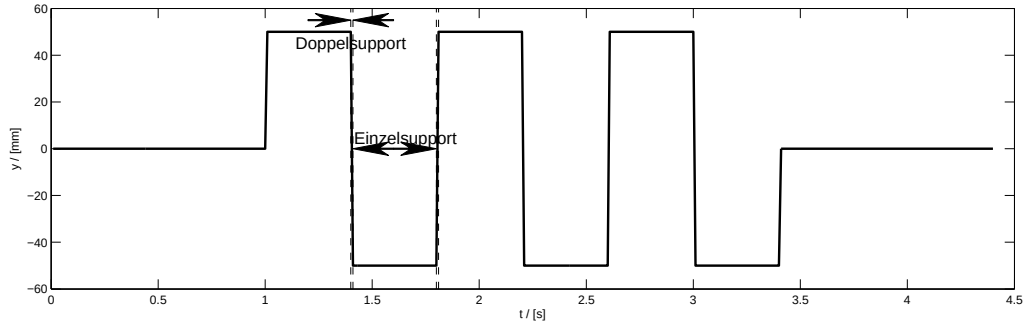
$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u$$

$$p = \begin{bmatrix} 1 & 0 & -z/g \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \end{bmatrix} \tag{5.2}$$

Der optimale Wert für u wird nach dem von Katayama et al. in [KOIK85] vorgestellten Verfahren berechnet. Zunächst wird das kontinuierliche Zustandsraummodell in



(a)



(b)

Abb. 5.7.: Referenztrajektorien für ZMP-Bewegung. (a) x-Richtung, (b) y-Richtung. Schrittweite 60mm, Schrittfrequenz: 2,5Hz

Gleichung (5.2) diskretisiert. Hierbei gibt T die Abtastzeit an.

$$x(k+1) = Ax(k) + Bu(k)$$

$$p(k) = Cx(k)$$

mit

$$
\begin{aligned}
x(k) &= \begin{bmatrix} x(kT) & \dot{x}(kT) & \ddot{x}(kT) \end{bmatrix}^T \\
u(k) &= u(kT) \\
p(k) &= p(kT) \\
A &= \begin{bmatrix} 1 & T & T^2/2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \\
B &= \begin{bmatrix} T^3/6 \\ T^2/2 \\ T \end{bmatrix} \\
C &= \begin{bmatrix} 1 & 0 & -z/g \end{bmatrix}
\end{aligned} \tag{5.3}
$$

Hierbei bedeutet $f(k) = f(kT)$. Als nächster Schritt wird der *Performance Index* J definiert.

$$
J = \sum_{i=k}^{\infty} \left\{ Q_e e(i)^2 + \Delta x^T(i) Q_x \Delta x(i) + R \Delta u^2(i) \right\} \tag{5.4}
$$

Dieser muss anschließend minimiert werden, um die optimale Lösung für u zu finden. In den *Performance Index* fließen verschiedene Faktoren ein. Zunächst einmal soll der Fehler $e(k) = p(k) - p^{ref}(k)$ zwischen dem Systemausgang und der Referenztrajektorie minimiert werden. Desweiteren soll die Stellgröße zwischen zwei Abtastschritten nicht zu stark verändert werden. Das Gleiche gilt für den Zustandsvektor. Hierzu werden die Änderungen $\Delta u(k) = u(k) - u(k-1)$ und $\Delta x(k) = x(k) - x(k-1)$ definiert. Für die Gewichte gilt hierbei $Q_e, R > 0$ und Q_x ist eine nicht negativ definite 3×3 Matrix. Über diese Faktoren lässt sich einstellen, auf welche Größe bei der Minimierung das höchste Gewicht fällt. Ein großes R sorgt z.B. dafür, dass eine Lösung gefunden wird, bei welcher Δu klein bleibt.

Zur Berechnung der optimalen Stellgröße muss zunächst noch die Anzahl der prädiktiven Schritte N definiert werden. Diese gibt an, wie viele zukünftige Schritte von $p^{ref}(k)$ zur Berechnung verwendet werden. Die optimale Stellgröße ergibt sich zu:

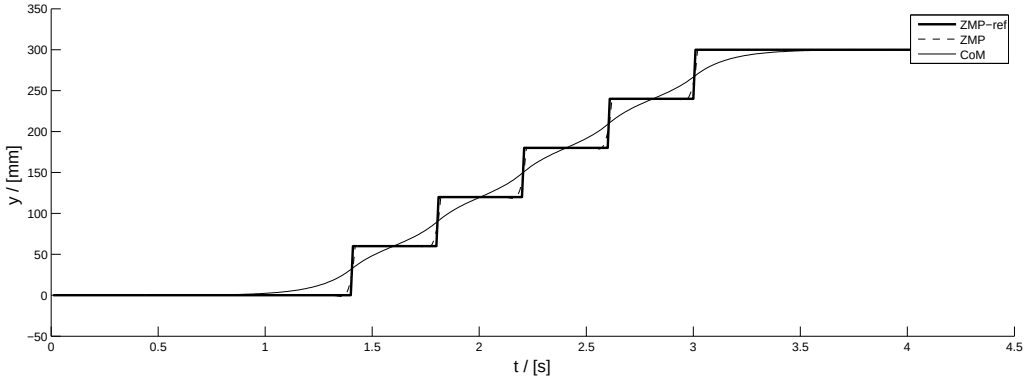
$$
u(k) = -G_i \sum_{i=0}^{k} e(k) - G_x x(k) - \sum_{j=1}^{N} G_p(j) p^{ref}(k+j) \tag{5.5}
$$

Die Gewichte G_i, G_x und $G_p(j)$ können durch Erweiterung des durch Gleichung (5.3) gegebenen Systems, den Parametern Q_e, Qx, R und anschließendes Lösen einer Algebraischen Riccatigleichung berechnet werden. Für weitere Informationen und den Beweis, dass u wirklich optimal ist, sei hier auf [KOIK85] verwiesen.

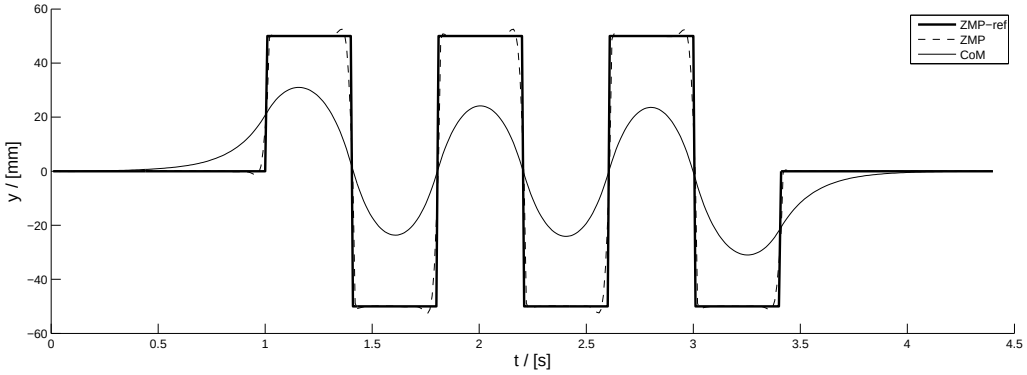
Der Verlauf des Schwerpunktes ergibt sich nun aus dem mit optimalem u getriebenen Zustandsraummodell. Der Zustandsvektor des Modelles beinhaltet als erstes Element die Position des Schwerpunktes, der durch

$$x(k) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x(k) \\ \dot{x}(k) \\ \ddot{x}(k) \end{bmatrix} \tag{5.6}$$

extrahiert werden kann. Die Komponenten für die x- und y-Position des Schwerpunk-



(a)



(b)

Abb. 5.8.: Ermittlung des Schwerpunktverlaufes aus ZMP-Trajektorie. (a) x-Richtung, (b) y-Richtung; $T = 10ms, z = 0.24m, N = 100, R = 1 \cdot 10^{-4}, Qx = 0.01 \cdot I_3, Qe = 0.25$

tes sind die Elemente x_x und x_y der Größe x. Dass hier x sowohl zur Bezeichnung für die Position der Schwerpunktsprojektion als auch als Bezeichnung für die x-Komponente im Koordinatensystem gewählt wird, ist der Tatsache geschuldet, dass

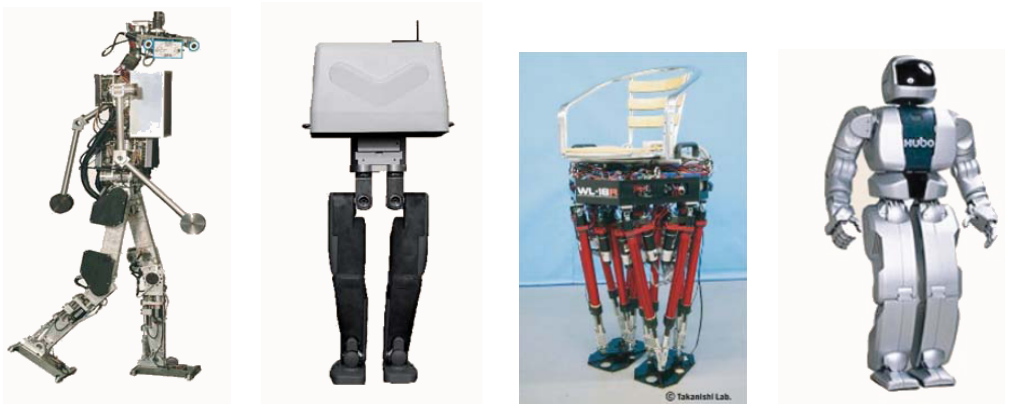
dies in der referenzierten Literatur ebenfalls so gehandhabt wird.

Der optimale Schwerpunktverlauf für die zuvor definierten ZMP-Referenztrajektorien ist in Abbildung 5.8 dargestellt. Neben der Referenztrajektorie und dem Schwerpunktsverlauf ist auch der ZMP-Verlauf eingezeichnet, der als Systemausgang des Modelles definiert ist. Es ist zu erkennen, dass Referenz und Modellvorhersage sehr gut übereinstimmen.

Um einen Gang des Roboters zu erzeugen, müssen die Füße des Roboters relativ zum Schwerpunkt gesetzt werden. Die Position der Füße ist durch die ZMP-Trajektorie gegeben und der Schwerpunktsverlauf durch den prädiktiven Regler. Die Positionierung von Gliedern relativ zu einem Punkt im Koordinatensystem erfordert die Ermittlung der Gelenkwinkel, die zum Erreichen dieser Position führen. Diese Berechnung wird als inverse Kinematik bezeichnet und wird in Abschnitt 6.2 vorgestellt.

Regelung

Mit der zuvor vorgestellten Vorgehensweise lassen sich bereits langsame Laufalgorithmen für humanoide Roboter implementieren. Allerdings wird auf Störungen nicht reagiert. Zudem ist ein stark vereinfachtes Modell als Grundlage für die Berechnungen gewählt worden. Um einen robusten und schnellen Gang zu erreichen, bedarf es einer Regelung. Für eine Regelung ist es allerdings notwendig, die Regelgröße, in diesem Fall den ZMP, messen zu können oder zumindest eine gute Approximation der Größe zur Verfügung zu haben. Für viele erfolgreiche Regelungen wird



(a) (b) (c) (d)

Abb. 5.9.: Beispiele für Roboter, die die Messung des ZMP durch Kraftsensoren in den Füßen durchführen. (a) Johnnie, (b) HRP-2L, (c) WL-16R, (d) HUBO [SK08]

die Messung des ZMP durch Kraftsensoren in den Füßen erreicht. Hierbei wird die Übereinstimmung vom ZMP mit dem CoP innerhalb der Supportfläche ausgenutzt. Beispiele für Roboter, die auf diese Weise eine Messung des ZMP vornehmen, sind

unter anderem der von Gienger et al. der Technischen Universität München entwickelte Roboter Johnnie (Abb. 5.9(a)) [GLP01], der von Kaneko et al. entwickelte HRP-2L (Abb. 5.9(b)) [KKK+02b], der von Takanishi et al. entwickelte WL-16R (Abb. 5.9(c)) [SKM+04] sowie der Roboter HUBO (Abb. 5.9(d)), der von Oh et al. entwickelt wurde [PKLO06].

Eine andere Möglichkeit, eine Abschätzung des ZMP zu erhalten, ist die Messung des Roboterschwerpunktes und dessen Beschleunigung. Der ZMP wird dann nach Gleichung 5.1 berechnet. Ein solche Messung wird in [CKU10] für den *Nao* verwendet.

Die verwendeten Regelstrategien basieren zumeist auf Zustandsreglern, bei denen die Zustände über Beobachter approximiert werden. Der Vorteil von Zustandsreglern ist, dass der Regler für bestimmte Vorgaben durch bekannte Verfahren (z.B. LQR [Lun10]) optimal eingestellt werden kann.

5.3.2. 3DLIPM

In diesem Abschnitt wird das *3DLIPM* [5] [KKK+02a] erläutert. Abbildung 5.10 zeigt das Modell. Es ist durch ein inverses Pendel gegeben, dessen Pendelarm als masselos angenommen wird. Die Gesamtmasse des Pendels befindet sich im Kopf des Pendels. Für das Pendel gilt die vereinfachte Annahme, dass sich der Schwerpunkt nur in einer Ebene bewegen kann. Im Folgenden wird eine Ebene parallel zum Boden angenommen, sodass die Höhe h des Schwerpunkts als konstant betrachtet werden kann.

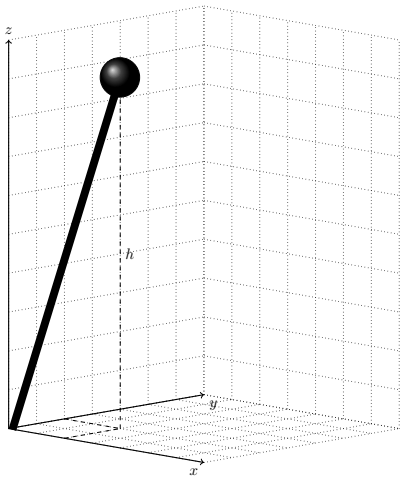


Abb. 5.10.: 3DLIPM

Die Bewegung des Schwerpunktes kann durch folgende zwei Gleichungen beschrieben

[5] Three Dimensional Linear Inverted Pendulum Mode

werden.

$$x(t) = x_0 \cdot \cosh(k \cdot t) + \dot{x}_0 \cdot \frac{1}{k} \cdot \sinh(k \cdot t) \tag{5.7}$$

$$\dot{x}(t) = x_0 \cdot k \cdot \sinh(k \cdot t) + \dot{x}_0 \cdot \cosh(k \cdot t) \tag{5.8}$$

Hierbei ist $k = \sqrt{\frac{g}{h}}$, x_0 ist die Position des Schwerpunkts relativ zum Ursprung des Pendels zur Zeit $t = 0$ und $\dot{x}_0$ die Geschwindigkeit des Schwerpunkts für $t = 0$.

Die Gleichungen, die das zuvor vorgestellte *Cart-Table-Modell* beschreiben, gehen auf ein *3DLIPM* zurück [KKK+03]. Die Strategie für einen Laufalgorithmus unter Nutzung des *3DLIPM* ist jedoch eine andere. Beim *Cart-Table-Modell* wurde eine Referenztrajektorie für den ZMP und somit für die Fußpositionen vorgegeben. Im Gegensatz hierzu werden nun die Gleichungen 5.7 und 5.8 genutzt, um einen Verlauf des Roboterschwerpunktes vorherzusagen und passende Fußpositionen zu berechnen. Der grundlegende Unterschied ist also, dass nicht versucht wird, den Roboter so zu bewegen, dass das zuvor definierte Schrittmuster erreicht wird, sondern den Roboter als Pendel zu betrachten und der natürlichen Pendelbewegung zu folgen, indem ein hierzu passendes Schrittmuster generiert wird.

Implementierungen dieses Konzeptes sind unter anderem in [KKK+02a] für den in Abbildung 5.9(b) gezeigten HRP-2L sowie für den *Nao* umgesetzt worden. Die Umsetzung für den HRP-2L sieht dennoch eine Referenztrajektorie für den ZMP vor und stabilisiert den Gang durch Messung des ZMP. Für den *Nao* wird im Folgenden die in [GR10] vorgestellte Methode vorgestellt, die vollständig auf eine Messung des ZMP verzichtet. Stattdessen wird die Schwerpunktsposition des Roboters gemessen und in die Bodenebene projiziert. Warum dies für den *Nao* besser geeignet ist, wird an späterer Stelle noch aufgezeigt.

Gleichungen 5.7 und 5.8 definieren die Bewegung des Schwerpunktes in einer Einzelsupportphase. Der Ursprung des Pendels wird dabei so gewählt, dass er möglichst dicht am Zentrum des Supportfußes liegt. Bei einem Wechsel des Supportfußes wird ebenso der Ursprung des Pendels verändert. Ziel ist es nun, zu Beginn einer Einzelsupportphase einen Zustand zu erreichen, der zu einem stabilen Zustand der nächsten Einzelsupportphase führt. Die meisten Ansätze nutzen eine kurze Doppelsupportphase, um den Schwerpunkt zu beschleunigen oder zu bremsen und somit die Startbedingungen für eine Einzelsupportphase zu erreichen. In diesem Ansatz wird auf eine Doppelsupportphase verzichtet, da hierdurch größere Schrittweiten und somit ein schnellerer Gang möglich sind.
Da ohne eine Doppelsupportphase dennoch die Anfangsbedingungen für die Einzelsupportphase erreicht werden müssen, wird der Zeitpunkt des Supportwechsels zum Kontrollieren der Schwerpunktsgeschwindigkeit in y-Richtung gewählt.
Zunächst muss eine Definition für $t = 0$ getroffen werden. Als $t = 0$ wird der Zeitpunkt definiert, bei dem das Pendel seine Bewegungsrichtung entlang der y-Achse umkehrt. Zu diesem Zeitpunkt ist die y-Komponente der Geschwindigkeit $\dot{x}_{0,y} = 0$. Die Position $x_{0,y}$ ist ein frei wählbarer Parameter, der angibt, wie weit das Pendel

in Richtung Pendelursprung schwingt, bevor es die Bewegungsrichtung ändert. Da $\dot{x}_{0,y} = 0$ gilt, können die Bewegungsgleichung für die y-Komponente zu

$$x_y(t) = x_{0,y} \cdot \cosh(k \cdot t) \tag{5.9}$$

$$\dot{x}_y(t) = x_{0,y} \cdot k \cdot \sinh(k \cdot t) \tag{5.10}$$

vereinfacht werden. Abbildung 5.11 zeigt die Momentaufnahme einer Pendelbewegung in y-Richtung. Des Weiteren sind die Koordinatensystem Q und $\bar{Q}$ eingezeichnet. Diese Koordinatensysteme besitzen einen Abstand r_y bzw. $\bar{r}_y$ zum Pendelursprung. Alle gestrichenen Größen geben Parameter der nächsten Pendelphase an. Die Größe $\bar{s}_y$ definiert die Schrittweite des Roboters in y-Richtung. Ist sie Null, so gilt $Q = \bar{Q}$. Der Betrag von r_y entspricht bei diesem Ansatz dem Abstand vom Torso zur Hüfte des *Naos*.
Um nun den Zeitpunkt für einen Supportwechsel bestimmen zu können und somit

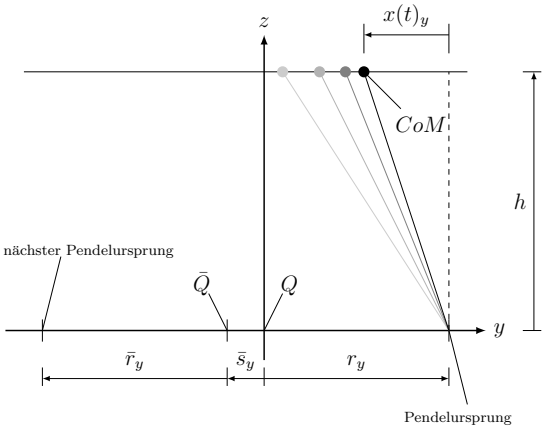


Abb. 5.11.: Momentaufnahme der Pendelbewegung in y-Richtung [GR10]

die Größe $\dot{x}_y$ zu kontrollieren, müssen Bedingungen aufgestellt werden, die bei einem Supportwechsel gelten müssen. Hierzu wird die Startzeit t_b und die Endzeit t_e einer Pendelphase eingeführt. Es gilt $t_b < 0$ sowie $t_e > 0$.
Bei einem Wechsel des Supportfußes bilden der Endzustand der aktiven Phase die Anfangsbedingung für die darauf folgende Phase. Es gilt demnach:

$$x(t_e)_y - \bar{x}(\bar{t}_b)_y = \bar{r}_y + \bar{s}_y - r_y \tag{5.11}$$

$$\dot{x}(t_e)_y = \dot{\bar{x}}(\bar{t}_b)_y \tag{5.12}$$

Abbildung 5.12 veranschaulicht die für den Wechsel geltenden Beziehungen. Aus diesen Gleichungen können nun die Größen $\bar{t}_b$ sowie t_e berechnet werden. In [GR10] wird ein iterativer Algorithmus zur Lösung des Problems vorgestellt.

Für die Bewegung in y-Richtung sind demnach alle Größen vorhanden, um Modellvorhersagen für die Schwerpunktsbewegung bezüglich der Gleichungen 5.11 und 5.12 treffen zu können. Das Grundprinzip für die Bewegung ist, den Wechsel des Fußes

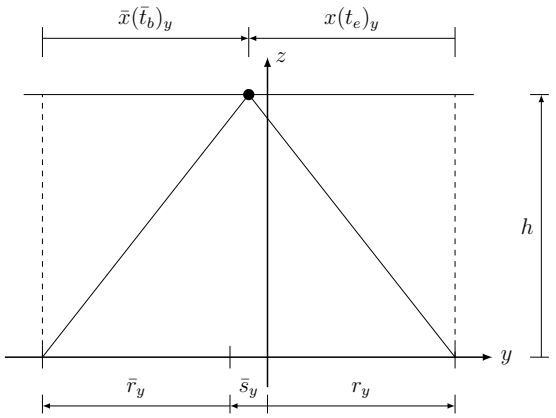


Abb. 5.12.: Wechsel des Supportfußes und der Pendelphase [GR10]

so zu gestalten, dass ein zuvor definierter Wert $\bar{x}_{0,y}$ in der nächsten Pendelphase erreicht wird.

Um eine Bewegung in x-Richtung zu erreichen, muss der Pendelursprung für die nächste Phase in einem Abstand $\bar{r}_x - \bar{s}_x - r_x$ vom aktuellen Pendelursprung platziert werden. Die benötigten Größen entsprechen den x-Komponenten der in Abbildung 5.13 eingezeichneten Größen. Des Weiteren gilt für die Schwerpunktgeschwindigkeit

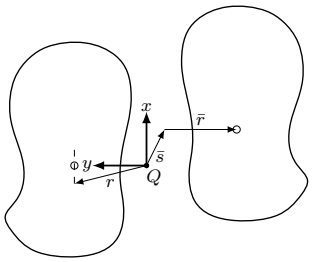


Abb. 5.13.: Fußpositionen in der x-y-Ebene. [GR11]

in x-Richtung die Bedingung, dass sie während eines Supportwechsels unverändert bleibt. Es ergeben sich analog zu den Bedingungen für die y-Bewegung die Bedingungen

$$x(t_e)_x - \bar{x}(\bar{t}_b)_x = \bar{r}_x + \bar{s}_x - r_x \tag{5.13}$$

$$\dot{x}(t_e)_x = \dot{\bar{x}}(\bar{t}_b)_x \tag{5.14}$$

für die x-Richtung. Um einen möglichst stabilen Gang zu erreichen, wird die Bedingung $\bar{r}_x = 0$ gesetzt. Diese Bedingung garantiert, dass ein Pendelmodell für die nächste Phase berechnet wird, das den Ursprung direkt im Zentrum des Fußes platziert. Des Weiteren wird gefordert, dass $\bar{x}_{0,x} = 0$ gilt. Diese Bedingung sorgt dafür, dass sich zum Zeitpunkt $\bar{t} = 0$ der Schwerpunkt des Pendels über dem Ursprung des Pendels befindet. Dies sorgt für eine möglichst gleichverteilte Pendelbewegung über der Zeit. Zum Beginn einer Pendelphase müssen nun noch die Größen $r_x, x_{0,x}$ und

$\dot{x}_{0,x}$ für eine vorgegebene Schrittweite $\bar{s}_x$ berechnet werden. Diese sind durch Lösen des folgenden Gleichungssystems ermittelbar.

$$\begin{array}{llll} r + x_0 \cdot \cosh(k \cdot t) & + \dot{x}_0 \cdot \frac{\sinh(k \cdot t)}{k} & & = x_t \\ x_0 \cdot k \cdot \sinh(k \cdot t) & + \dot{x}_0 \cdot \cosh(k \cdot t) & & = \dot{x}_t \\ x_0 \cdot k \cdot \sinh(k \cdot t_e) & + \dot{x}_0 \cdot \cosh(k \cdot t_e) & - \bar{x}_0 \cdot k \cdot \sinh(k \cdot \bar{t}_b) - \bar{\dot{x}}_0 \cdot \cosh(k \cdot \bar{t}_b) & = [0,0]^T \\ r + x_0 \cdot \cosh(k \cdot t_e) & + \dot{x}_0 \cdot \frac{\sinh(k \cdot t_e)}{k} & - \bar{x}_0 \cdot \cosh(k \cdot \bar{t}_b) - \bar{\dot{x}}_0 \cdot \frac{\sinh(k \cdot \bar{t}_b)}{k} - \bar{s} & = \bar{r} \end{array} \tag{5.15}$$

Die Größen x_t und $\dot{x}_t$ sind hierbei die aktuellen Werte für die Schwerpunktposition und die Schwerpunktsgeschwindigkeit. Diese müssen messtechnisch erfasst werden. Bei der Berechnung der Größe r_x kann es vorkommen, dass diese Werte annimmt, die größer als erlaubt sind. In diesem Fall würde der Pendelursprung außerhalb der Supportfläche liegen. Dies widerspricht der Bedingung für einen stabilen Gang. Daher kann diese Größe limitiert werden. Unter erneuter Lösung des Gleichungssystems 5.15 mit limitiertem r_x kann eine neue Schrittweite s_x berechnet werden, die den Bedingungen entspricht. Hierdurch wird gewährleistet, dass der Pendelursprung immer innerhalb der Supportfläche bleibt und nur Schrittweiten genutzt werden, die diese Bedingung nicht verletzen.

Warum ein Pendelursprung innerhalb der Supportfläche die Bedingung für einen stabilen Gang erfüllt, wird in [GR10] und [GR11] allerdings nicht geklärt. Es wird lediglich die Bedingung gestellt, dass der Ursprung des Pendels möglichst dicht am Zentrum des Fußes liegen muss.
Die Erklärung für diese Bedingung ergibt sich daraus, dass der ZMP genau im Pendelursprung liegt. Die klassische Definition des ZMP besagt, dass er nur innerhalb der Supportfläche existiert. Für ein Pendel reduziert sich die Supportfläche allerdings auf einen Punkt, den Pendelursprung.
Wird die erweiterte Definition des ZMP, die auch zulässt, dass er außerhalb der Supportfläche existieren kann genutzt, so kann der ZMP durch Gleichung 5.1 berechnet werden. Um zu zeigen, dass auch hier der ZMP im Pendelursprung liegt, wird zunächst Gleichung 5.7 nach der Zeit differenziert.

$$\ddot{x}(t) = x_0 \cdot k^2 \cdot \cosh(k \cdot t) + \dot{x}_0 \cdot k \cdot \sinh(k \cdot t) \tag{5.16}$$

Einsetzen der Gleichungen 5.7 und 5.16 in Gleichung 5.1 ergibt:

$$p = x_0 \cdot \cosh(k \cdot t) + \dot{x}_0 \cdot \frac{1}{k} \cdot \sinh(k \cdot t) - \frac{h}{g} \cdot \left(x_0 \cdot k^2 \cdot \cosh(k \cdot t) + \dot{x}_0 \cdot k \cdot \sinh(k \cdot t) \right)$$

$$= x_0 \cdot \cosh(k \cdot t) \cdot \left(1 - \frac{h}{g} \cdot k^2 \right) + \dot{x}_0 \cdot \sinh(k \cdot t) \left(\frac{1}{k} - \frac{h}{g} \cdot k \right)$$

mit $k = \sqrt{\frac{g}{h}}$ ergibt sich:

$$p = x_0 \cdot \cosh(k \cdot t) \cdot \left(1 - \frac{h}{g} \cdot \frac{g}{h} \right) + \dot{x}_0 \cdot \sinh(k \cdot t) \left(\sqrt{\frac{h}{g}} - \frac{h}{g} \cdot \sqrt{\frac{g}{h}} \right) = 0$$

Somit liegt der ZMP im Zentrum des Fußes und der Gang ist dynamisch stabil, wenn zu Beginn einer Pendelphase die Schrittweite und die Endzeit der Phase so berechnet werden, dass daraus ein Pendelursprung für die nächste Phase entsteht, der ebenso im Zentrum des Fußes liegt.

Bisher wurden Pendelparameter für die Einzelsupportphasen berechnet, allerdings noch nicht geklärt, auf welcher Basis der Roboter seine Bewegung durchführt. Dies soll im Folgenden geschehen.

Sind alle Parameter des Pendels berechnet, so ist eine Vorhersage der Position und Geschwindigkeit des Schwerpunkts für alle Zeiten gegeben. Ein Gang entsteht dadurch, dass der Schwerpunkt des Roboters so bewegt wird, dass er der Pendelvorhersage folgt. Dies bedeutet, dass das Supportbein den Pendelarm darstellt und dem vorhergesagten Pendelverlauf folgt. Das andere Bein kann in dieser Zeit um die berechnete Schrittweite versetzt werden. Die Positionierung der Füße relativ zum Schwerpunkt des Roboters erfolgt ebenso wie beim *Cart-Table-Modell* durch inverse Kinematik.

Regelung

Der vorgestellte Algorithmus reagiert auf Störungen, indem zu Beginn einer Phase die gemessene Schwerpunktsposition und Schwerpunktsgeschwindigkeit als Messwerte für die Berechnung der Pendelparameter herangezogen werden. Auf Störungen in der aktuellen Phase wird allerdings nicht reagiert. Sind die Pendelparameter für eine Phase berechnet, so erfolgt die Vorhersage auf Basis dieser Parameter. Um aber auch auf Störungen innerhalb einer Phase reagieren zu können, bedarf es einer Anpassung der Pendelparameter. Ein solcher Vorgang wird als Adaption bezeichnet. Eine Regelung auf Basis einer Adaption wird *Adaptive Regelung* genannt.
Die Anpassung der Parameter erfolgt durch die Messwerte der Schwerpunktsposition und Geschwindigkeit des Roboters. Hierzu werden die Gleichungen zu Vorhersage dieser Größen genutzt. Für die Bewegung in y-Richtung werden die Größen $x_{0,y}, t$ und t_e neu berechnet. Hierbei gibt die Zeit t den aktuellen Zeitpunkt des Pendels und nicht die tatsächliche Zeit an. Wird der Roboter z.B. festgehalten, so ist die Geschwindigkeit des Schwerpunktes Null. Zuvor wurde allerdings definiert, dass dies für $t = 0$ gilt. Das Pendelmodell wird in einem solchen Fall die Zeit entsprechend auf Null setzen. Dies ist auch sinnvoll, da ja kein Wechsel des Supportfußes vorgenommen werden muss, solange der Roboter festgehalten wird.
Für die x-Richtungen werden die Parameter x_0, $\dot{x}_0$ und r_x aktualisiert. Dies kann wiederum durch Lösen des Gleichungssystem 5.15 erfolgen.

Da eine Neuberechnung der Pendelparameter rein auf Basis der Messwerte nur unter optimalen, rauschfreien Messungen zu empfehlen ist, dies aber in der Realität nicht möglich ist, kann eine Filterung der Messwerte mit der Modellvorhersage vorgenommen werden, auf deren Basis dann die Modellparameter neu berechnet werden können.

5.3.3. Roboterspezifisches Modell

Neben den bereits vorgestellten vereinfachten Modellen, die den Roboter als Punktmasse in dessen Schwerpunkt darstellen, ist es auch möglich, ein auf den Roboter angepasstes Modell zu benutzen, das die Dynamik des Systems berücksichtigt. Hierzu wird nicht der Roboter als Ganzes als Punktmasse dargestellt, sondern die einzelnen Glieder werden durch ihren Schwerpunkt sowie deren Inertialmatrizen beschrieben. Anschließend werden die auf den Roboter wirkenden Momente unter Berücksichtigung dieser Größen sowie der Geschwindigkeit und Position jedes Gliedes berechnet. Aus den ermittelten Größen ist es anschließend möglich den ZMP des Modells zu berechnen. Das Grundprinzip dieser Berechnungsmethode ist in [SK08] beschrieben. Ein auf den Roboter angepasstes Modell zu nutzen, das die Dynamik des Systems berücksichtigt und somit das System besser nachbilden kann als das *Cart-Table-Modell* oder das *3DLIPM*, ist allerdings rechenaufwendig und erfordert Kenntnis über sehr viele Größen. Da diese Größen, wie z.B. Geschwindigkeit oder Position der Gliederschwerpunkte, Messschwankungen unterliegen, ist die Genauigkeit einer Modellvorhersage begrenzt.
Eine Veröffentlichung über die Implementierung eines solchen Modells für den *Nao* ist zudem nicht bekannt.

6. Kinematik

In diesem Abschnitt werden die kinematischen Grundlagen erklärt, die zur Entwicklung eines Laufalgorithmus notwendig sind. Die Kinematik[1] beschreibt die Bewegung von Punkten und Körpern im Raum. Sie bildet die Grundlage für die Bewegungssteuerung von Robotern. Humanoide Roboter bestehen meist aus einer Aneinanderreihung von Aktoren. In viele Einführungen zur Kinematik werden verschiedene Typen von Aktoren unterschieden, wie z.B. rotatorische Aktoren und lineare Aktoren. Da der *Nao* allerdings nur über rotatorische Aktoren verfügt, werden im Folgenden nur diese Aktoren betrachtet.

6.1. Vorwärtskinematik

Die Vorwärtskinematik dient dazu, die Position und Orientierung eines Robotergliedes in Abhängigkeit der Winkelstellung der Aktoren zu beschreiben. Es ist z.B. möglich, aus Kenntnis der Stellung aller Beingelenke die Position und Orientierung des Fußes im körpereigenen Koordinatensystem des *Naos* abzuleiten. Dieses befindet sich im Torso und dient als Bezugskoordinatensystem für die weiteren Ausführungen.

Für die Berechnung der gesuchten Größen muss zunächst eine geeignete Darstellung für Position und Orientierung gewählt werden. Die Position kann als einfacher Spaltenvektor der Form

$$p = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \tag{6.1}$$

beschrieben werden. Die Orientierung eines Koordinatensystems i relativ zu einem Koordinatensystems j kann durch Beschreibung der Basisvektoren $(e_x^i\ e_y^i\ e_z^i)$ durch die Basisvektoren von $(e_x^j\ e_y^j\ e_z^j)$ erfolgen. Hierbei wird die Notation ${}^jR_i$ für die Orientierung des Koordinatensystems j relativ zu i verwendet [SK08].

$${}^jR_i = \begin{pmatrix} e_x^i \cdot e_x^j & e_y^i \cdot e_x^j & e_z^i \cdot e_x^j \\ e_x^i \cdot e_y^j & e_y^i \cdot e_y^j & e_z^i \cdot e_y^j \\ e_x^i \cdot e_z^j & e_y^i \cdot e_z^j & e_z^i \cdot e_z^j \end{pmatrix} \tag{6.2}$$

Da die Basisvektoren Einheitsvektoren sind und das Skalarprodukt zweier Einheitsvektoren dem Kosinus des Winkels zwischen den Vektoren entspricht, werden die Einträge dieser Matrix auch als Richtungskosinus bezeichnet. Die Matrix R wird

[1] gr.: kinema, Bewegung

Rotationsmatrix genannt.
Für elementare Drehungen um eine Achse des Bezugskoordinatensystems gilt:

$$R_z(\alpha) = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{6.3}$$

$$R_y(\alpha) = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix} \tag{6.4}$$

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} \tag{6.5}$$

Hierbei gibt der Index x, y oder z die Achse und α den Winkel an, um den gedreht wird. Eine besondere Eigenschaft von Rotationsmatrizen ist, dass ihre Inverse gleich ihrer Transponierten ist.

$$R^{-1} = R^T \tag{6.6}$$

Um einen Punkt ${}^{i}r$ aus dem Koordinatensystem i in ein Koordinatensystem j zu transformieren, das um ${}^{j}R_i$ gedreht und um ${}^{j}p_i$ verschoben ist, ist folgende Berechnung notwendig:

$${}^{j}r = {}^{j}R_i \cdot {}^{i}r + {}^{j}p_i \tag{6.7}$$

Dies kann kompakt als

$$\begin{pmatrix} {}^{j}r \\ 1 \end{pmatrix} = {}^{j}T_i \cdot \begin{pmatrix} {}^{i}r \\ 1 \end{pmatrix}, \qquad {}^{j}T_i = \begin{pmatrix} {}^{j}R_i & {}^{j}p_i \\ 0 & 1 \end{pmatrix} \tag{6.8}$$

geschrieben werden. Die Transformationsmatrix T beinhaltet also beide Informationen, die Verschiebung und die Drehung. Für eine Transformationsmatrix dieser Art gilt nun nicht mehr die Beziehung aus Gleichung 6.6. Die Inverse kann dennoch vereinfacht berechnet werden. Es gilt:

$${}^{j}T_i^{-1} = {}^{i}T_j = \begin{pmatrix} {}^{j}R_i^T & -{}^{j}R_i^T {}^{j}p_i \\ 0 & 1 \end{pmatrix} \tag{6.9}$$

Transformationsmatrizen, die lediglich eine Drehung (Rot) oder eine Verschiebung ($Trans$) des Koordinatensystems darstellen haben folgende Form:

$$Rot_w(\alpha) = \begin{pmatrix} R_w(\alpha) & 0 \\ 0 & 1 \end{pmatrix}, \qquad Trans_w(d) = \begin{pmatrix} I & p_w(d) \\ 0 & 1 \end{pmatrix} \tag{6.10}$$

Hierbei gibt w die Achse an, um die gedreht bzw. verschoben wird. Die entsprechende Komponente w des Vektor p_w nimmt den Wert d an, die beiden anderen Komponenten sind Null. Für hintereinander ausgeführte Transformationen gilt:

$${}^{k}T_i = {}^{k}T_j {}^{j}T_i \tag{6.11}$$

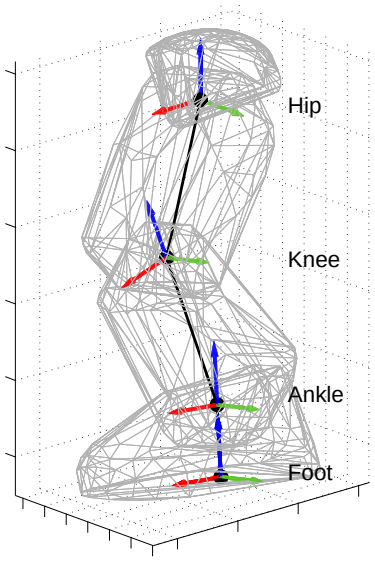


Abb. 6.1.: Bein mit eingezeichneten Koordinatensystemen der einzelnen Glieder. x-Achse: rot; y-Achse: grün; z-Achse: blau

Abbildung 6.1 zeigt die kinematische Kette des linken Beines eines *Nao*-Roboters. Um die Position des Fußes relativ zur Hüfte zu berechnen, sind einige Transformationen nötig. Hierzu müssen alle Gelenkwinkel sowie die Abstände der Gelenke zueinander bekannt sein. Zunächst muss der Fuß in das Koordinatensystem des Knöchels überführt werden. Da sich zwischen Fuß und Knöchel kein Aktor befindet, sodass diese nicht zueinander verdreht sein können, sind die Koordinatensysteme also lediglich gegeneinander verschoben. Die Position des Fußes im Knöchelkoordinatensystem entspricht ${}^{Ankle}p = {}^{Ankle}T_{Foot}\,{}^{Foot}p$. Nun kann diese Position wiederum in das Kniekoordinatensystem überführt werden. Das Kniekoordinatensystem ist gegenüber dem Knöchelkoordinatensystem verdreht und verschoben. Der Knöchel besteht aus 2 Aktoren, die um verschiedene Achsen drehen. Bei der Transformation ist nun auf die Reihenfolge der Rotationen zu achten. Zuerst muss die Rotation des Aktors erfolgen, der auf die Orientierung des anderen Aktors keinen Einfluss hat. Anschließend erfolgt die Verschiebung um die Länge des Schenkels. Beim *Nao* muss zunächst eine Drehung um die x-Achse des Knöchels erfolgen, anschließend noch eine Drehung um die y-Achse. Die Verschiebung in Richtung Knie geht entlang der z-Achse.

$$ {}^{Knee}p = Trans_z(ThighLength) \cdot (Trans_y(\alpha_{AnklePitch}) \cdot (Rot_x(\alpha_{AnkleRoll}) \cdot {}^{Ankle}p)) \tag{6.12} $$

Hierbei entspricht $ThighLength$ der Länge des Schienenbeins des Roboters. Der Winkel $\alpha_{AnklePitch}$ der Winkelstellung des Aktors *AnklePitch* sowie $\alpha_{AnkleRoll}$ der Winkelstellung des Aktors *AnkleRoll*. Die Bezeichnungen der Aktoren sind Kombinationen aus den Gelenknamen, in denen sie platziert sind, sowie der Achse, um die diese Drehen. Hierbei steht *Roll* für die Drehung um die x-Achse des Gelenkes, *Pitch* für die Drehung um die y-Achse und *Yaw* für die Drehung um die z-Achse. Diese Art der Bezeichnung war zunächst nur für Luftfahrzeuge gebräuchlich (Abb.

6.2), hat sich allerdings auch in der Robotik und anderen Bereichen verbreitet.

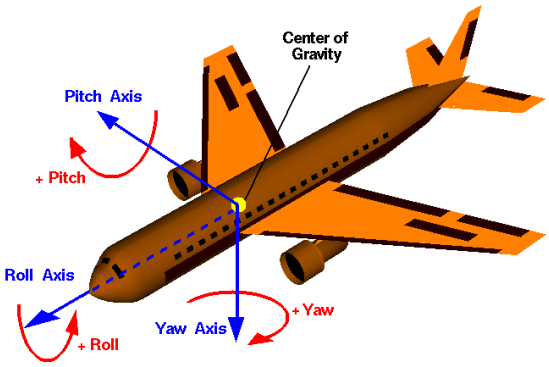


Abb. 6.2.: Drehachsenbezeichnungen bei einem Flugezug

Die weiteren Transformationen des berechneten Punktes in das Hüftkoordinatensystem erfolgen nach dem gleichen Prinzip.

Die Vorwärtskinematik ist nicht nur zur Berechnung der Positionen von einzelnen Gliedern von Nutzen. Sie ermöglicht auch eine schnelle und einfache Berechnung des Schwerpunktes einer kinematischen Kette. Ist der Schwerpunkt eines Gliedes relativ in seinem eigenen Koordinatensystem bekannt, so kann dieser Punkt in ein anderes Koordinatensystem überführt werden. Um den Schwerpunkt des gesamten Roboters zu berechnen, ist es also möglich, alle Schwerpunkte der einzelnen Glieder in das Torsokoordinatensystem zu transformieren. Bei Kenntnis der Lage der einzelnen Schwerpunkte, sowie der Masse der zugehörigen Glieder, kann der Schwerpunkt folgendermaßen berechnet werden:

$$p_{CoM} = \frac{\sum_i (p_i \cdot m_i)}{\sum_i m_i} \tag{6.13}$$

Hierbei ist p_i die Position des Schwerpunktes des Gliedes i im Torsokoordinatensystem und m_i die Masse des Gliedes i.

Des Weiteren erlaubt die Vorwärtskinematik nicht nur die Transformation von Punkten in ein bestimmtes Koordinatensystem, sondern ebenso die Transformation von einem vollständigen Koordinatensystem in ein anderes. Wird im vorigen Beispiel der Punkt ${}^{Foot}p$ durch die Matrix

$${}^{Foot}T = \begin{pmatrix} I & {}^{Foot}p \\ 0 & 1 \end{pmatrix}$$

ersetzt, so ist das Ergebnis der Transformation kein Spaltenvektor mit den Koordinaten des Punktes, sondern eine 4×4 Matrix im Format einer Transformationsmatrix, die neben der Positionsinformation ebenso die Orientierungsinformation beinhaltet. Somit kann die Orientierung des Fußes relativ zum Torso berechnet werden.

6.2. Inverse Kinematik

Die inverse Kinematik stellt das Gegenstück zur Vorwärtskinematik dar. Die Vorwärtskinematik diente dazu, die Position und Orientierung eines Robotergliedes bei Kenntnis der Gelenkstellungen zu berechnen. Häufig möchte man aber ein bestimmtes Körperteil an eine bestimmte Position bewegen und eine bestimmte Orientierung erreichen, ohne die Gelenkwinkel zu kennen, die zu genau dieser Position führen. Dieses Problem kann durch inverse Kinematik gelöst werden. Bei Vorgabe der Position und Orientierung eines Gliedes werden die zugehörigen Gelenkwinkel berechnet.

Die Lösung dieses Problems ist allerdings nicht in allen Fällen eindeutig. So können mehrere Lösungen zur gleichen Orientierung und Position eines bestimmten Gliedes führen. Ein Beispiel hierzu ist in Abbildung 6.3 zu sehen. Des Weiteren besteht die

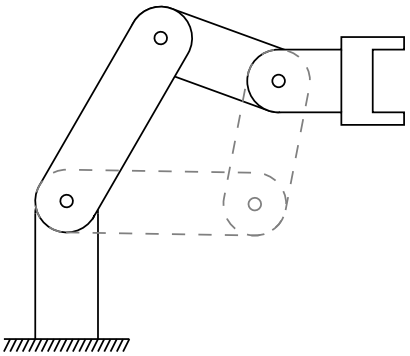


Abb. 6.3.: Mehrere Lösungsmöglichkeiten um Zielposition und -orientierung des Endgliedes zu erreichen

Möglichkeit, dass keine passende Lösung für das Problem existiert. Dies ist unter anderem immer dann der Fall, wenn die gewünschte Position außerhalb des Arbeitsbereiches des Roboters liegt. Es ist aber auch möglich, dass die vorgegebene Position eines Gliedes erreicht werden kann, allerdings nicht in Zusammenhang mit der gewünschten Orientierung. In diesen Fällen wird meist versucht, die Position und Orientierung zu finden, bei der eine möglichst geringe Abweichung zur vorgegeben Position und Orientierung besteht.

Zur Lösung des Problems gibt es verschiedene Ansätze. In manchen Fällen ist es möglich die Gelenkwinkel analytisch zu berechnen. Für eine Gelenkkette mit sechs Freiheitsgraden, wie es für das Bein des *Naos* der Fall ist, besteht eine analytische Lösung, wenn sich die Drehachsen drei aufeinander folgender Aktoren in einem Punkt schneiden, wie es bei einem Kugelgelenk der Fall ist. Zusätzlich müssen ebenfalls drei aufeinander folgende Drehachsen parallel zueinander verlaufen. Dies ist eine hinreichende Bedingung für die Existenz einer analytischen Lösung [SK08]. Für die Beine des *Naos* sind beide Bedingungen erfüllt.

Neben den analytischen Methoden gibt es die numerischen Methoden, die sich durch Iterationsprozesse der Lösung des Problems annähern. Da diese Methoden rechenaufwendiger sind als die analytischen Methoden, ist bei Existenz einer analytischen

Lösung, die letztere natürlich vorzuziehen.

Im Folgenden werden die Methoden zur Berechnungen der Bein- und Armgelenkwinkel für den *Nao* erklärt.

6.2.1. Inverse Beinkinematik

Die Beine des *Naos* bestehen aus Hüfte, Knie und Fuß. Gesteuert wird das Bein durch sechs Servomotoren, wovon drei das Hüftgelenk, einer das Kniegelenk und die restlichen zwei das Knöchelgelenk bilden.

Die Berechnung der Beingelenkwinkel für eine definierte Fußposition und -orientierung erfolgt nach dem in [GHRL09] vorgestellten Verfahren. Es ist die analytische Lösung zu diesem Problem.
In Abbildung 6.4 sind die Beinaktoren mit den jeweiligen Drehachsen eingezeichnet. Zuvor wurden die Bedingungen für die Existenz einer analytischen Lösung des Pro-

(a) HipYawPitch (b) HipRoll (c) HipPitch

(d) KneePitch (e) AnklePitch (f) AnkleRoll

Abb. 6.4.: Beinaktoren mit eingezeichneten Drehachsen [Ald11]

blems definiert. Die drei aufeinander folgenden Aktoren, deren Achsen sich in einem Punkt schneiden, sind die Aktoren der Hüfte (*HipYawPitch*, *HipRoll*, *HipPitch*). Die

Aktoren, deren Drehachsen parallel zueinander liegen, liegen in der Hüfte, dem Knie und dem Knöchel (*HipPitch*, *KneePitch*, *AnklePitch*). Da dies eine hinreichende Bedingung ist, kann das Problem also analytisch gelöst werden.

Bei den Beinen des *Naos* gibt es die Besonderheit, dass die Drehachse des Gelenks *HipYawPitch* um 45° rotiert ist, und dass beide Beine über diesen Aktor gekoppelt sind. Es ist also nicht möglich die Beine unabhängig voneinander um diese Achse zu drehen. Diese Besonderheit muss bei der Lösung des Problems berücksichtigt werden.

Für die Berechnung wird angenommen, dass eine Fußposition und -orientierung im Torsokoordinatensystem definiert wird. Diese Information wird in einer Transformationsmatrix gespeichert (Gleichung 6.8). Die Informationen über den Fuß sind also zunächst in der Form

$$^{Torso}T_{Foot}$$

vorhanden. Für die weiteren Berechnungen ist es sinnvoll, nicht die Fußposition, sondern die Knöchelposition heranzuziehen. Da der Knöchel gegenüber dem Fuß nur verschoben ist, kann die definierte Fußposition in eine Knöchelposition überführt werden.

$$^{Torso}T_{Ankle} = (Trans_z(l_{Foot,z}) \cdot {}^{Torso}T_{Foot}^{-1})^{-1}$$

Die Größe $l_{Foot,z}$ entspricht hierbei der Fußhöhe. Diese und weitere Größen sind Abbildung 2.4 zu entnehmen. Die Invertierungen sind notwendig, da die Verschiebung entlang der z-Achse des Fußkoordinatensystems erfolgen muss. Hierzu wird zunächst in das Fußkoordinatensystem transformiert, dann die Verschiebung durchgeführt, und anschließend wieder in das Torsokoordinatensystem transformiert, um die gewünschte Knöchelposition im Torsokoordinatensystem zu erhalten.

Die nächste Transformation findet in das Hüftkoordinatensystem statt.

$$^{Hip}T_{Ankle} = Trans_z(l_{Hip,z}) \cdot Trans_y(l_{Hip,y}) \cdot {}^{Torso}T_{Ankle} \tag{6.14}$$

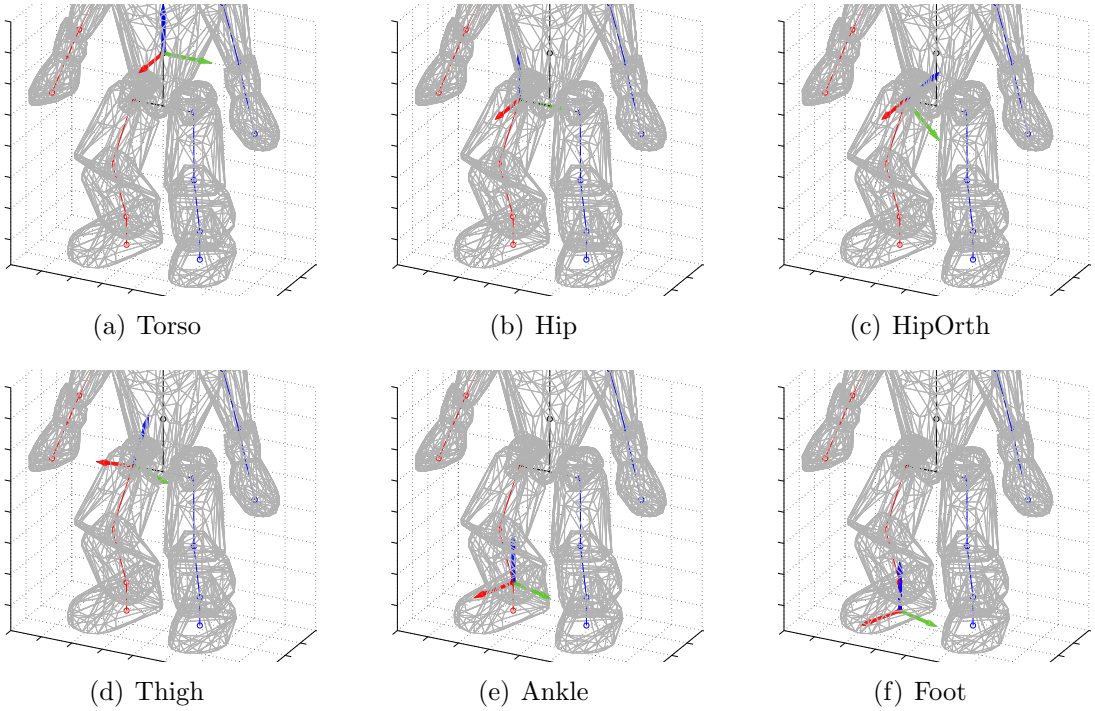
Die Rotation des *HipYawPitch* wird durch Transformation in das um 45° gedrehte Koordinatensystem *HipOrth* berücksichtigt.

$$^{HipOrth}T_{Ankle} = Rot_x\left(\frac{\pi}{4}\right) \cdot {}^{Hip}T_{Ankle} \tag{6.15}$$

Eine Übersicht der Koordinatensysteme ist in Abbildung 6.5 dargestellt. Zur Berechnung der Knie- und Knöchelwinkel wird zunächst das gerade erhaltene Koordinatensystem *HipOrth* durch *Ankle* ausgedrückt.

$$^{Ankle}T_{HipOrth} = {}^{HipOrth}T_{Ankle}^{-1} \tag{6.16}$$

Der Kniewinkel kann mit Hilfe des Kosinussatzes aus dem Dreieck, das durch den Oberschenkel (Thigh), das Schienenbein (Tibia) und dem Abstand von Hüfte zu Knöchel, der aus $^{Ankle}T_{HipOrth}$ hervorgeht, berechnet werden. Abbildung 6.6 zeigt eine zweidimensionale Skizze der Anordnung.



(a) Torso (b) Hip (c) HipOrth

(d) Thigh (e) Ankle (f) Foot

Abb. 6.5.: Beinkoordinatensysteme: x-Achse: rot, y-Achse: grün, z-Achse: blau

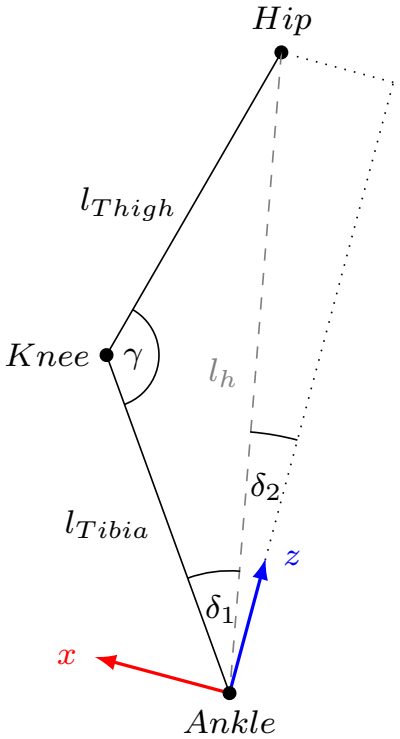


Abb. 6.6.: Zweidimensionale Skizze des Beines mit eingezeichneten Größen zur Berechnung der Winkel für *KneePitch* und *AnklePitch*

$$\gamma = arccos \frac{l_{Thigh}^2 + l_{Tibia}^2 - l_h^2}{2 \cdot l_{Thigh} \cdot l_{Tibia}} \tag{6.17}$$

Der Winkel γ entspricht allerdings nicht dem tatsächlichen Kniewinkel, der für den Nao eingestellt werden muss, denn die Definition ist so gewählt, dass das Bein vollständig gestreckt ist, wenn ein Kniewinkel von null Grad eingestellt wird. Daher ergibt sich der tatsächliche Kniewinkel zu

$$\delta_{KneePitch} = \pi - \gamma \tag{6.18}$$

Der Winkel *AnklePitch* wird zweigeteilt berechnet. Der erste Teil δ_1 kann ebenfalls mit Hilfe des Kosinussatzes berechnet werden.

$$\delta_1 = arccos \frac{l_{Tibia}^2 + l_h^2 - l_{Thigh}^2}{2 \cdot l_{Tibia} \cdot l_h} \tag{6.19}$$

Der zweite Teil δ_2 kann aus der Position der Hüfte im Fußkoordinatensystem bestimmt werden.

$$\delta_2 = \text{atan2}(x, \sqrt{y^2 + z^2}) \tag{6.20}$$

Hierbei ist die Funktion atan2 eine erweiterte Arkustangensfunktion, welche aus der Vorzeicheninformation der übergebenen Parameter den Quadranten und somit das Vorzeichen des Winkels bestimmt. Da als Skizze eine zweidimensionale Darstellung gewählt wurde, ist die y-Komponente nicht eingezeichnet. Diese muss im dreidimensionalen Raum allerdings mit berücksichtigt werden. Der resultierende Winkel ergibt sich zu

$$\delta_{AnklePitch} = \delta_1 + \delta_2 \tag{6.21}$$

Der noch fehlende Knöchelwinkel *AnkleRoll* kann ebenfalls aus der Lage der Hüfte im Fußkoordinatensystem bestimmt werden. Er ergibt sich zu:

$$\delta_{AnkleRoll} = \text{atan2}(y, z) \tag{6.22}$$

Die Größen x, y und z in den Gleichungen geben hierbei die Position der Hüfte im Fußkoordinatensystem an. Drei der sechs Beinwinkel konnten bisher bestimmt werden, und werden nun genutzt, um die Stellung des Oberschenkels relativ zum Knöchel zu bestimmen.

$$\begin{aligned} {}^{Ankle}T_{Thigh} = & Rot_x(\delta_{AnkleRoll}) \cdot Rot_y(\delta_{AnklePitch}) \cdot Trans_z(l_{Tibia}) \cdot \\ & Rot_y(\delta_{KneePitch}) \cdot Trans_z(l_{Thigh}) \end{aligned} \tag{6.23}$$

Die soeben berechnete Matrix wird wiederum dazu benutzt, das um 45° gedrehte Hüftkoordinatensystem *HipOrth* durch das Koordinatensystem *Thigh* beschreiben zu können. Nach Gleichung 6.11 gilt:

$${}^{Thigh}T_{HipOrth} = {}^{Ankle}T_{Thigh}^{-1} \cdot {}^{Ankle}T_{HipOrth} \tag{6.24}$$

Diese beiden Koordinatensysteme sind nicht gegeneinander verschoben, sondern nur rotiert. Die Information der Drehung steckt in der Rotationsmatrix, die in ${}^{Thigh}T_{HipOrth}$ enthalten ist. Die Reihenfolge der Verdrehung ist durch die kinematische Kette der Aktoren in der Hüfte bestimmt. Das erste Glied (*HipYawPitch*) dreht um die z-Achse, das zweite Glied (*HipRoll*) um die x-Achse und das dritte Glied (*HipPitch*) um die y-Achse. Eine Rotationsmatrix, die Drehungen in dieser Reihenfolge beschreibt, hat folgende Gestalt:

$$\begin{aligned} Rot_{z,x,y} &= Rot_z(\delta_z) \cdot Rot_x(\delta_x) \cdot Rot_y(\delta_y) \\ &= \begin{pmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{pmatrix} \end{aligned} \tag{6.25}$$

Hierbei gilt $c_x = cos(\delta_x)$ und $s_x = sin(\delta_x)$ und Entsprechendes für die anderen Winkel. Aus den Elementen dieser Matrix können nun die Winkel $\delta_{HipYawPitch}$, $\delta_{HipRoll}$ und $\delta_{HipPitch}$ bestimmt werden.

$$\begin{aligned} \delta_{HipRoll} &= \delta_x - \tfrac{\pi}{4} &&= \arcsin(r_{32}) - \tfrac{\pi}{4} \\ \delta_{HipPitch} &= \delta_y &&= \text{atan2}(-r_{31}, r_{33}) \\ \delta_{HipYawPitch} &= \delta_z &&= \text{atan2}(-r_{12}, r_{22}) \end{aligned} \tag{6.26}$$

Hierbei beschreiben die verschiedenen r_{ij} die Elemente der Matrix $Rot_{z,x,y}$.

Nachdem alle Beinwinkel für die passende Fußposition berechnet sind, bleibt das Problem offen, dass für das andere Bein bereits ein Winkel, wegen der Kopplung über das *HipYawPitch*-Gelenk, festgelegt ist. Es ist möglich, für beide Beine zunächst die passenden Winkel zu berechnen und anschließend einen mittleren Winkel für dieses Gelenk anzusetzen. Beim Gehen ist es allerdings so, dass das Supportbein seine Position möglichst exakt erreichen soll. Hierbei werden die Winkel dann zunächst für das Supportbein berechnet und der erhaltene Winkel für das *HipYawPitch*-Gelenk auch für das andere Bein angenommen. Dies ist meist kein Problem, da die Position und Orientierung des Schwungbeines nicht so genau erreicht werden müssen.

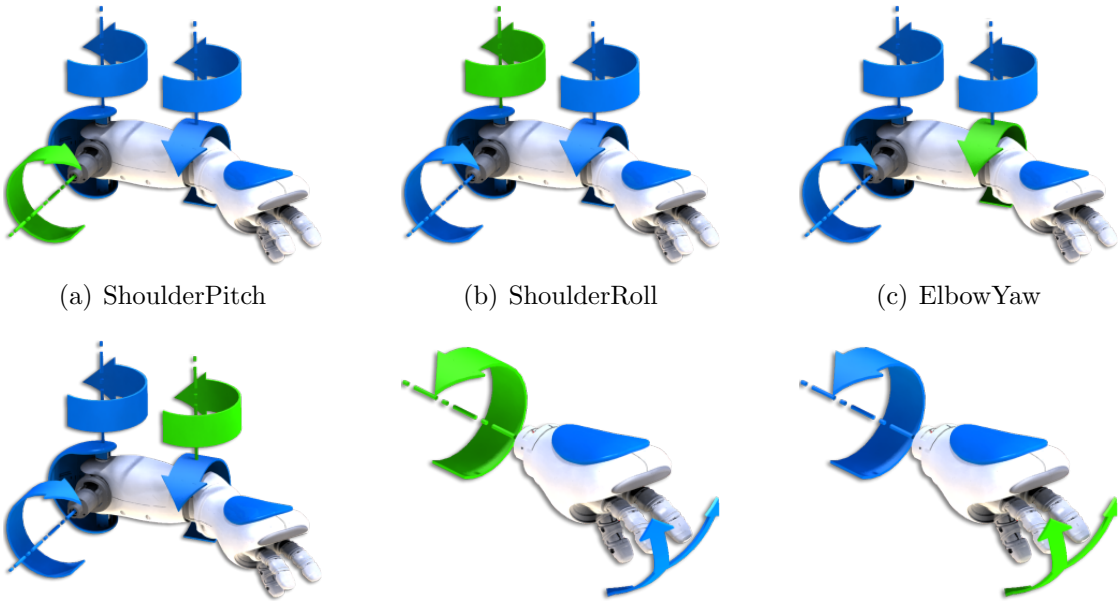
Die Berechnung der Beinwinkel mit festgesetztem *HipYawPitch*-Gelenk erfolgt ähnlich wie zuvor, allerdings in umgekehrter Reihenfolge. Aus der Sollposition und -orientierung des Fußes können die Hüftwinkel und der Kniewinkel bestimmt werden. Anschließend wird mit diesen Winkeln eine Transformation in das Knöchelkoordinatensystem vorgenommen. Die Verdrehung dieses Koordinatensystems zum Sollkoordinatensystem des Knöchels beinhaltet die restlichen Informationen, die zur Berechnung der Knöchelwinkel notwendig sind.

Bei allen Berechnungen ist darauf zu achten, dass Winkel berechnet werden, die für das Gelenk erreichbar sind. Werden Winkel bestimmt, für die das nicht erfüllt ist,

können diese begrenzt werden und die weiteren Berechnungen auf den begrenzten Winkeln erfolgen. In manchen Fällen ist es allerdings auch notwendig, die Sollposition des Fußes in den Arbeitsbereich des Roboters zu transformieren.

6.2.2. Inverse Armkinematik

Die Arme des *Naos* bestehen jeweils aus Oberarm, Unterarm und der Hand. Gesteuert wird der Arm durch sechs Servomotoren, wovon zwei das Schultergelenk und zwei weitere das Ellenbogengelenk bilden. Ein weiterer realisiert das Handgelenk. Der verbleibende Motor dient dazu, über Seilzüge die Finger zu bewegen. Somit ist es dem Roboter möglich, die Hand zu öffnen und zu schließen.
Die einzelnen Freiheitsgrade werden im Folgenden als *ShoulderPitch*, *ShoulderRoll*, *ElbowRoll*, *ElbowYaw*, *WristYaw* und *Hand* bezeichnet. In Abbildung 6.7 sind die jeweiligen Drehachsen der einzelnen Motoren dargestellt.



(a) ShoulderPitch (b) ShoulderRoll (c) ElbowYaw

(d) ElbowRoll (e) WristYaw (f) Hand

Abb. 6.7.: Armaktoren mit eingezeichneten Drehachsen [Ald11]

Die Reihenfolge dieser Motoren entspricht der kinematischen Kette des Armes. Dies bedeutet für einen bestimmten Motor, dass alle nachfolgenden Motoren an diesem aufgehängt sind.
Die inverse Kinematik des Armes ermöglicht es, für eine bestimmte Handposition sowie -orientierung passende Gelenkstellungen zu berechnen. In den folgenden Unterkapiteln wird schrittweise die implementierte inverse Kinematik hergeleitet und erläutert [2].

[2] Aufgrund eines fehlerhaften Datenblattes über die Körperabmessungen des Naos ist der in Abbildung 2.4 eingezeichnete ElbowOffsetY bei der nachfolgenden Berechnung nicht berücksichtigt.

Definition der Handnullposition

Als Handnullposition wird im Folgenden die Stelle der Hand bezeichnet, die das vorgegebene Ziel erreichen wird. Der Hersteller *Aldebaran Robotics* definiert diese Position als die Stelle, um welche die Finger des *Nao* greifen. Diese Position ist im Koordinatensystem des Handgelenks durch einen Offset in x- sowie in z-Richtung beschrieben (siehe Abbildung 6.8). Diese Definition ist zwar sinnvoll, bereitet bei der Berechnung der inversen Kinematik allerdings erhebliche Probleme. Diese gehen soweit, dass eine geeignete Gelenkstellung für die vorgegebene Handposition nur durch Iteration über mehrere Freiheitsgrade gefunden werden kann.

Vernachlässigt man den Offset in z-Richtung, so vereinfacht dies das Problem erheblich. In diesem Fall liegt die Handnullposition in Verlängerung des Unterarmes. Der Grund für die Vereinfachung liegt darin, dass durch Verlagerung der Handnullposition auf die Drehachse des *WristYaw*, nur noch 4 Freiheitsgrade für die Positionierung der Hand eine Rolle spielen. Durch Drehung um die WristYaw-Achse wird nun keine Positionsänderung der Handnullposition mehr bewirkt. Da der vernachlässigte Offset klein ist, und ein gezieltes Greifen für diese Arbeit keine Rolle spielt, kann diese Vereinfachung aber gemacht werden.

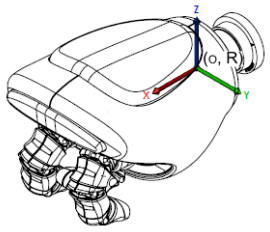


Abb. 6.8.: Hand des *Naos* [Ald11]

Berechnung der Armgelenkwinkel

Die Berechnung der Armgelenkwinkel ist für Punkte, die sowohl in der gewünschten Orientierung, als auch positionsgenau erreichbar sind, ein Problem, das recht einfach und analytisch gelöst werden kann. Leider ist dies nur für wenige Punkte im Raum der Fall. Einige Punkte können zwar positionsgenau erreicht werden, allerdings nicht in allen Orientierungen. Andere Punkte wiederum sind überhaupt nicht erreichbar. Begrenzt wird der Raum der erreichbaren Punkte zum einen durch die maximale Länge des Armes, zum anderen aber auch durch die Begrenzung der Drehwinkel für die einzelnen Freiheitsgrade.
Zunächst wird der Fall betrachtet, dass ein Punkt in Orientierung und Position exakt erreichbar ist. In diesem Fall kann zunächst von der gegebenen Handposition eine Verschiebung entlang der *WristYaw*-Achse, die durch die Orientierung der Hand gegeben ist, durchgeführt werden. Der Betrag der Verschiebung entspricht der Summe aus Handlänge und Unterarmlänge. Der gefundene Punkt stellt die Position dar, die der Ellenbogen erreichen muss. In Abbildung 6.9 ist diese Verschiebung dargestellt. Zusätzlich zur Verschiebung sind ebenfalls das Handkoordinatensystem sowie das Schulterkoordinatensystem eingezeichnet, wobei die x-Achse durch den roten, die y-Achse durch den grünen und die z-Achse durch den blauen Pfeil gekennzeichnet sind. Diese Konvention wird im Folgenden beibehalten. Die Orientierung des Schul-

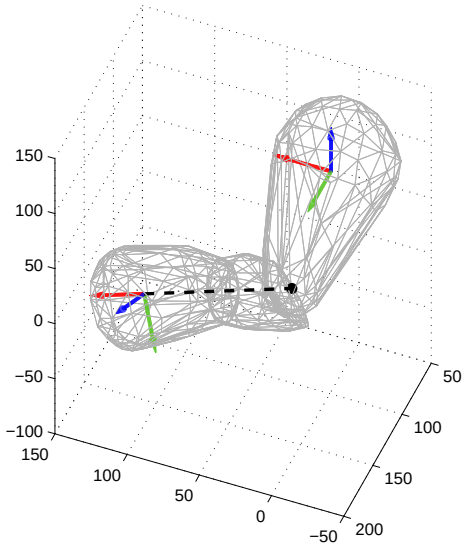


Abb. 6.9.: Position des Ellenbogens aus Verschiebung entlang der *WristYaw*-Achse

terkoordinatensystems entspricht der des Torsokoordinatensystems. Es ist lediglich entlang der y- und z-Achse verschoben.

Aus der zuvor ermittelten Ellenbogenposition können bereits für zwei Freiheitsgrade die Winkel bestimmt werden. In Abbildung 6.10 ist die Ellenbogenposition im Schulterkoordinatensystem dargestellt. Der eingezeichnete Winkel α entspricht da-

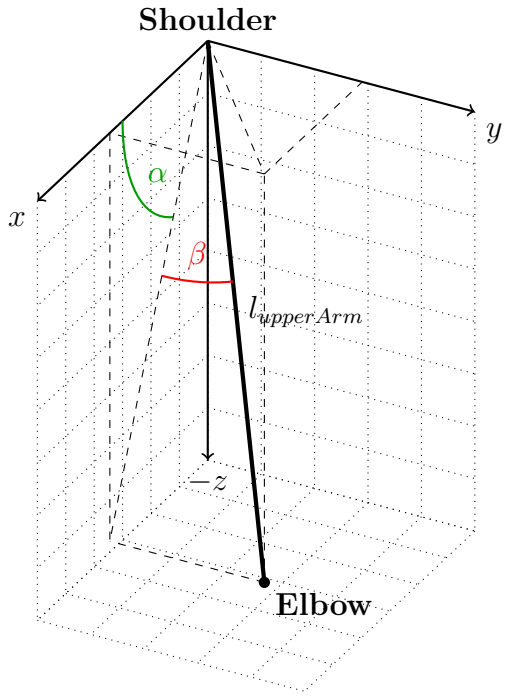


Abb. 6.10.: Ellenbogenposition im Schulterkoordinatensystem

bei dem *ShoulderPitch*- und β dem *ShoulderRoll*-Winkel. Im Einzelnen lassen sich

die Winkel folgendermaßen berechnen:

$$\begin{aligned} \delta_{ShoulderPitch} &= \alpha &&= \arcsin \frac{y}{l_{upperArm}} \\ \delta_{ShoulderRoll} &= \beta &&= \text{atan2}(z, x) \end{aligned} \tag{6.27}$$

x,y,z = Ellenbogenkoordinaten, $l_{upperArm}$ = Oberarmlänge

Ein weiterer Winkel kann direkt aus dem Abstand der vorgegebenen Handposition zum Schultergelenk berechnet werden. Abbildung 6.11 zeigt eine Skizze des Armes im Schulterkoordinatensystem. Der Winkel δ, der die *ElbowRoll*-Stellung angibt,

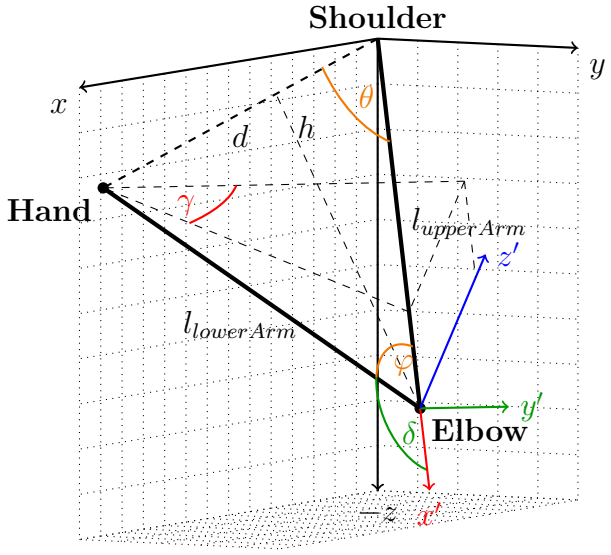


Abb. 6.11.: Arm im Schulterkoordinatensystem mit δ = ElbowRoll, γ = ElbowYaw

kann vom Winkel φ abgeleitet werden. Dieser lässt sich mit Hilfe des Kosinussatzes bestimmen.

$$\varphi = \arccos \frac{l_{lowerArm}^2 + l_{upperArm}^2 - d^2}{2 \cdot l_{lowerArm} \cdot l_{upperArm}} \tag{6.28}$$

$$\delta_{ElbowRoll} = \delta = \pi - \varphi \tag{6.29}$$

$$d = \sqrt{x^2 + y^2 + z^2} \tag{6.30}$$

$l_{upperArm}$ = Oberarmlänge, $l_{lowerArm}$ = Unterarmlänge + Handlänge, x,y,z = Handposition

Der ebenfalls eingezeichnete Winkel γ entspricht dem des *ElbowYaw*. Um diesen berechnen zu können, ist zunächst eine Transformation des Schulterkoordinatensystems in das Ellenbogenkoordinatensystem notwendig, dessen Achsen mit x', y' und

z' gekennzeichnet sind. Die Transformation entspricht einer Rotation und Translation des Schulterkoordinatensystems. Die Handposition im Ellenbogenkoordinatensystem kann durch Multiplikation der Transformationsmatrix mit der Handposition im Schulterkoordinatensystem ermittelt werden.

$$^{Elbow}T_{Shoulder} = Trans_z(l_{upperArm}) \cdot Rot_z(\beta) \cdot Rot_y(\alpha) \tag{6.31}$$

$$^{Elbow}T_{Hand} = {}^{Elbow}T_{Shoulder} \cdot {}^{Shoulder}T_{Hand} \tag{6.32}$$

Der Winkel δ entspricht dem Drehwinkel um die x'-Achse, sodass y' in der Ebene des durch die Arme aufgespannten Dreiecks liegt. Die z'-Achse steht dann senkrecht auf dieser Ebene.

$$\delta_{ElbowYaw} = \gamma = \text{atan2}(z', y') \tag{6.33}$$

z', y' = Handkoordinaten im Ellenbogenkoordinatensystem

Somit sind alle Winkel bestimmt, die für das Erreichen der vorgegebenen Handposition notwendig sind, da der *WristYaw*-Winkel keinen Einfluss auf die Position hat. Da aber nach Möglichkeit nicht nur Position sondern auch Orientierung der Hand exakt erreicht werden sollen, muss dieser Winkel dennoch bestimmt werden. Zunächst wird eine Transformation in das Handbasiskoordinatensystem *HandBase* vorgenommen. Es unterscheidet sich vom Handkoordinatensystem dahingehend, dass die Rotation unterschiedlich sein kann.

$$^{HandBase}T_{Elbow} = Trans_x(l_{lowerArm}) \cdot Rot_z(\delta_{ElbowRoll}) \cdot Rot_x(\delta_{ElbowYaw}) \tag{6.34}$$

$$^{HandBase}T_{Hand} = {}^{HandBase}T_{Elbow} \cdot {}^{Elbow}T_{Hand} \tag{6.35}$$

Da zuvor angenommen wurde, dass die Handposition in Orientierung und Position exakt erreichbar ist, beinhaltet die Matrix $^{HandBase}T_{Hand}$ lediglich die Drehinformation um die *WristYaw*-Achse, die in diesem Fall die x-Achse des Handkoordinatensystems ist. Somit gilt:

$$^{HandBase}T_{Hand} = Rot_x(\delta_{WristYaw}) \tag{6.36}$$

Aus dieser Matrix kann der Drehwinkel bestimmt werden. Der letzte verbleibende Freiheitsgrad *Hand* bleibt außer Betracht, da er nur zum Öffnen und Schließen der Hand dient, auf Position und Orientierung aber keinen Einfluss hat.

Orientierung oder Position nicht exakt erreichbar

Wie zuvor erwähnt ist der Fall, dass eine vorgegebene Handposition und Orientierung exakt erreicht werden kann, äußerst unwahrscheinlich. Ein Beispiel hierfür ist in Abbildung 6.12 zu sehen. Die roten und blauen Linien stellen jeweils den Arm für verschiedene Vorgaben der Handposition dar. In diesem Beispiel wurde lediglich der

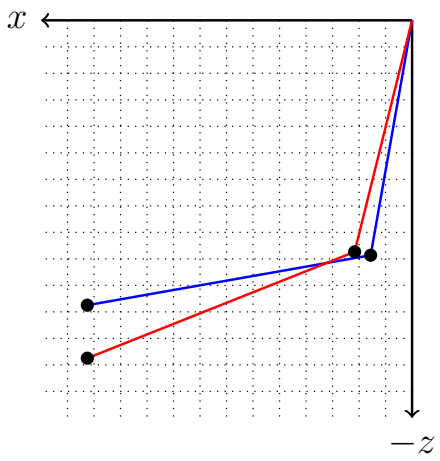


Abb. 6.12.: Orientierung der Hand für unterschiedliche Positionen

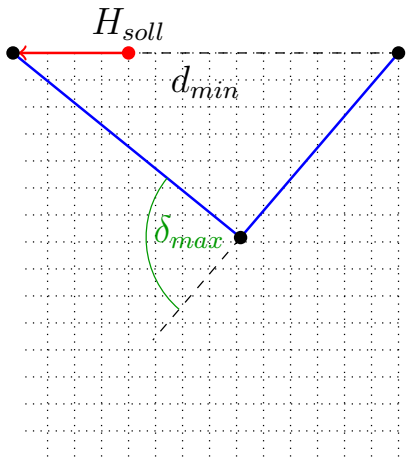


Abb. 6.13.: Verschiebung der Handpositionsvorgabe auf erreichbare Position

z-Parameter der Handposition verändert. Es ist zu erkennen, dass eine leicht Veränderung der Position dazu führt, dass die Orientierung der Hand, die in Verlängerung des Unterarmes liegt, nicht mehr dieselbe sein kann. Für jeden Punkt, der mit einer bestimmten Orientierung erreichbar ist, gibt es unendlich viele Punkte in direkter Umgebung, die mit derselben Orientierung nicht erreichbar sind. Von daher ist das bisherige Vorgehen zur Bestimmung der Armgelenkwinkel meist nicht zielführend. Für die folgende Herleitung der inversen Kinematik spielt die Orientierung der Hand eine untergeordnete Rolle. Es wird in erster Linie versucht die vorgegebene Position zu erreichen. Ein Winkel, der direkt aus der Handposition abgeleitet werden kann, ist wie zuvor erwähnt, der des *ElbowRoll*. Er kann nach Gleichung 6.30 berechnet werden. Allerdings muss der Abstand von Schulter zu Hand im zulässigen Bereich liegen. Dieser Bereich ist durch die Maximal- und Minimalstellung des *ElbowRoll* sowie die Länge der Gliedmaßen gegeben. Liegt die gewünschte Handposition außerhalb des zulässigen Bereiches, so wird diese entlang des Vektors, der von Schulter zu Hand zeigt, in den zulässigen Bereich verschoben. Abbildung 6.13 zeigt dies skizzenhaft für eine Handsollposition H_{Soll}, die zu nahe am Schultergelenk liegt und daher nicht erreicht werden kann. Ist der *ElbowRoll*-Winkel bekannt, so gibt es mehrere mögliche Positionen für den Ellenbogen, die sich alle auf einem Kreis befinden, dessen Mittelpunkt auf der Verbindungslinie zwischen Schulter und Hand liegt (siehe Abb. 6.14). Der Normalenvektor $\vec{n}$ der Ebene, in der sich der Kreis befindet, ist durch den Vektor von Schulter zu Hand gegeben. Der Radius r des Kreises entspricht der Höhe h des Dreiecks, das durch Schulter, Ellenbogen und Hand aufgespannt wird.

Diese Höhe kann, ebenso wie der Abstand des Kreismittelpunktes zur Schulter d, mit Hilfe des Winkels θ (siehe Abb. 6.11) berechnet werden.

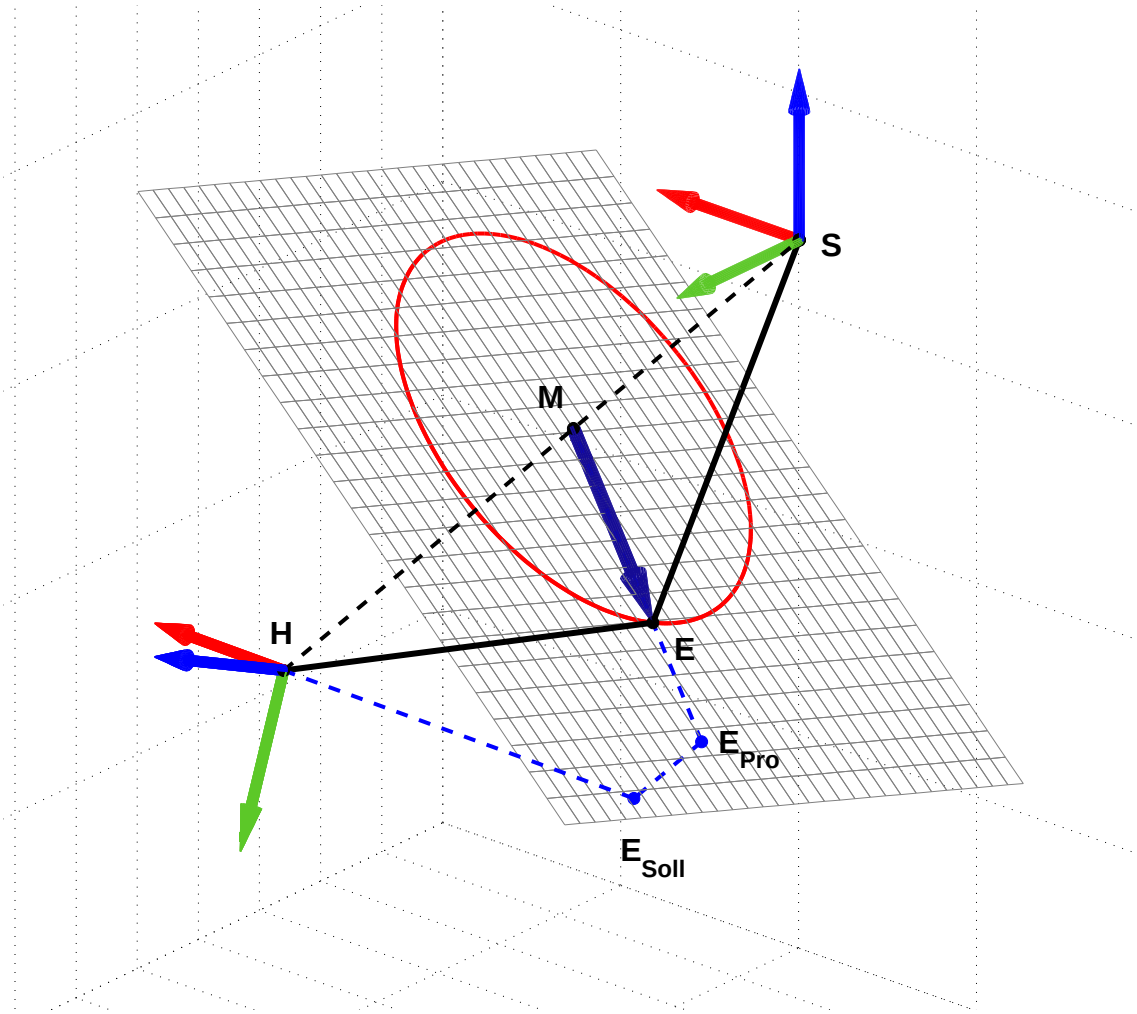


Abb. 6.14.: Mögliche Ellenbogenpositionen für vorgegebene Handposition. H = Hand, S = Schulter, E = Ellenbogen, E_{Soll} = Ellenbogenposition um Ziel in Position und Orientierung zu erreichen, E_{Pro} = In Kreisebene Projezierte Ellenbogenposition

$$h = \sin\theta \cdot l_{upperArm} = r \tag{6.37}$$

$$d = \cos\theta \cdot l_{upperArm} \tag{6.38}$$

mit

$$\theta = \arccos \frac{d^2 + l_{upperArm}^2 - l_{lowerArm}^2}{2 \cdot d \cdot l_{upperArm}} \tag{6.39}$$

Die Kreisebene kann in der Hesseschen Normalform mit normiertem $\vec{n}$ wie folgt beschrieben werden:

$$E : \vec{n} \cdot \vec{p} = d \tag{6.40}$$

Hierbei ist $\vec{p}$ der Ortsvektor zu einem Punkt in der Ebene. Im Folgenden werden alle Punkte als Ortsvektoren im Schulterkoordinatensystem betrachtet und als solche gekennzeichnet.
Da der Ellenbogen $\vec{E}$ eine Position auf dem Kreis annehmen muss, um die vorgegebene Handposition $\vec{H}$ exakt zu erreichen, muss eine geeignete Kreisposition ermittelt werden, für die der Rotationsfehler der Hand klein gehalten wird. Hierzu wird zunächst, ebenso wie im vorigen Abschnitt, eine Verschiebung entlang der *WristYaw*-Achse durchgeführt, um $\overrightarrow{E_{Soll}}$ zu erhalten. Dieser Punkt muss anschließend in die Kreisebene projiziert werden. Hierzu wird zunächst der Abstand s von $\overrightarrow{E_{Soll}}$ zur Ebene ermittelt. Anschließend berechnet sich $\overrightarrow{E_{Pro}}$ zu:

$$s = \vec{n} \cdot \overrightarrow{E_{Soll}} - d \tag{6.41}$$

$$\overrightarrow{E_{Pro}} = \overrightarrow{E_{Soll}} - s \cdot \vec{n} \tag{6.42}$$

Vom ermittelten Punkt $\overrightarrow{E_{Pro}}$ muss der nächstgelegene Punkt auf dem Kreis ermittelt werden. Hierzu wird der Vektor $\vec{v}$ von Mittelpunkt $\vec{M}$ zu $\overrightarrow{E_{Pro}}$ berechnet und normiert. Die gesuchte Ellenbogenposition $\vec{E}$ befindet sich im Abstand r zum Mittelpunkt entlang dieses Vektors.

$$\vec{M} = \vec{n} \cdot d \tag{6.43}$$

$$\vec{v} = \frac{\overrightarrow{E_{Pro}} - \vec{M}}{\left|\overrightarrow{E_{Pro}} - \vec{M}\right|} \tag{6.44}$$

$$\vec{E} = \vec{M} + \vec{v} \cdot r \tag{6.45}$$

Es wurde also eine Position für den Ellenbogen ermittelt, jedoch ist noch nicht klar, ob diese Ellenbogenposition auch erreichbar ist. Die Grenzen für die Erreichbarkeit sind durch die maximalen und minimalen Winkel des Schultergelenkes gegeben. Der *ElbowRoll*-Winkel beeinflusst nur die y-Koordinate des Ellenbogens, sodass die Grenzen der y-Koordinate im Schulterkoordinatensystem durch

$$y_{Roll}^{max} = \sin(\delta_{ShoulderRoll}^{max}) \cdot l_{upperArm} \tag{6.46}$$

$$y_{Roll}^{min} = \sin(\delta_{ShoulderRoll}^{min}) \cdot l_{upperArm} \tag{6.47}$$

gegeben sind. Das *ShoulderPitch*-Gelenk begrenzt die Ellenbogenposition in x-Richtung. Der Verlauf der Begrenzung ist allerdings eine Funktion von y. Von z ist diese Funktion unabhängig, da der maximale und minimale ShoulderPitch-Winkel vom Betrag

gleich sind. Um die Begrenzungsfunktion $f_{xLim}(y)$ zu ermitteln, wird der Oberarm zunächst auf den Maximalwinkel $\delta^{max}_{ShoulderPitch}$ gedreht. Dies entspricht einer Rotation um die y-Achse. Anschließend erfolgt eine Rotation um die z-Achse, die dem *ShoulderRoll*-Gelenk entspricht. Da der Arm in Nulllage ($\alpha = 0$, $\beta = 0$) in x-Richtung zeigt, kann dies wie folgt geschrieben werden.

$$\vec{E} = Rot_z \cdot Rot_y \cdot \begin{pmatrix} l_{upperArm} \\ 0 \\ 0 \end{pmatrix} \tag{6.48}$$

$$\vec{E} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \cos(\delta^{max}_{ShoulderPitch}) \cdot \cos(\beta) \cdot l_{upperArm} \\ \sin(\beta) \cdot l_{upperArm} \\ -\sin(\delta^{max}_{ShoulderPitch}) \cdot \cos(\beta) \cdot l_{upperArm} \end{pmatrix} \tag{6.49}$$

$$\Rightarrow$$

$$\sin(\beta) = \frac{y}{l_{upperArm}} \tag{6.50}$$

$$x^2 = k^2 \cdot \cos^2(\beta) \cdot l^2_{upperArm} \tag{6.51}$$

$$= k^2 \cdot (1 - \sin^2(\beta)) \cdot l^2_{upperArm} \tag{6.52}$$

$$= k^2 \cdot (1 - (\frac{y}{l_{upperArm}})^2) \cdot l^2_{upperArm} \tag{6.53}$$

$$x = k \cdot \sqrt{l^2_{upperArm} - y^2} = f_{xLim}(y) \tag{6.54}$$

mit

$$k = \cos(\delta^{max}_{ShoulderPitch}) \tag{6.55}$$

Die Begrenzungen sind in Abbildung 6.15 eingezeichnet. Es ist die xy-Ebene dargestellt, sodass die Begrenzungsfunktionen als Linien erscheinen. Es ist zu erkennen, dass für die vorgegebene Handposition und Orientierung H eine Ellenbogenposition ermittelt wird, die Außerhalb der Begrenzung durch y^{Roll}_{max} liegt und daher nicht erreicht werden kann. Es können nur Kreispunkte im Bereich A erreicht werden, der durch $y^{Roll}_{max}, y^{Roll}_{min}$ und $f_{xLim}(y)$ begrenzt ist.
Die Ellenbogenposition muss in diesen Bereich verschoben werden. Man kann nun versuchen, die Schnittpunkte des Kreises mit den Begrenzungsfunktionen zu ermitteln, was für die Begrenzungen durch das *ShoulderRoll*-Gelenk auch analytisch möglich ist, jedoch nicht für die Schnittpunkte mit $f_{xLim}(y)$. Und selbst wenn diese Schnittpunkte numerisch bestimmt werden, ist noch nicht klar, ob für diesen Punkt das *ElbowYaw*-Gelenk in seinen Grenzen bleibt. Es muss also eine Ellenbogenposition gefunden werden, die sowohl auf dem Kreis liegt, als auch die Begrenzungen des

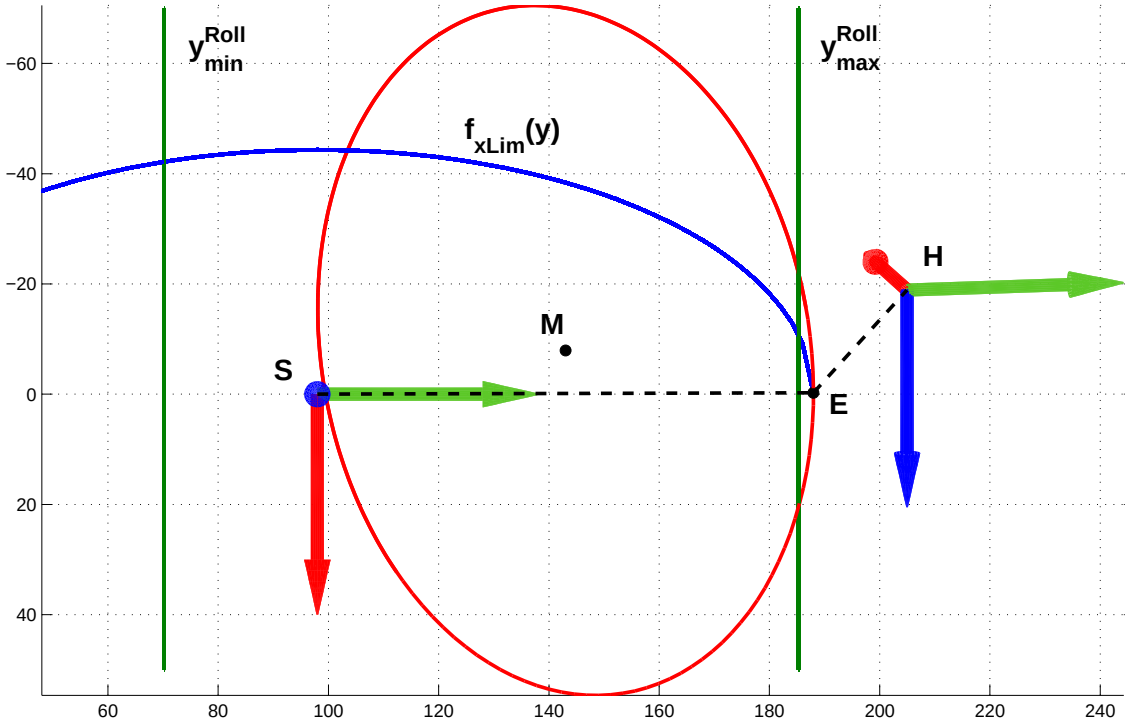


Abb. 6.15.: Begrenzung durch Schultergelenk

Schultergelenkes und des *ElbowYaw* erfüllt.
Eine Möglichkeit, einen solchen Punkt zu finden, ist es, eine Iteration auf dem Kreis durchzuführen und zu prüfen, ob die Bedingungen erfüllt sind. Hierzu muss zunächst die Kreisgleichung aufgestellt werden.

$$\vec{p} = \vec{M} + \vec{u} \cdot \sin(t) + \vec{v} \cdot \cos(t) \tag{6.56}$$

mit

$$\vec{u} \perp \vec{v} \perp \vec{n}, \qquad |\vec{u}| = |\vec{v}| = r \tag{6.57}$$

Zwei Vektoren $\vec{u}$ und $\vec{v}$ zu finden, die die Bedingungen erfüllen, kann auf verschiedene Arten geschehen. Eine davon ist, zwei Vektoren $r \cdot \vec{e_y}$ und $r \cdot \vec{e_z}$ aus dem Schulterkoordinatensystem zu drehen, sodass sie in der Kreisebene liegen. Die Drehwinkel können dann aus dem Ortsvektor $\vec{M}$ bestimmt werden. Der Vorteil dieser Vorgehensweise ist, dass man einen weiteren Drehwinkel aus der Lage von $\vec{E}$ relativ zu dem Koordinatensystem der Kreisebene (aufgespannt durch $\vec{n}$, $\vec{u}$ und $\vec{v}$) bestimmen kann, der dieses Koordinatensystem so dreht, dass $\vec{p} = \vec{E}$ für $t = 0$ gilt. Für die iterative Suche wird nun t variiert, bis ein Ellenbogenpunkt gefunden wurde, der allen Anforderungen gerecht wird. Da nicht bekannt ist, in welche Richtung auf dem Kreis iteriert werden muss, um möglichst schnell in den zulässigen Bereich zu

kommen, ist es sinnvoll, in beide Richtungen gleichzeitig zu iterieren.

Es können auch Fälle auftreten, bei denen keine der Positionen auf dem Kreis den Anforderungen gerecht wird. In einem solchen Fall ist die vorgegebene Handposition nicht erreichbar. Aus diesem Grund wird bei der Iteration für alle Punkte der Abstand von der Handposition, die erreicht werden kann, zu der vorgegebenen Handposition ermittelt. Anschließend wird die Konfiguration angenommen, für die der Abstand am geringsten ist.
Zur Ermittlung des *WristYaw*-Winkels kann wie zuvor nach den Gleichungen 6.34 bis 6.36 vorgegangen werden. Da auf die Rotationsfehler um die y- und z-Achse nicht reagiert werden kann, wird davon ausgegangen, dass nur eine Rotation um die x-Achse vorliegt. Diese wird durch das *WristYaw*-Gelenk ausgeglichen, sofern dies innerhalb dessen Begrenzungen möglich ist.

Zusammengefasst konnte hiermit ein Verfahren zur Ermittlung der Armgelenkwinkel durch inverse Kinematik entwickelt werden, das weitestgehend auf analytische Lösungsmethoden setzt und somit die Rechenzeit erheblich verkürzen kann. Es ist lediglich die Iteration in einer Variablen nötig, um die bestmögliche Gelenkstellung zu finden.

7. Implementierung

In diesem Abschnitt wird die Implementierung eines Laufalgorithmus und die zugehörige Kinematik in den Prozessablauf von *NaoQi* bzw. *Webots* beschrieben. Welches Modell für den Laufalgorithmus gewählt wird, ist hierbei nicht relevant, da die Prozessabläufe zwischen den, im nachfolgenden vorgestellten Modulen, identisch sind. Es werden in den nächsten Unterabschnitten zunächst die Module kurz beschrieben, bevor abschließend die Architektur vorgestellt wird.

7.1. Blackboard

Das bereits in Abschnitt 4.4 vorgestellte *Blackboard*-Modul dient als zentrale Speicherstelle für Informationen, die von mehreren anderen Modulen benötigt werden. Hierzu zählen alle Sensorwerte, die zyklisch aktualisiert werden. Des Weiteren sind über das *Blackboard* Roboterkonstanten, wie z.B. Körperabmessungen, Gewichte der einzelnen Glieder oder Arbeitsbereiche der Motoren, abrufbar.
Neben diesen Variablen speichert das *Blackboard* auch kinematische Informationen, die von anderen Modulen bereitgestellt werden. So werden aus den Gelenkstellungen in jedem Zyklus die Positionen und Orientierungen aller Gelenke bestimmt und gespeichert. Gleiches gilt für den Schwerpunkt des Roboters und viele weitere Variablen.

Das *Blackboard* ist als statisches Element implementiert, was in der objektorientierten Programmierung bedeutet, dass die Funktionen und Variablen, die es bereitstellt, von anderen Modulen genutzt werden können, ohne dass hierfür ein Objekt vom Typ *Blackboard* angelegt werden muss. Somit ist ein direkter und einfacher Zugriff möglich.

7.2. ForwardKinematics

Das Vorwärtskinematikmodul (*ForwardKinematics*) stellt Funktionen bereit, die die Position und Orientierung von bestimmten Gelenken berechnen können. Hierzu sind die Aktorstellungen notwendig. Den Funktionen können hierzu die Sensorwerte der Aktoren, die im *Blackboard* gespeichert sind, übergeben werden, aber auch fiktive Aktorstellungen, um zu berechnen, welche Position ein Gelenk bei bestimmten Aktorwinkeln annehmen würde. Die Übergabe von fiktiven Aktorstellungen ist besonders zur Vorhersage des Körperschwerpunktes wichtig, da dieser nicht über inverse Kinematik bestimmt werden kann. Es kann somit für eine Liste von Aktorbefehlen überprüft werden, wo sich der Schwerpunkt des Roboters befinden wird, noch bevor

diese Befehle gesendet worden sind.

Einmal pro Zyklus berechnet *ForwardKinematics* die Gelenkpositionen und -orientierungen und speichert diese im *Blackboard*. Zum Speichern dieser Informationen ist ein Datentyp `KinematicMatrix` erschaffen worden, die aus einer ebenfalls selbst implementierten 3×3 Rotationsmatrix sowie einem 3×1 Positionsvektor besteht. Im Wesentlichen bildet sie also die Form einer Transformationsmatrix nach wie sie in Gleichung 6.8 definiert wurde. Allerdings wird die letzte Zeile, die für alle Transformationsmatrizen gleich aufgebaut ist, nicht nachgebildet. Die Rechenregeln für Transformationsmatrizen, wie z.B. die Bildung der Inversen oder die Multiplikation mit anderen Transformationsmatrizen oder Vektoren, sind für die `KinematicMatrix` eigens implementiert.

7.3. Center of Mass

Das Schwerpunktmodul (*Com*) nutzt die von *ForwardKinematics* im *Blackboard* abgelegten Gelenkpositionen und -orientierungen, um einmal pro Zyklus den Roboterschwerpunkt zu berechnen und wiederum im *Blackboard* abzulegen.
Wie bereits zuvor erwähnt, kann es von Interesse sein, den Schwerpunkt für eine bestimmte Reihe von Aktorbefehlen zu kennen. In diesem Fall muss *Com* eine Liste der Gelenkpositionen und -orientierungen, welche im Format `KinematicMatrix` vorliegen muss, übergeben werden. Diese Liste kann mit dem Modul *ForwardKinematics* berechnet werden.

Das Schwerpunktmodul ist neben der Berechnung des Roboterschwerpunktes auch in der Lage, den Schwerpunkt einzelner Gelenkketten zu berechnen. Somit ist es möglich, auch nur den Schwerpunkt des Armes zu bestimmen. Solche Berechnungen können von Interesse sein, wenn z.B. der Einfluss einer Armbewegung auf die Roboterdynamik beschrieben werden soll.

7.4. InverseKinematics

Das Inverse Kinematik-Modul (*InverseKinematics*) führt die Berechnungen zur Bestimmung passender Gelenkwinkel für eine vordefinierte Position und Orientierung der Arme oder Füße durch. Hierzu muss die Zielposition und -orientierung in Form einer `KinematicMatrix` an *InverseKinematics* übergeben werden.
Aufgrund der besonderen Ausführung des Hüftgelenkes des *Naos*, bei dem sich das linke und rechte Bein einen Aktor, den *HipYawPitch*, teilen, verfügt das Modul über weitere spezielle Funktionen zur Berechnung der Beingelenkwinkel. Für ein Bein, das die Sollposition möglichst genau erreichen soll, müssen zunächst die passenden Winkel berechnet werden. Für das andere Bein werden dann die Winkel unter Berücksichtigung des zuvor ermittelten *HipYawPitch*-Winkels bestimmt.

7.5. Model

Das Modul *Model* beinhaltet das Modell, das für den Laufalgorithmus zu Grunde gelegt wird. Die in Abschnitt 5 vorgestellten Modelle geben entweder eine vollständige Trajektorie für den Schwerpunktverlauf oder eine Vorhersage zurück, wo sich der Schwerpunkt hinbewegen wird. Dieses Modul stellt also zu jedem Zeitpunkt eine Sollposition des Schwerpunktes bereit, auf dessen Grundlage die Fußpositionen durch inverse Kinematik berechnet werden können.

7.6. WalkingEngine

Das Modul *WalkingEngine* ist das Modul, das den Roboter tatsächlich veranlasst zu laufen. Es bekommt die aktuelle Schwerpunktsollposition vom Modul *Model* und berechnet mit Hilfe der inversen Kinematik die passenden Gelenkwinkel. Darüber hinaus überwacht das Modul die Schrittphasen und entscheidet, wann ein Supportwechsel erfolgen muss.
Sind alle Informationen verarbeitet und die Aktorbefehle berechnet, so sendet dieses Modul die Befehle über den, in Abschnitt 4.4 vorgestellten, *DCMConnector* an den *DCM* bzw. an den Simulator *Webots*.

Neben der Berechnung stellt *WalkingEngine* Funktionen bereit, die den Roboter zum Laufen veranlassen. Diesen Funktionen können die Zielposition und -orientierung des Roboters übergeben werden. Um eine bestimmte Zielposition und -orientierung zu erreichen, ist ein Pfadplanung notwendig. Hierzu stehen Funktionen bereit, die den Pfad entlang einer *Dubin*-Kurve planen. Die Planung des Pfades wird im Folgenden beschrieben.

Pfadplanung

Dubin-Kurven bestehen aus einzelnen Segmenten, die durch Kreisbögen oder lineare Anteile beschrieben werden können. Diese Kurven dienen eigentlich zur Pfadplanung von bereiften Fahrzeugen, sind aber einfach zu berechnen und in Code umzusetzen. Da die Kurven nur aus Kreisanteilen oder Linien bestehen, ist es möglich, die Länge des Pfades analytisch zu berechnen. Des Weiteren kann die Tangente an den Pfad in jedem Punkt einfach bestimmt werden. Die Tangente beschreibt die Fußorientierung, die der Roboter annehmen muss, um dem Pfad zu folgen. Bei *Dubin*-Kurven wird zwischen zwei Typen unterschieden. Diese sind Kreis-Strecke-Kreis-Kurven und Kreis-Kreis-Kreis-Kurven. Abbildung 7.1 zeigt ein Beispiel für beide Typen. In der Pfadplanung der *WalingEngine* werden nur Kreis-Strecke-Kreis Pfade berücksichtigt. Hierbei wird der Pfad für die vier Varianten *RSL, RSR, LSR* und *LSL* geplant. Der erste Buchstabe beschreibt hierbei die Drehrichtung auf dem ersten Kreis ($R =$ im Uhrzeigersinn, $L =$ gegen den Uhrzeigersinn), der zweite Buchstabe S beschreibt, dass der mittlere Teil durch eine Strecke gegeben ist, und der letzte Buchstabe gibt die Orientierung auf dem zweiten Kreis an. Für alle geplanten Pfade wird anschlie-

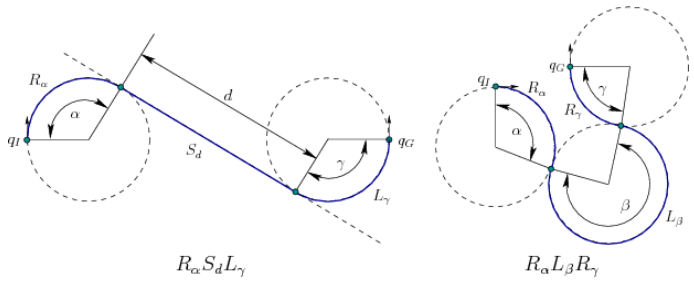
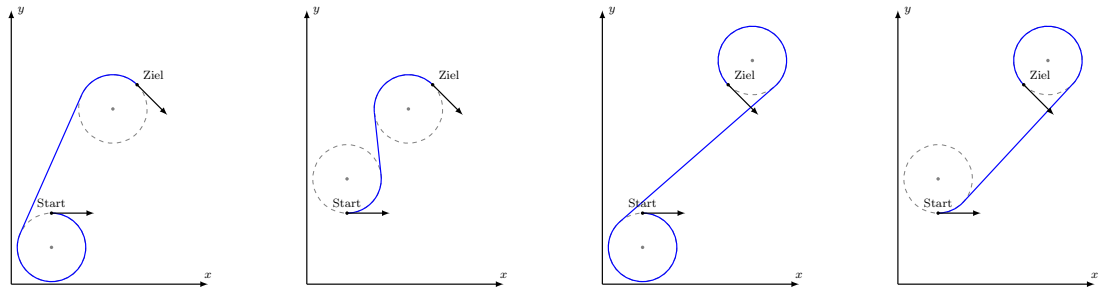


Abb. 7.1.: Dubin-Kurven: links: Kreis-Strecke-Kreis, rechts: Kreis-Kreis-Kreis [LaV06]

ßend die Länge berechnet und der Kürzeste von ihnen ausgewählt. Abbildung 7.2 zeigt die verschiedenen Pfade für eine bestimmte Zielposition und -orientierung. Für



(a) Dubin RSR (b) Dubin LSR (c) Dubin RSL (d) Dubin LSL

Abb. 7.2.: Verschiedene Dubin Pfade. Kürzester Pfad ist Dubin LSR (b)

alle Pfade wurde die gleiche Start- und Zielposition gewählt. Des Weiteren sollte der Roboter, ausgehend von seiner Startorientierung, um 45° im Uhrzeigersinn gedreht am Ziel ankommen. Der kürzeste mögliche *Dubin*-Pfad ist in Abbildung 7.2(b) zu sehen. Parameter, die bei der *Dubin*-Kurve eingestellt werden können, sind die Radien der Kreise. Für die, in dieser Arbeit umgesetzte Lösung, sind die Radien für den ersten und zweiten Kreis stets gleich groß.

7.7. Prozessarchitektur

In den vorherigen Abschnitten wurden die wichtigsten entwickelten Module vorgestellt. Neben diesen existieren zudem kleinere Module, die Datentypen repräsentieren, wie z.B. die `KinematicMatrix`, sowie deren Mathematik definieren. Da diese Module nicht zum Verständnis der Softwarearchitektur und internen Prozesse beitragen, werden sie hier nicht näher erläutert.
Die Verknüpfung der einzelnen Module untereinander ist in Abbildung 7.3 dargestellt. Einige Verbindungen der einzelnen Module zum *Blackboard* sind in der Grafik aus Gründen der Übersichtlichkeit nicht dargestellt, dennoch greifen alle Module auf im *Blackboard* gespeicherte Daten zu. Alle Aktualisierungen, die zyklisch erfolgen, werden vom *DCMConnector* gestartet, da dieser die Schnittstelle zum *DCM* bzw. zur *DCMEngine* bildet und somit die Möglichkeit besitzt, Funktionen zyklisch

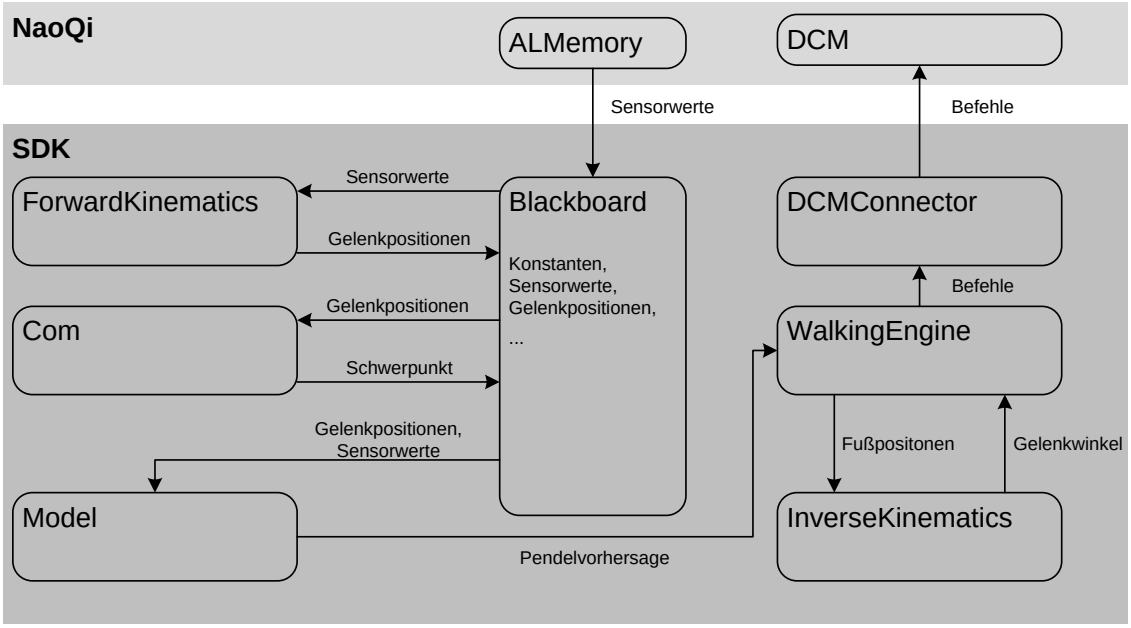


Abb. 7.3.: Architektur und Einbettung in *NaoQi*

aufrufen zu lassen. Beim echten Roboter kann eine solche Funktion an den *DCM*-Zyklus angebunden werden, im Simulator genügt dagegen ein Controller-Aufruf der Funktion innerhalb eines Simulationsschrittes.

8. Analyse und Optimierung

In diesem Kapitel wird die Analyse und Optimierung des verwendeten Modells beschrieben. Im folgenden Abschnitt wird zunächst erläutert, warum das *3DLIPM* dem *Cart-Table-Modell* vorgezogen wurde. Anschließend wird auf die Besonderheiten bei der Implementierung dieses Modelles für den *Nao*-Roboter eingegangen. Hiernach werden die Probleme, die bei der Implementierung entstanden sind, beschrieben und durch Optimierung des Modells versucht, diese zu beheben. Anschließend findet eine Evaluierung des umgesetzten Laufalgorithmus durch Messungen statt. Der letzte Abschnitt dieses Kapitels widmet sich einem neu entwickelten Ansatz zur Neuberechnung der Pendelparameter.

8.1. Auswahl des Modells

In Abschnitt 5 wurden verschiedene Modelle beschrieben, die zur Implementierung von Laufalgorithmen humanoider Roboter genutzt werden. Zunächst muss aus diesen Modellen ein geeignetes Modell ausgewählt werden.

Als Erstes wird das roboterspezifische, vollständige Modell zur Berechnung des ZMP betrachtet. Dieses Modell ist unter Umständen das Modell, dessen Vorhersage für den ZMP am genausten ist. Die Umsetzung eines solchen Modelles erfordert allerdings einen hohen Aufwand. Zudem sind die Berechnungen, die später auf dem Roboter durchgeführt werden müssen, aufwendiger als bei den anderen Modellen. Des Weiteren sind keine Veröffentlichungen einer Implementierung dieser Art für das *Nao*-Robotiksystem bekannt, sodass der Erfolg ungewiss ist. Dennoch könnte ein solches Modell den anderen überlegen sein. Die Inertialmatrizen für die einzelnen Glieder, die für ein solches Modell benötigt werden, sind in der Dokumentation des Roboters vorhanden.
Dieses Modell wird dennoch nicht verwendet, da zunächst versucht werden sollte mit möglichst einfachen Modellen ein gutes Ergebnis zu erzielen, um damit den Aufwand der Implementierung möglichst gering zu halten.

Das *Cart-Table-Modell* ist eines der meistbenutzten Modelle zur Implementierung von Laufalgorithmen. Es war also zunächst naheliegend auch für den *Nao* ein solches Modell zu verwenden. Für das Modell werden zunächst ZMP-Trajektorien erzeugt und daraus ein Schwerpunktsverlauf generiert. Bei Störungen muss dieser Verlauf neu berechnet werden. Das Problem besteht in der Erfassung von Störungen. Für eine Regelung wird ein gemessener ZMP benötigt. Dieser wird bei anderen Robotern mit im Fuß befindlichen Kraftsensoren bestimmt. Über solche Sensoren verfügt auch der *Nao*, allerdings unterliegen diese Sensoren so starken Schwankungen, dass

sie zur Bestimmung des ZMP nicht geeignet sind. Abbildung 8.1 zeigt eine Messung der Kraftsensorwerte. In der Abbildung ist zu erkennen, dass die Sensoren eine un-

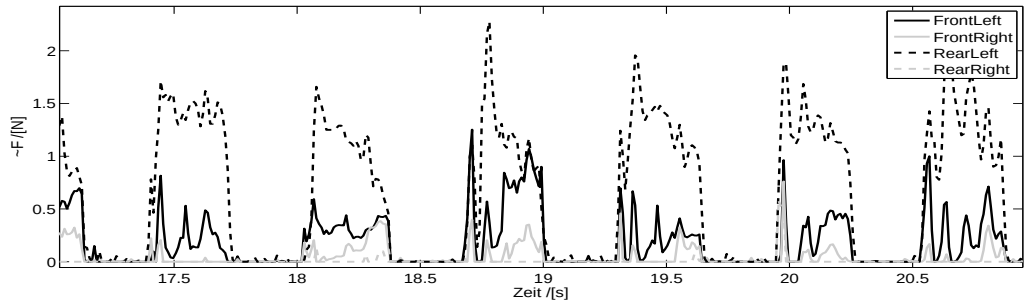


Abb. 8.1.: Zeitlicher Verlauf der Kraftsensorwerte während des Laufens

terschiedliche Sensibilität aufweisen. Der Sensor hinten rechts im Fuß (*RearRight*) reagiert kaum und gibt die meiste Zeit den Wert Null zurück. Die y-Achse stellt hierbei einen Wert dar, der proportional zur Kraft ist. Die genaue Skalierung hängt von der Kalibrierung der Sensoren ab. Der Hersteller gibt an, dass bei entsprechender Kalibrierung mit relativen Messfehlern von 20% zu rechnen ist. Sind die Sensoren nicht hinreichend genau kalibriert, so kann der Messfehler größer sein. Die Kalibrierung selbst kann allerdings nur vom Hersteller vorgenommen werden.
Die Messkurve zeigt ebenso, dass die Sensoren nicht ausreichend genau kalibriert sind, was keinen Einzelfall darstellt. Des Weiteren hat der *Nao* häufig Probleme beim Initialisieren der Füße, sodass teilweise gar keine Sensorwerte verfügbar sind. Eine Bestimmung des ZMP über die Kraftsensoren im Fuß ist daher nur bedingt möglich. Die Roboter, bei denen eine Messung des ZMP über Kraftsensoren im Fuß erfolgreich implementiert wurde (Abb. 5.9), weisen zumeist ein viel größeres Gewicht (8 bis 20 mal größer) auf, sodass eine Messung bei diesen Robotern unter Umständen einfacher ist.
Eine weitere Möglichkeit den ZMP zu bestimmen, besteht in der Verwendung der Beschleunigungssensoren im Torso des Roboters. Die Gleichung für den ZMP, die für das *Cart-Table-Modell* verwendet wird, wurde in Kapitel 5.3.1 bereits vorgestellt (siehe S. 43). Allerdings besagt die Gleichung, dass nicht die Torsobeschleunigung, sondern die Schwerpunktbeschleunigung benötigt wird. Diese ist allerdings nicht messbar. Des Weiteren werden die Beschleunigungskomponenten parallel zum Boden benötigt, da für das *Cart-Table-Modell* von einem mit dem Boden verklebten Supportfuß ausgegangen wird. Daher muss auch der Torsowinkel relativ zum Boden bekannt sein. Dieser kann zwar über die Beschleunigungssensoren und Gyroskope bestimmt werden, allerdings unterliegt auch dieser Wert Messschwankungen. In [Str09] wird beschrieben, dass es nicht möglich war, den ZMP mit Hilfe der Beschleunigungssensoren ausreichend genau zu bestimmen, um ihn als Sensorfeedback für eine Regelung verwenden zu können. Eine Umsetzung des *Cart-Table-Modells* für den *Nao* ist auch in [CKU10] vorgenommen worden. Die implementierte Regelung basiert auf einer Zustandsregelung, wobei die Zustände durch einen Beobachter

approximiert werden. Auch hier wird der ZMP über die Beschleunigungssensoren gemessen.
Ein Problem ist, dass zur Trajektorienerstellung, Zustandsbestimmung und ZMP-Messung stets die Gleichung des stark vereinfachten *Cart-Table-Modells* verwendet wird. Befindet sich der Supportfuß zu irgendeiner Zeit nicht mit der vollen Fläche auf dem Boden, sondern in einer Schräglage, so sind alle Berechnungen fehlerhaft. Dennoch ist es Czarnetzki et al. [CKU10] gelungen, einen Laufalgorithmus auf Basis dieses Modells erfolgreich zu implementieren.

Seitdem in der *Standard Platform League* des *RoboCup Nao*-Roboter genutzt werden, ist das Team der Universität Bremen (BHuman) ungeschlagener Weltmeister. In den aktuellsten Veröffentlichungen beschreiben sie die Nutzung des *3DLIPM* für ihre Laufalgorithmen. Des Weiteren schreiben die Autoren in [GR11], dass der von Czarnetzki et al. umgesetzte Laufalgorithmus auf Basis des *Cart-Table-Modells* in den Wettkämpfen wenig robust gegen Störungen zu sein schien. Die Roboter fielen häufig um. Dies ist allerdings nicht der ausschlaggebende Grund, warum auch in dieser Arbeit das *3DLIPM* verwendet wird.
Der große Vorteil des *3DLIPM* liegt darin, dass der ZMP nicht messtechnisch erfasst werden muss. Stattdessen basiert die Regelung auf einer Messung der aktuellen Schwerpunktposition. Diese kann messtechnisch einfacher erfasst werden als der ZMP und liefert bessere Ergebnis in der Genauigkeit. Dass das *3DLIPM*-Modell zudem den Kriterien für dynamische Stabilität genügt und somit den ZMP innerhalb der Supportfläche hält, wurde bereits in Abschnitt 5.3.2 gezeigt.

Für eine möglichst robuste Regelung, bei der Messwerte zu Verbesserung der Stabilität herangezogen werden können, scheint das *3DLIPM* für den *Nao* das geeignetste Modell zu sein.

8.2. Naospezifisches 3DLIPM

Im vorigen Abschnitt wurde erläutert, warum das *3DLIPM* für den Nao als Modell zur Schwerpunktvorhersage gewählt wurde. In Abschnitt 5.3.2 wurden die allgemeinen Modellansätze erläutert. In diesem Abschnitt soll nun auf die Besonderheiten bei der Umsetzung und Anpassung des Modells auf den *Nao* eingegangen werden.

Das umgesetzte Modell entspricht im Wesentlichen dem in [GR10] und [GR11] vorgestellten Verfahren. Zunächst wird versucht, den optimalen Zeitpunkt zu berechnen, um den Supportfuß zu wechseln. Es wurde gezeigt, dass hierdurch eine Stabilisierung der Lateralbewegung des Roboters erreicht werden kann. Hierfür wird in der referenzierten Literatur ein iteratives Verfahren genutzt, und damit begründet, dass das Problem nicht analytisch lösbar sei.
Das Problem des iterativen Verfahren ist allerdings, dass geeignete Startbedingungen für eine schnelle Konvergenz gewählt werden müssen. Daher wurden als Startbedingungen die berechneten Zeiten der vorherigen Phase herangezogen. Eine Berechnung mit diesem Verfahren ist möglich, solange das System keinen größeren Störungen

ausgesetzt ist. Beim Auftrete von größeren Störungen kann der Iterationsprozess allerdings an der Rechnergenauigkeit scheitern, was sowohl in der Simulation als auch auf dem Roboter festgestellt werden konnte.
Das Problem hierbei sind die hyperbolischen Funktionen, die zur Berechnung benötigt werden. Diese nehmen sehr schnell große Werte an. Dies kann dadurch verhindert werden, die Funktionen künstlich zu begrenzen.
Ein solches Verfahren garantiert zwar, dass das Problem nicht an der Rechnergenauigkeit scheitert, dennoch kann eine Konvergenz aber nicht garantiert werden.

Entgegen der Behauptung existiert aber eine analytische Lösung, die im Folgenden hergeleitet wird.

Gesucht werden die Größen t_e und $\bar{t}_b$. Die Bedingungen, die zum Supportwechsel gegeben sein müssen, sind durch die Gleichungen 5.11 und 5.12 gegeben. Einsetzen der Gleichungen 5.9 und 5.10 in diese Gleichungen ergibt:

$$x_{0,y} \cdot \cosh(k \cdot t_e) - \bar{x}_{0,y} \cdot \cosh(k \cdot \bar{t}_b) = \underbrace{\bar{r}_y + \bar{s}_y - r_y}_{a} \tag{8.1}$$

$$x_{0,y} \cdot k \cdot \sinh(k \cdot t_e) = \bar{x}_{0,y} \cdot k \cdot \sinh(k \cdot \bar{t}_b) \tag{8.2}$$

Quadrieren der Formeln ergibt:

$$x_{0,y}^2 \cdot \cosh^2(k \cdot t_e) = a^2 + \bar{x}_{0,y}^2 \cdot \cosh^2(k \cdot \bar{t}_b) + 2 \cdot a \cdot \bar{x}_{0,y} \cdot \cosh(k \cdot \bar{t}_b) \tag{8.3}$$

$$x_{0,y}^2 \cdot \sinh^2(k \cdot t_e) = \bar{x}_{0,y}^2 \cdot \sinh^2(k \cdot \bar{t}_b) \tag{8.4}$$

Differenz (8.3) - (8.4)

$$x_{0,y}^2 \cdot \underbrace{(\cosh^2(k \cdot t_e) - \sinh^2(k \cdot t_e))}_{=1} = a^2 + \bar{x}_{0,y} \cdot \underbrace{(\cosh^2(k \cdot \bar{t}_b) - \sinh^2(k \cdot \bar{t}_b))}_{=1} + 2 \cdot a \cdot \bar{x}_{0,y} \cdot \cosh(k \cdot \bar{t}_b) \tag{8.5}$$

$\Rightarrow$

$$x_{0,y}^2 = a^2 + \bar{x}_{0,y}^2 + 2 \cdot a \cdot \bar{x}_{0,y} \cdot \cosh(k \cdot \bar{t}_b) \tag{8.6}$$

Auflösen von Gleichung (8.6) nach $\bar{t}_b$ unter Beachtung der Bedingung $\bar{t}_b < 0$:

$$\bar{t}_b = -\frac{1}{k} \cdot \operatorname{acosh}\left(\frac{x_{0,y}^2 - a^2 - \bar{x}_{0,y}^2}{2 \cdot a \cdot \bar{x}_{0,y}}\right) \tag{8.7}$$

Einsetzen in (5.12) ergibt für t_e:

$$t_e = \frac{1}{k} \cdot \operatorname{asinh}\left(\frac{\bar{x}_{0,y} \cdot \sinh(k \cdot \bar{t}_b)}{x_{0,y}}\right) \tag{8.8}$$

Die analytische Berechnung von t_e und $\bar{t}_b$ hat entscheidende Vorteile gegenüber der iterativen Methode. Zum einen bestehen die Probleme der Konvergenz nicht, zum anderen wird weniger Rechenzeit benötigt.

Eine weitere Änderung, die vorgenommen wurde, ist die Begrenzung der Pendelbewegungsvorhersage in y-Richtung. Da die Strategie des Laufalgorithmus darin liegt, den Schwerpunkt des Roboters so zu bewegen, dass er der Pendelvorhersage folgt, kann dies zu Situationen führen, die zum Sturz des Roboters führen. Während des Laufens schwingt der Schwerpunkt zunächst in Richtung Supportfuß, ändert dann zum Zeitpunkt $t = 0$ seine Richtung und bewegt sich vom Schwerpunkt weg. Kommt es nun zu einer Vorhersage, dass der Schwerpunkt über den Pendelursprung schwingt, so entsteht keine Richtungsänderung. Das Modell wird eine Vorhersage treffen, die dazu führt, dass der Roboter seitlich umkippt. Abbildung 8.2 verdeutlicht die auftretenden Situationen. Abbildung 8.2(a) zeigt eine erwartete Pendelschwingung, auf

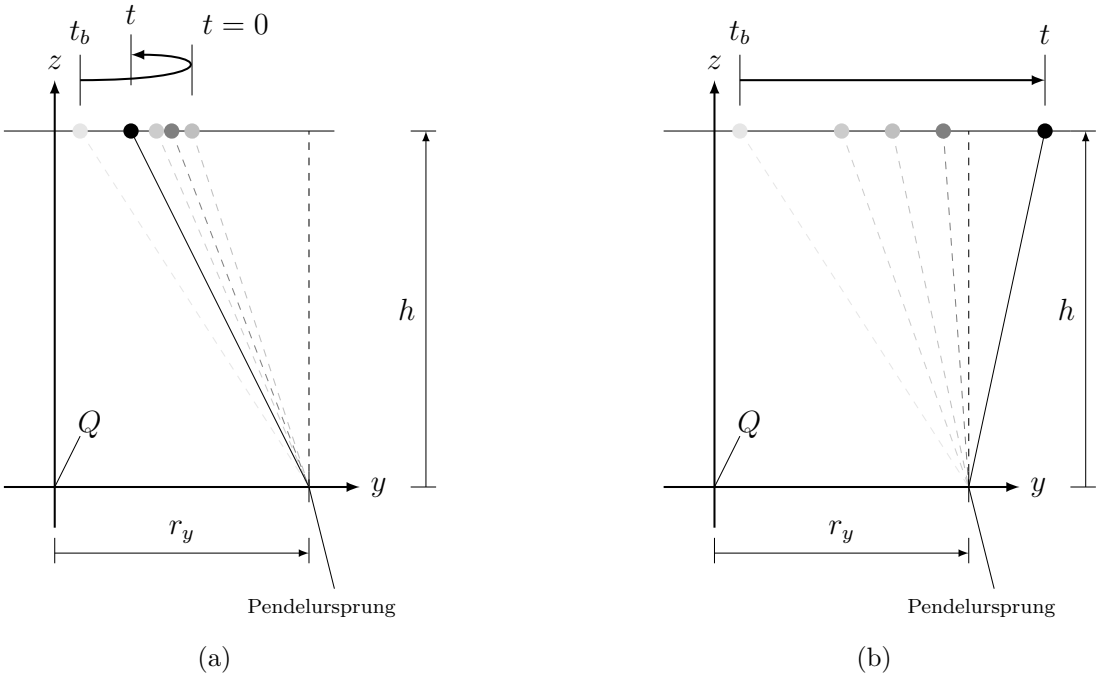


Abb. 8.2.: Mögliche Pendelbewegungen in y-Richtung. (a) Erwartete Schwingung; (b) Überschwingen

deren Basis eine Berechnung der Endzeit und somit eines Supportwechsels erfolgen kann. Schwingt das Pendel allerdings wie in Abbildung 8.2(b) über den Ursprung hinaus, so kann keine Lösung für t_e gefunden werden. Eine solche Situation ist unbedingt zu vermeiden, weshalb eine Begrenzung für die Distanz zwischen Pendelursprung und Schwerpunkt implementiert werden muss.
Diese Begrenzung sorgt dafür, dass der Roboter seinen Schwerpunkt nicht in einen unzulässigen Bereich verlagert. Sind externe Störungen so groß, dass sich der Schwerpunkt dennoch über den Pendelursprung hinaus bewegt, so ist eine Stabilisierung des Roboters auf Grundlage des Modelles ohnehin nicht mehr möglich.

Messung des Schwerpunktes

Wie in Abschnitt 5.3.2 bereits erwähnt, muss für die Stabilisierung des Ganges das Modell adaptiv angepasst werden. Hierzu werden gefilterte Messwerte des Schwerpunktes genutzt. Bisher wurde noch nicht erläutert, wie der Schwerpunkt des Roboters messtechnisch erfasst werden kann. Dies soll im Folgenden geschehen.

Die Messung des Roboterschwerpunktes im körpereigenen Koordinatensystem erfolgt über die Gelenkstellung des Roboters mit Hilfe der Vorwärtskinematik. Allerdings gibt das Modell keine Vorhersage des Schwerpunktes in diesem Koordinatensystem, sondern im Koordinatensystem, das parallel zum Boden in konstanter Pendelhöhe h liegt. Die Projektion des Schwerpunktes auf die Bodenebene entspricht somit den x- und y-Komponenten des Schwerpunktes in diesem Koordinatensystem. Um nun die Modellvorhersage mit dem tatsächlichen Schwerpunkt vergleichen zu können, ist die Projektion des Schwerpunktes in die Bodenebene notwendig. Hierzu muss die Torsoorientierung relativ zum Boden ermittelt werden.

Zunächst wird angenommen, dass der Supportfuß vollständig auf dem Boden aufliegt. Somit liegt dessen Koordinatensystem parallel zum Boden. Die Orientierung des Torsos relativ zum Fuß lässt sich durch Vorwärtskinematik bestimmen. Die Information der Verdrehung liegt in Form einer Transformationsmatrix vor, aus der die Drehwinkel des Torsos bestimmt werden können.
Da diese Art der Winkelbestimmung von einem parallel zum Boden orientierten Fuß ausgehen, können Störungen auf diese Weise nicht erfasst werden. Daher wird der Winkel zusätzlich aus den Messwerten der Gyroskope bestimmt.
Aus beiden Informationen wird mit Hilfe eines Kalmanfilters [WB06] eine resultierende Torsoorientierung berechnet. Diese Größe dient nun dazu, den tatsächlichen Roboterschwerpunkt in die Bodenebene zu projizieren (Abb. 8.3). Die Lage des Roboterschwerpunktes im Fußkoordinatensystem wird durch Vorwärtskinematik bestimmt. Die Transformation in das Fußkoordinatensystem entspricht einer Drehung um den Winkel α, der auf die zuvor beschriebene Weise ermittelt werden kann.

$$x_{m,y} = \cos(\alpha) \cdot CoM_{y'} - \sin(\alpha) \cdot CoM_{z'} \tag{8.9}$$

$CoM_{y'}$, $CoM_{z'}$ = Schwerpunktkoordinaten im Fußkoordinatensystem

Somit ist ein Messwert der Schwerpunktsprojektion im Koordinatensystem des Fußes vorhanden. Da der Ursprung des Fußkoordinatensystem des Supportfußes ebenfalls den Ursprung des Pendels darstellt, können Modellvorhersage und Messwert verglichen werden. Die Berechnungen für die Schwerpunktprojektion in x-Richtung verläuft nach demselben Prinzip.

Für die Aktualisierung der Pendelparameter wird ein gefilterter Wert aus Modellvorhersage und Messung genutzt. Die Filterung erfolgt ebenso wie die Winkelfilterung durch ein Kalmanfilter. Die Kovarianzmatrizen Q und R des Filters können dazu genutzt werden, die Gewichtung der einzelnen Werte für den gefilterten Wert zu

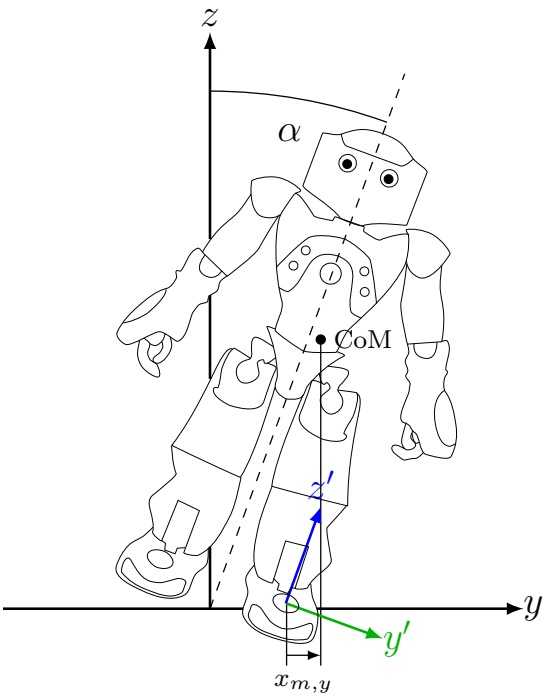


Abb. 8.3.: Projektion des Roboterschwerpunktes in die Bodenebene

konfigurieren. Somit kann eingestellt werden, in wie weit dem Modell und in wie weit den Messwerten vertraut wird.

Der Filteralgorithmus wird im Folgenden erläutert, da als Messgröße lediglich die Schwerpunktposition erfasst wird, jedoch sowohl Position als auch Geschwindigkeit als gefilterte Werte benötigt werden. Zunächst wird die Differenz zwischen der Modellvorhersage x_{e_i} und der Messung x_{m_i} zur Zeit t_i ermittelt:

$$x_{e_i} = r + x(t_i) \tag{8.10}$$

$$\Delta x_i = x_{m_i} - x_{e_i} \tag{8.11}$$

Für das Kalmanfilter wird ein Zustandsvektor

$$\mu_i' = \begin{bmatrix} x_{e_i} \\ \dot{x}(t_i) \end{bmatrix}$$

definiert. Der Filterprozess des Kalmanfilters gestaltet sich folgendermaßen:

Vorhersage der Fehlerkovarianzmatrix

$$P_i' = A \cdot P_{i-1} \cdot A^T + Q \tag{8.12}$$

Berechnung der Kalmanverstärkung

$$K_i = P_i' \cdot C^T \cdot (C \cdot P_i' \cdot C^T + R)^{-1} \tag{8.13}$$

Aktualisierung des Zustandsvektors

$$\mu_i = \mu'_i + K_i \cdot \Delta x_i \tag{8.14}$$

Aktualisierung der Fehlerkovarianzmatrix

$$P_i = P'_i - K_i \cdot C \cdot P'_i \tag{8.15}$$

Hierbei gilt:

$$A = \begin{bmatrix} 1 & \Delta t_i \\ 0 & 1 \end{bmatrix},\ C = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

$$Q = \text{Prozessrauschkovarianz}$$

$$R = \text{Messrauschkovarianz}$$

Neuberechnung der Pendelparameter

Aus den gefilterten Messwerten können nun die Pendelparameter neu berechnet werden. Die Vorgehensweise ist ebenfalls in [GR11] beschrieben. Zunächst wird angenommen, dass die aus Modellvorhersage und Messung gefilterten Werte, den tatsächlichen, aktuellen Zustand des Pendels beschreiben. Es gilt:

$$r' + x'(t'_i) = x_{f_i} \tag{8.16}$$

$$\dot{x}'(t'_i) = \dot{x}_{f_i} \tag{8.17}$$

Hierbei sind x_{f_i} die gefilterte Schwerpunktsprojektion in Bodenebene und $\dot{x}_{f_i}$ die dazugehörige gefilterte Geschwindigkeit. Die gestrichenen Größen stellen hierbei die aktualisierten Parameter und Funktionen dar.

Für die Pendelbewegung in y-Richtung müssen die Größen $x'_{0,y}$, t'_i, t'_e und $\bar{t}'_b$ berechnet werden. Die Berechnung der Start- und Endzeiten einer Pendelphase t'_e und $\bar{t}'_b$ ist lediglich die Kenntnis von $x'_{0,y}$ erforderlich. Anschließend können sie durch die Gleichungen 8.7 und 8.8 berechnet werden. Zum Zeitpunkt t_i gilt:

$$x_{f_i,y} = x'_{0,y} \cdot \cosh(k \cdot t'_i) \tag{8.18}$$

$$\dot{x}_{f_i,y} = x'_{0,y} \cdot k \cdot \sinh(k \cdot t'_i) \tag{8.19}$$

Hieraus folgt:

$$x'_{0,y} = \sqrt{x^2_{f_i,y} - \frac{\dot{x}^2_{f_i,y}}{k^2}} \tag{8.20}$$

$$t'_i = \frac{1}{k} \cdot \operatorname{arcsinh}\left(\frac{\dot{x}_{f_i,y}}{k \cdot x'_{0,y}}\right) \tag{8.21}$$

Bei der Berechnung von $x'_{0,y}$ ist darauf zu achten, dass der Wurzelterm nicht negativ wird. Ist dies der Fall, so ist eine Berechnung nicht möglich. Physikalisch bedeutet dies, dass das Pendel über den Pendelursprung hinwegschwingen wird. Dies wurde aus den zuvor genannten Gründen schon für die Pendelvorhersage vermieden. Ebenso ist es wichtig, dass auch bei dieser Berechnung ein solches Ergebnis vermieden wird. Es ist also notwendig eine Begrenzung für ein berechnetes $x_{0,y}$ festzusetzen.

Die Pendelparameter für die x-Richtung können durch das Gleichungssystem 5.15 berechnet werden. Hierbei wird ebenfalls ein neuer Pendelursprung berechnet. Liegt dieser außerhalb eines zuvor definierten Bereiches, so erfolgt, wie in Abschnitt 5.3.2 beschrieben, eine Neuberechnung der Schrittweite. Somit kann auf Störungen in x-Richtung reagiert werden.

Totzeit

Eine weitere Eigenschaft, die unbedingt beachtet werden muss, ist die Totzeit, die zwischen dem Senden eines Befehls und der Reaktion des Befehls in den Motoren besteht. Diese Totzeit ist in [GR11] mit 40 ms angegeben und konnte durch eigene Messungen bestätigt werden. Aufgrund der Totzeit macht es keinen Sinn, die Füße des Roboters relativ zur aktuellen Pendelvorhersage zu setzen. Stattdessen wird das durch die gefilterten Werte aktualisierte Modell dazu genutzt, eine Vorhersage zu treffen, wo sich der Schwerpunkt in 40 ms befinden wird. Auf Basis dieser Vorhersage werden die Fußpositionen berechnet.
Eine Besonderheit der Vorhersage besteht zum Zeitpunkt eines Pendelwechsels. Da das Modell phasenweise arbeitet, und je nach Phase einen unterschiedlichen Ursprung und unterschiedliche Pendelparameter aufweist, ist zum Übergangszeitpunkt kein passendes Modell vorhanden. Es muss also versucht werden, eine möglichst passende Vorhersage für die Position in der nächsten Pendelphase treffen zu können.
Für den betreffenden Zeitraum wird in [GR11] folgende Berechnung vorgeschlagen:

$$x_{e_{i+40ms}} = \bar{s} + \bar{r} + \bar{x}(\bar{t}'_b + t'_i + 40\,\text{ms} - t'_e) \quad \text{für } t'_i + 40\,\text{ms} > t'_e \tag{8.22}$$

Hierzu müssen allerdings die Pendelparameter für die nächste Phase bekannt sein, um $\bar{x}(t)$ berechnen zu können. Tatsächlich sind alle benötigten Parameter bekannt. $\bar{x}_{0,y}$ ist eine definierte Größe und $\bar{x}_{0,x}$ und $\bar{\dot{x}}_{0,x}$ gehen aus dem Gleichungssystem 5.15 hervor. Somit können auch Vorhersagen für die nächste Phase getroffen werden.

Laufgeschwindigkeit

Die Laufgeschwindigkeit des Roboters wird bei diesem Verfahren bislang einzig über eine vorgegeben Schrittweite definiert. Allerdings wird diese Schrittweite auch nicht zwingend umgesetzt, da stets eine Überprüfung erfolgt, ob der ZMP und somit der Pendelursprung innerhalb der Supportfläche gehalten werden kann. Ist dies nicht der Fall, so wird die Schrittweite entsprechend angepasst. Es ist allerdings wünschenswert, den Roboter dazu zu bewegen, einer vordefinierten Geschwindigkeit anzunehmen. Für eine Geschwindigkeit kann, durch Kenntnis der zeitlichen Länge einer Phase, die passende Schrittweite berechnet werden. Um nun Einfluss auf die Geschwindigkeit des Roboters zu nehmen, wird, nachdem die Fußpositionen berechnet sind, ein zusätzlicher Winkel für die *HipPitch*-Gelenke angelegt. Dieser Winkel führt zu einer Neigung des Torsos um die y-Achse. Somit findet eine Gewichtsverlagerung statt. Ist der Torso nach vorne geneigt, so wird der Roboter stärker nach vorne kippen als das Pendel dies vorhersagt. Durch die adaptive Anpassung der Parameter wird hierauf Rücksicht genommen und es wird eine neue, größere Schrittweite berechnet, um einen Sturz zu verhindern. Dieses Prinzip wird ebenfalls beim *Segway* verwendet, einem einachsigen Fahrzeug der gleichnamigen Firma. Dieses Fahrzeug balanciert sich permanent aus. Ein Beschleunigen und Bremsen des Fahrzeuges ist über eine Gewichtsverlagerung über den Fahrer möglich.
Für den Roboter wird, um eine bestimmte Geschwindigkeit zu erreichen, die mittlere Geschwindigkeit über eine Pendelphase ermittelt und die Differenz zur Sollgeschwindigkeit berechnet. Eine PI-Regelung sorgt für die passende Einstellung des Torsowinkels. Passende Reglerparameter wurden hierbei experimentell ermittelt.

Besonderheiten

Bevor im nächsten Abschnitt eine Evaluierung des Modelles stattfindet, soll noch kurz auf Besonderheiten bei der Implementierung des Laufalgorithmus eingegangen werden.
Das Versetzen der Füße basiert auf dem Pendelmodell. Die Position der Supportfußes ist immer durch die Modellvorhersage relativ zum Schwerpunkt gegeben. Für den anderen Fuß stehen nur Start- und Endposition fest. Während einer Phase muss nun für diesen Fuß ebenfalls eine x-, y- und z-Position ermittelt werden. Die y-Position ist durch den Hüftabstand und die Position des Supportfuß gegeben. Die z-Position ergibt sich aus einer zuvor definierten Schritthöhe, die diese maximal annehmen wird. Dazwischen folgt die z-Position einem sinusförmigen Verlauf. Die x-Position muss aus der aktuellen Pendelzeit und der Schrittweite abgeleitet werden. Dazwischen wird ein linearer Verlauf dieser Position angenommen. Dieser Verlauf kann allerdings selbst bei kleinen Störungen dazu führen, dass der Roboter sich selbst ins Schwanken bringt. Der Grund hierfür ist, dass bei einem linearen Verlauf für die x-Position beim Supportwechsel noch eine Bewegung des Fußes in x-Richtung vorhanden ist. Kommt dieser Fuß nun früher mit dem Boden in Berührung als erwartet, so sorgt diese Bewegung für einen Impuls in negative x-Richtung. Dieser ist, so

lange der Fuß parallel zum Boden orientiert ist, durch die Reibungskräfte zwischen Boden und Fuß bedingt. Bei größeren Störungen, die eine Neigung des Roboters verursachen, tritt der Roboter hierbei mit der Fußspitze gegen den Boden. Der folgende Impuls ist dementsprechend größer. Um dies zu vermeiden, ist es sinnvoll, die Zielposition des Fußes noch vor Wechsel der Supportphase zu erreichen und diese Position bis zum Wechsel zu halten.

Eine weitere Besonderheit bei der Implementierung ist, dass bei Berechnung der Beingelenkwinkel relativ zum Schwerpunkt des Roboters bisher nicht berücksichtigt wird, dass nach Anwendung dieser Winkel, der Roboterschwerpunkt innerhalb des Körpers verändert wird. Es geht hierbei im Folgenden nicht um die projizierte Schwerpunkposition, sondern um die Position des Schwerpunktes im Torsokoordinatensystem. Zunächst kann man davon ausgehen, dass der Fehler zwischen aktueller Schwerpunktposition und der, nach 40 ms durch die Befehle folgenden Schwerpunktposition, klein ist und dies nicht zu berücksichtigen ist. Tatsächlich finden auch keine großen Änderungen der Schwerpunktposition statt.
Um dies dennoch berücksichtigen zu können, werden zunächst für die aktuelle Schwerpunktposition, die aus den Gelenkstellungen bekannt ist, die Gelenkwinkel berechnet. Anschließend kann durch Vorwärtskinematik der daraus folgende Schwerpunkt berechnet werden. Der Fehler zwischen der Sollposition, die durch das Pendel gegeben ist, und der berechneten Positionen wird für eine Aktualisierung der aktuellen Schwerpunktposition genutzt, auf deren Grundlage die Berechnung erneut durchgeführt wird. Dieses Vorgehen kann solange wiederholt werden, bis der Fehler zwischen Sollposition und berechneter Position ausreichend klein ist. In der Regel wird bei dreimaliger Iteration ein Fehler erreicht, der kleiner als 1 mm in allen Achsen ist.
Der Grund, warum dieses Vorgehen unbedingt genutzt werden sollte, ist nicht der, dass der Roboterschwerpunkt möglichst exakt dem Pendel folgen muss, sondern der, dass durch die iterative Berechnung ein errechneter Schwerpunktwert vorliegt, auf dessen Basis die Gelenkwinkel berechnet werden. Im Gegensatz hierzu ist der aus den Gelenkstellungen berechnete aktuelle Schwerpunkt, ein Wert, der auf Messwerten beruht und Schwankungen unterliegt.
Die Berechnung der Gelenkwinkel auf Basis eines von Messwerten unabhängig berechneten Schwerpunktes konnte die Laufstabilität des Roboters deutlich erhöhen.

Der Begriff *Stiffness* (Steifheit) wurde an einigen Stellen dieser Arbeit bereits erwähnt, allerdings noch nicht erklärt. Die *Stiffness* ist ein Kontrollparameter für die Motoren, der Werte zwischen null und eins annehmen kann. Je größer dieser Wert ist, desto aktiver versucht der Motor die vorgegebene Stellung zu erreichen oder zu halten. Eine höhere *Stiffness* ist allerdings auch mit einem höheren Stromverbrauch der Motoren verbunden. Es muss also ein geeigneter Wert für diesen Parameter gefunden werden, der eine ausreichende Agilität der Motoren gewährleistet, aber dennoch keine zu hohen Motorströme verursacht. Vor allem in den Kniegelenken des Roboters ist ein erhöhter Stromverbrauch festzustellen. Dieser wird zudem erhöht, je tiefer der Roboterschwerpunkt und somit die Pendelhöhe gewählt wird. In diesem Fall ist die Belastung der Knie enorm hoch. Für eine Entlastung muss daher

ein ausreichend hoher Schwerpunkt gewählt werden, sodass die Beine des Roboters annähernd gestreckt sein können. Ein hoher Schwerpunkt verursacht allerdings größere Pendelbewegungen und macht das System anfälliger gegen Störungen. Allerdings können mit hohem Schwerpunkt und gestreckteren Beinen auch größere Schrittweiten erreicht werden. Es sind also viele Faktoren bei der Anpassung des Modells zu berücksichtigen.

8.3. Evaluierung

In den vorigen Abschnitten wurden immer wieder die Vorzüge des *3DLIPM* und verschiedene Strategien zur Verbesserung des Systems vorgestellt. In diesem Abschnitt soll das Modell evaluiert werden. Dies wird anhand von Messungen geschehen.

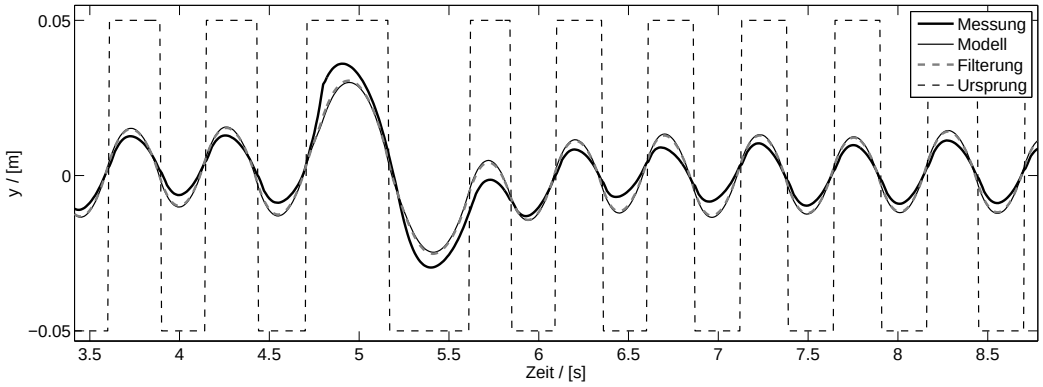


Abb. 8.4.: Pendelbewegung in y-Richtung durch Simulation

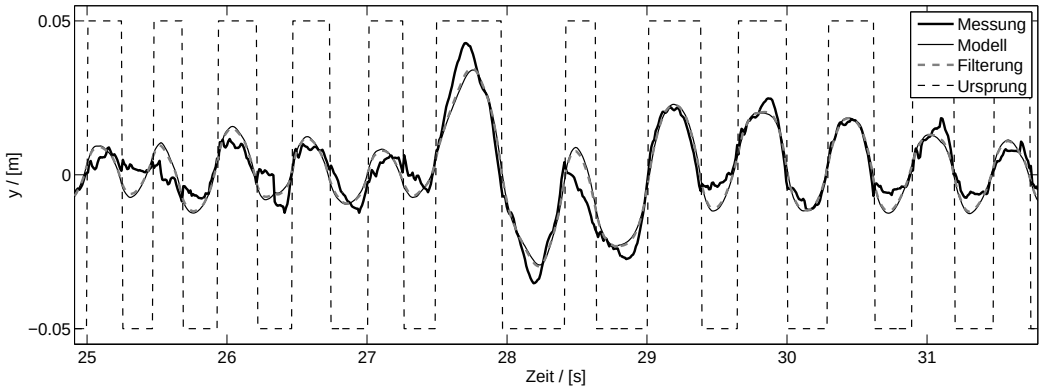


Abb. 8.5.: Pendelbewegung in y-Richtung für den realen Roboter

Zunächst wird gezeigt, dass der Roboter durch Anpassung der Pendelzeit und der

Phasenlänge, Störungen in y-Richtung ausgleichen kann. In den Abbildungen 8.4 und 8.5 sind die Bodenprojektionen des Schwerpunktes sowie die Pendelursprungsposition für eine Simulation und für den echten Roboter aufgetragen. In beiden

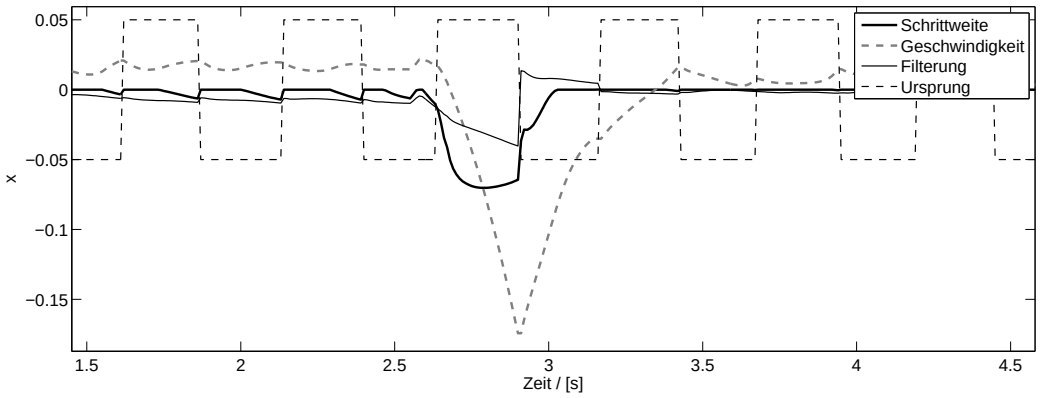


Abb. 8.6.: Pendelbewegung in x-Richtung durch Simulation

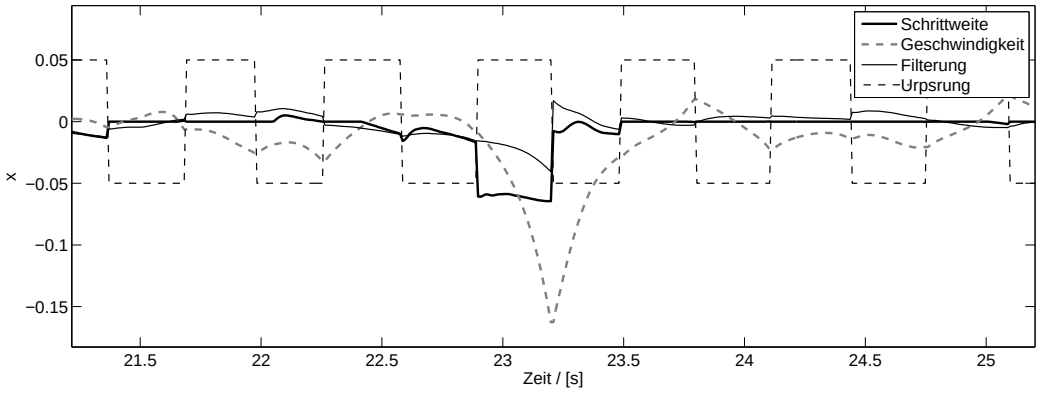


Abb. 8.7.: Pendelbewegung in x-Richtung für den realen Roboter

Fällen wurde der Roboter durch ein seitliches Stoßen in der Pendelbewegung gestört. Der Pendelursprung wechselt zum Ende einer Pendelphase vom einen auf den anderen Fuß. Er verdeutlicht in den Abbildungen die zeitliche Dauer einer Pendelphase. Nach dem Stoß (in der Simulation zum Zeitpunkt 5 s und im Realfall bei ca. 27.5 s) des Roboters ist eine deutliche Verlängerung der Pendelphase zu erkennen. Hierdurch gleicht der Roboter die Störungen aus. Anschließend ist ein schnelles Einschwingen in die normale Pendelbewegung zu erkennen. Es ist also gezeigt, dass eine Veränderung der Pendelphasendauer dazu genutzt werden kann, den Roboter zu stabilisieren. Eine solche Veränderung ist bei dem Regelansatz über das *Cart-Table-Modell* nicht vorgesehen. Hier wird aktiv versucht gegen die Veränderung anzugehen

und sie klein zu halten. Je nach Art und Stärke der Störung können beide Methoden Vor- und Nachteile haben. Der Vorteil der hier vorgestellten Lösung ist, dass der Roboter die Störung zunächst zulässt und sanft abfängt. Hierdurch entstehen keine schnellen Reaktionsbewegungen, die die Stabilität des Roboters gefährden könnten.

Eine Stabilisierung in y-Richtung kann durch Veränderung der Pendelzeit erreicht werden. Für die Stabilisierung in x-Richtung wird auf eine adaptive Anpassung der Schrittweite gesetzt. Schwingt der Roboter bei einer Vorwärtsbewegung weiter nach vorne als erwartet, so wird die Schrittweite erhöht. Analog dazu wird die Schrittweite verkürzt, wenn der Roboter weniger weit nach vorne schwingt als erwartet. Die Abbildungen 8.6 und 8.7 zeigen eine Messung für den Fall, dass der Roboter auf der Stelle gehen soll. Die Schwerpunktgeschwindigkeit in x-Richtung und die Schrittweite sollen dementsprechend null sein. Während der Bewegung erfährt der Roboter einen Stoß von vorne, der den Roboter dazu veranlasst einen Schritt nach hinten zu machen, um den Stoß abzufangen. In der Grafik sind neben der Schwerpunktgeschwindigkeit und der Schrittweite auch die gefilterte Schwerpunktposition und die Pendelursprungsposition in y-Richtung aufgetragen. Der Ursprung ist aufgetragen, um die einzelnen Phasen zu kennzeichnen, hat aber keinen Einfluss auf die Bewegung in x-Richtung. Die gefilterte Schwerpunktsposition ist jeweils relativ zum Supportfuß aufgetragen, weshalb beim Supportwechsel Sprünge entstehen können. Die Höhe der Sprünge ist von der Schrittweite abhängig. Für die Ordinate ist keine Einheit angegeben, da die einzelnen Messkurven verschiedene Einheiten aufweisen. Es gelten die jeweiligen SI-Einheiten, sodass die Positionen und die Schrittweite in [m] und die Geschwindigkeit in [m/s] angegeben sind.
Zum Zeitpunkt des Stoßes ist eine starke Erhöhung der Geschwindigkeit in negative x-Richtung zu erkennen. Der Roboter reagiert hierauf mit einer negativen Schrittweite. Zur nächsten Pendelphase springt die Schwerpunktposition auf einen positiven Wert. Dieser Sprung resultiert daraus, dass der Roboter einen Schritt nach hinten macht und somit seinen Fuß hinter den Schwerpunkt bewegt. Ausgehend von dieser Position befindet sich der Schwerpunkt in der nächsten Phase im positiven Bereich.

Im vorigen Abschnitt wurde erläutert, dass durch Gewichtsverlagerung eine bestimmte Laufgeschwindigkeit erreicht werden soll. In den Abbildungen 8.8 und 8.9 sind Geschwindigkeit, Schrittweite, gefilterte Position und die zusätzliche Torsoneigung eingezeichnet, die die Gewichtsverlagerung hervorruft. Es ist zu erkennen, dass eine zusätzliche Neigung nach vorne entsteht, um der Geschwindigkeitsvorgabe von $0,2\,\mathrm{m/s}$ zu folgen. Für die Messungen wurde eine Pendelhöhe von 265 mm gewählt, was im Vergleich zu anderen Implementierungen relativ hoch ist.

In der Simulation konnten mit diesem Algorithmus Geschwindigkeiten von bis zu $37\,\mathrm{cm/s}$ erreicht werden. Bei Tests auf dem realen Roboter konnten Geschwindigkeiten von bis zu $25\,\mathrm{cm/s}$ gemessen werden. Allerdings sinkt die Stabilität des Laufalgorithmus bei hohen Geschwindigkeiten. Das Weltermeisterteam aus Bremen gibt Geschwindigkeiten von bis zu 31 cm/s für ihre Roboter an, schreibt in [GR11] aber auch, dass der Gang bei niedrigeren Geschwindigkeiten robuster gegen Störungen ist.

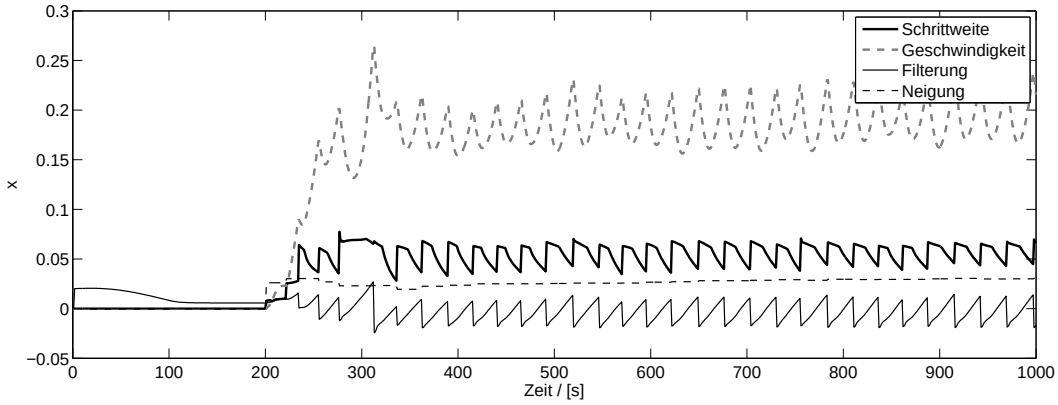


Abb. 8.8.: Geschwindigkeitsangleichung an Vorgabe durch zusätzliche Torsoneigung durch Simulation

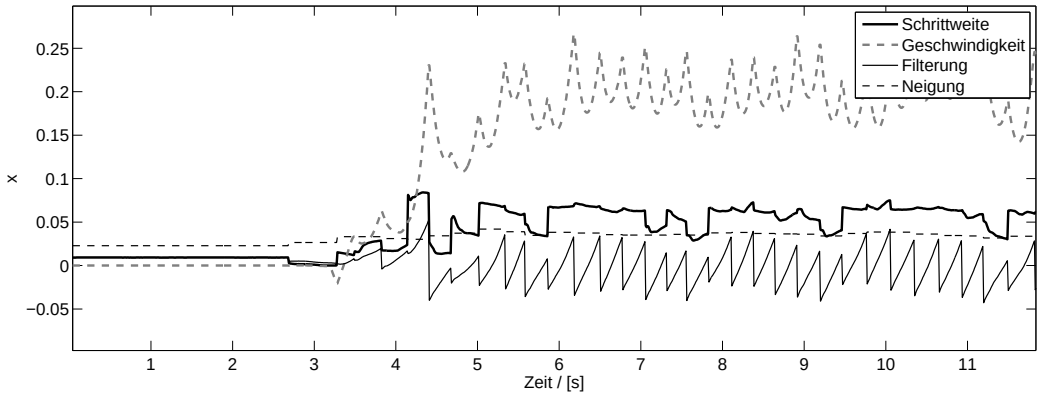


Abb. 8.9.: Geschwindigkeitsangleichung an Vorgabe durch zusätzliche Torsoneigung für den realen Roboter

Parameter, die die Stabilität des Laufalgorithmus beeinflussen, sind die Pendelhöhe, die Kovarianzmatrizen der Kalmanfilter, der Bereich für den zulässigen Pendelursprung innerhalb des Fußes sowie die Vorgabe der lateralen Pendelschwungweite. Die für die Versuche eingestellten Parameter wurden experimentell bestimmt.

8.4. Alternative Neuberechnung der Pendelparameter

In den vorherigen Abschnitten wurden kleine Änderungen bezüglich des in [GR10] vorgestellten Laufalgorithmus implementiert, um diesen zu optimieren. Während dieser Implementierungen konnte beobachtet werden, dass die Modellvorhersage für

die Geschwindigkeit des Pendels teilweise große Abweichungen zur tatsächlichen Geschwindigkeit aufwies. Hierzu wurde die tatsächliche Geschwindigkeit durch Differenzierung der Pendelposition erfasst. Abbildung 8.10 zeigt die Geschwindigkeitsvorhersage in x-Richtung durch das Pendel im Vergleich zur tatsächlichen gemessenen Geschwindigkeit. Die Messung wurde hierbei im Simulator durchgeführt. Es ist

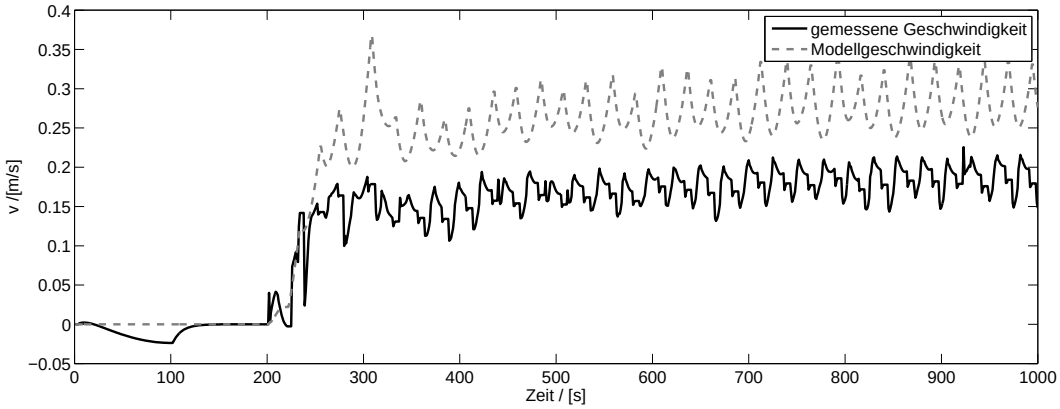


Abb. 8.10.: Vergleich der Geschwindigkeitsvorhersage des Pendelmodells mit der gemessenen Geschwindigkeit

ein klare Abweichung der Pendelvorhersage zur Messung zu erkennen. Das Modell sagt in den meisten Fällen eine deutlich höhere Geschwindigkeit voraus, als tatsächlich vorhanden ist. Dies macht sich vor allem bei Störungen bemerkbar, bei denen die Modellvorhersage sehr sensibel reagiert. Dies hat zur Folge, dass der Roboter stärker auf Störungen reagiert, als dies notwendig wäre. Das Problem der Modellgeschwindigkeit ist, dass diese Größe nicht direkt durch Messungen korrigiert wird. Bei der Filterung der Messwerte wird lediglich die Differenz zwischen gemessener Schwerpunktposition und vorhergesagter Schwerpunktposition als Aktualisierungsgröße verwendet und ebenso zur Korrektur der Position wie für die Geschwindigkeit genutzt. Aus diesen korrigierten Größen wird dann wiederum das Pendel neu berechnet, um eine Positions- und Geschwindigkeitsvorhersage zu treffen. Wie gut die Geschwindigkeitsvorhersage tatsächlich ist, wird an keiner Stelle überprüft.

Im Folgenden wird aus diesen Beobachtungen ein neues Verfahren zur Neuberechnung der Pendelparameter für die x-Komponente hergeleitet, das aus der aktuellen Pendelposition und der einen Zeitschritt zurückliegenden Position die Parameter neu berechnet. Die Parameter für die y-Komponente werden nach dem zuvor vorgestellten Verfahren berechnet. Die Gleichung 5.7 wird hierzu für beide Positionen aufgestellt, wobei als Positionen die gefilterten Werte genutzt werden.

$$x(t)_{f,x} = x_{0,x} \cdot \cosh(k \cdot t) + \dot{x}_{0,x} \cdot \frac{1}{k} \cdot \sinh(k \cdot t) \tag{8.23}$$

$$x(t_{-1})_{f,x} = x_{0,x} \cdot \cosh(k \cdot t_{-1}) + \dot{x}_{0,x} \cdot \frac{1}{k} \cdot \sinh(k \cdot t_{-1}) \tag{8.24}$$

Hierbei gibt t_{-1} den Zeitpunkt des vorigen Prozessschrittes an. Aus diesen beiden Gleichungen können die unbekannten Größen x_0 und $\dot{x}_0$ berechnet werden. Die Änderung der Position vom letzten zum aktuellen Prozessschritt wird in den Gleichungen berücksichtigt. Diese Änderung entspricht der Schwerpunktgeschwindigkeit.

Der Pendelursprung, der bei der anderen Parameterberechnungsart ebenfalls variiert werden konnte, wird bei diesem Ansatz als null angenommen. Somit wird ein Pendelmodell genutzt, das den Pendelursprung immer an der gleichen Position hält. Ein solches Verhalten ist auch bei dem anderen Ansatz möglich. Hier konnte ein zulässiger Bereich für den Ursprung definiert werden. Dieser Bereich ist nun auf einen Punkt beschränkt.

Mit den Pendelparametern, die nun auch die tatsächliche Geschwindigkeit des Pendels berücksichtigen, muss nun eine Schrittweite berechnet werden, die zu einem stabilen Gang führt. Hierfür kann die Bedingung gesetzt werden, dass für die nächste Pendelphase zum Zeitpunkt $t = 0$ wie im vorigen Ansatz auch $\bar{x}_{0,x} = 0$ gilt.

Die passende Schrittweite kann jetzt auf folgende Weise berechnet werden:

$$\dot{x}(t_e)_x = x_{0,x} \cdot k \cdot \sinh(k \cdot t_e) + \dot{x}_{0,x} \cdot \cosh(k \cdot t_e) \tag{8.25}$$

$$\bar{\dot{x}}(\bar{t}_b)_x = \bar{\dot{x}}_{0,x} \cdot \cosh(k \cdot \bar{t}_b) \tag{8.26}$$

Zum Pendelwechsel gilt:

$$\dot{x}(t_e)_x = \bar{\dot{x}}(\bar{t}_b)_x \tag{8.27}$$

$$\Rightarrow$$

$$\bar{\dot{x}}_{0,x} = \frac{\dot{x}(t_e)}{\cosh(k \cdot \bar{t}_b)}$$

Weiterhin gilt:

$$\bar{x}(\bar{t}_b)_x = \frac{\bar{\dot{x}}_{0,x} \cdot \sinh(k \cdot \bar{t}_b)}{k} \tag{8.28}$$

$$= \frac{\dot{x}(t_e)}{\cosh(k \cdot \bar{t}_b)} \cdot \frac{\sinh(k \cdot \bar{t}_b)}{k} \tag{8.29}$$

Für die Schrittweite gilt:

$$\bar{s}_x = x(t_e)_x - \bar{x}(\bar{t}_b)_x \tag{8.30}$$

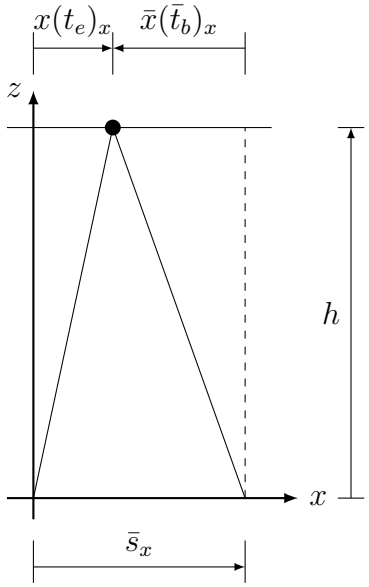


Abb. 8.11.: Wechsel der Pendelphase in x-Richtung

Abbildung 8.11 verdeutlicht das Vorgehen. Aus diesen Gleichungen kann eine Schrittweite berechnet werden, die dafür sorgt, dass sich in der nächsten Pendelphase der Schwerpunkt zum Zeitpunkt $t = 0$ über dem Ursprung des Pendels befindet. Dieser Ansatz wurde bereits bei dem anderen Verfahren ebenso verwendet. Es ist nun allerdings nicht immer zweckmäßig, den Schwerpunkt zu diesem Zeitpunkt über dem Ursprung zu haben.
Angenommen der Roboter soll in positive x-Richtung laufen, bekommt aber einen Stoß von vorne, sodass das Pendel nach hinten schwingt. Es wird nun eine negative Schrittweite berechnet, die ebenso dafür sorgt, dass der Schwerpunkt in der nächsten Phase über dem Ursprung liegt. In diesem Fall aber eventuell mit einer negativen Geschwindigkeit, sodass der Schwerpunkt auch am Ende der Phase eine negative Geschwindigkeit besitzen wird. Es wird wiederum ein Schritt nach hinten berechnet und somit unter Umständen mehr Schritte nach hinten getätigt als notwendig wären.
Um dies zu umgehen kann bei positiver Geschwindigkeitsvorgabe und einem negativen Wert für $\dot{x}(t_e)_x$ ein anderes Verfahren für die Berechnung der Schrittweite angewandt werden. Die verfolgte Strategie ist dabei, dass ausgehend von der Endposition $x(t_e)_x$ eine Schrittweite berechnet wird, die dafür sorgt, dass sich der Schwerpunkt zum Ende der nächsten Phase wieder möglichst nah an dieser Position befinden wird. Abbildung 8.12 verdeutlicht die Situation zum Zeitpunkt $t = t_e$ der aktuellen Pendelphase. Für die Berechnung wird im Folgenden angenommen, dass wie auch für die Lateralbewegung zum Zeitpunkt $t = 0$ die Geschwindigkeit null ist und somit

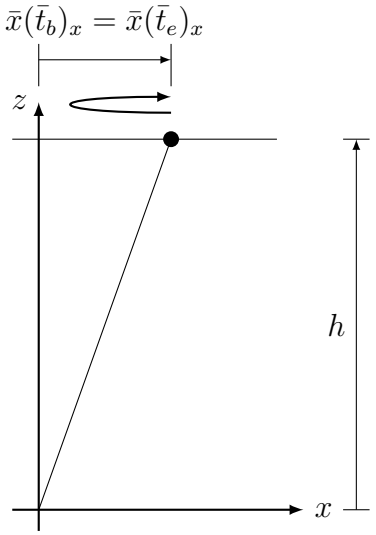


Abb. 8.12.: Erreichen der Ausgangsposition am Ende der Pendelphase

$\dot{\bar{x}}_{0,x} = 0$ gilt. Weiterhin ergibt sich:

$$\dot{x}(t_e)_x = \dot{\bar{x}}(\bar{t}_b)_x = \bar{x}_{0,x} \cdot k \cdot \sinh(k \cdot \bar{t}_b) \tag{8.31}$$

$$\Rightarrow$$

$$\bar{x}_{0,x} = \frac{\dot{x}(t_e)_x}{k \cdot \sinh(k \cdot \bar{t}_b)}$$

Zudem gilt:

$$\bar{x}(\bar{t}_b)_x = \bar{x}_{0,x} \cdot \cosh(k \cdot \bar{t}_b) \tag{8.32}$$

$$= \frac{\dot{x}(t_e)_x}{k \cdot \sinh(k \cdot \bar{t}_b)} \cdot \cosh(k \cdot \bar{t}_b) \tag{8.33}$$

Die Schrittweite errechnet sich wie zuvor zu $\bar{s}_x = x(t_e)_x - \bar{x}(\bar{t}_b)_x$. Es wird mit diesem Verfahren ein Schritt gewählt, der dafür sorgt, dass zu Beginn der nächsten Pendelphase wieder eine positive Schwerpunktgeschwindigkeit vorliegt. Es ist allerdings möglich, dass eine zu große Schrittweite berechnet wird, die der Roboter nicht ausführen kann. Somit ist eine Limitierung der Schrittweite notwendig.

Der hier vorgestellte neue Ansatz konnte bereits erfolgreich in der Simulation getestet werden. Es waren Geschwindigkeiten von bis zu 32 cm/s erreichbar. Ein Problem dieser Methode stellt sich beim Wechsel des Supportfußes dar. Da die Neuberechnung auf den zwei letzten Schwerpunktpositionen basiert, müssen diese Positionen auch im selben Koordinatensystem angegeben sein. Da beim Wechsel allerdings das Koordinatensystem ebenfalls seine Position verändert, ist dies aber nicht der Fall. Die Überführung der letzten Position in das neue Koordinatensystem muss so exakt

wie möglich erfolgen. Die Schrittweite ist zwar bekannt, wird aber in der Regel nicht exakt erreicht, sodass eine Transformation der letzten Position entlang der Schrittweite zu keinen guten Ergebnissen führte. Ein alternativer Ansatz ist die letzte Position aus der aktuellen Position und der Geschwindigkeit zu approximieren:

$$x(t_{-1})_x = x(t)_x - \dot{x}(t)_x \tag{8.34}$$

Dennoch ist auch diese Annäherung nicht exakt, sodass auch hier beim Wechsel große Störungen bei der Geschwindigkeit erfassbar sind, die sich auf die Schrittweite auswirken. Diese Störungen sind allerdings nur in einem Zyklus vorhanden, sodass diese Störungen keine allzu großen Auswirkungen haben. Abbildung 8.13 zeigt den Geschwindigkeitsverlauf für eine Vorgabe von $0,2$ m/s. Die Regulierung der Geschwindigkeit erfolgt wie zuvor auch durch eine Gewichtsverlagerung. Die Kurven für Geschwindigkeit und Schrittweite zeigen hierbei gefilterte Werte.

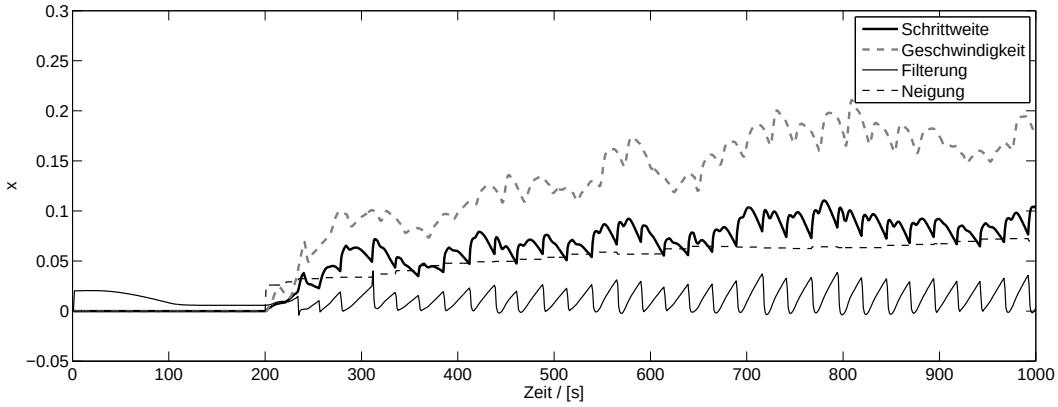


Abb. 8.13.: Geschwindigkeitsmessung für den neuen Ansatz

Der hier vorgestellte neue Ansatz berücksichtigt die tatsächliche Schwerpunktgeschwindigkeit des Roboters für die Parameteraktualisierung, allerdings ergeben sich hierdurch andere Probleme, die noch gelöst werden müssen. Vorteile konnten bei stärkeren frontalen Störungen während des Gehens festgestellt werden, da der Roboter bei diesem Ansatz einen größeren Ausfallschritt nach hinten macht als bei dem in Abschnitt 8.3 evaluierten Ansatz.

Die Übertragung dieses Ansatzes auf den realen Roboter konnte bisher noch nicht erfolgreich abgeschlossen werden, da die Messungen zu stark verrauscht gewesen sind und damit das Pendelmodell zwischen zwei Prozessschritten zu stark veränderten. Es ist aber dennoch denkbar, dass sich dieser vielversprechende Ansatz durch geeignete Filterung auch auf den realen Roboter erfolgreich übertragen lassen könnte. Dies wurde aus Zeitgründen allerdings nicht mehr versucht.

9. Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war die Implementierung eines dynamischen Laufalgorithmus für das *Nao*-Robotiksystem. Hierzu war zunächst die Einrichtung einer Simulations- und Testumgebung notwendig. Nach einem Überblick über mögliche Simulatoren wurde mit *Webots* ein, den Ansprüchen genügender Simulator ausgewählt und beschafft. Bei der Einrichtung der Simulationsumgebung wurde darauf geachtet, dass für neu entwickelte eigene Module sowohl für den Roboter als auch für den Simulator der identische Quellcode verwendet werden kann. Hierzu war die Entwicklung von Zwischenmodulen und die Nachbildung einiger Prozesse des Roboters notwendig.
Eine einfache Möglichkeit, Daten in der Simulation grafisch zu überwachen, wurde durch die Steuerung des Simulators durch *Matlab* erreicht. Die Verknüpfung von *Matlab* mit den in C++ entwickelten eigenen Modulen konnte über *mex*-Funktionen erreicht werden.
Um die Messaufzeichnungen durch Simulation mit den Messaufzeichnungen des Roboters vergleichen zu können, wurde ein Programm benötigt, das ermöglicht, Daten des Roboters aufzuzeichnen und abzuspeichern. Da keine Software zur Verfügung stand, wurde eine eigene Software entwickelt, die dies ermöglicht.
Nachdem sowohl eine Simulations- als auch eine Messumgebung eingerichtet waren, konnte mit der Entwicklung des Laufalgorithmus begonnen werden. Hierzu wurden zunächst verschiedene Modellansätze mit den Vor- und Nachteilen vorgestellt und ein für den *Nao* passendes Modell ausgewählt.
Die Gemeinsamkeit aller Laufstrategien liegt in der Positionierung der Roboterfüße relativ zu dessen Schwerpunkt. Um diese Positionierung durchführen zu können, sind jedoch einige kinematische Berechnungen notwendig. Daher wurden Module entwickelt, die sowohl Vorwärtskinematik als auch inverse Kinematik des Roboters realisieren. Hierbei wurde die Vorwärtskinematik hauptsächlich für die Schwerpunktsberechnung des Roboters genutzt. Die inverse Kinematik ermöglichte es, die Füße des Roboters relativ zu dessen Schwerpunkt zu setzen, indem sie die dazu notwendigen Gelenkwinkel berechnete.
Unter diesen Vorraussetzungen wurde auf Basis des *3DLIPM* ein Modell implementiert, das eine Vorhersage über die Schwerpunktsposition trifft. Die Strategie des Laufalgorithmus liegt nun darin, dieser Schwerpunktvorhersage zu folgen. Um dennoch Einfluss auf das System zu haben, und somit auf Störungen reagieren zu können, wurden die Länge der Supportphasen sowie die Schrittweite des Roboters angepasst.
Eine Regulierung der Laufgeschwindigkeit konnte durch eine zusätzliche Torsoneigung des Roboters erreicht werden.

Es konnte gezeigt werden, dass ein Laufalgorithmus dieser Art in der Lage ist, auf

Störungen zu reagieren und diese in gewissen Grenzen abzufangen. Des Weiteren konnte gezeigt werden, dass eine Regulierung einer zusätzlichen Torsoneigung dazu geeignet ist, die Laufgeschwindigkeit des Roboters zu kontrollieren.

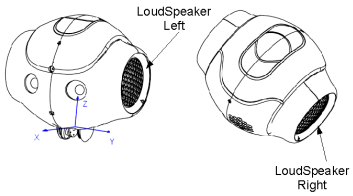
Obwohl die Ziele dieser Arbeit erreicht werden konnten, bietet sie dennoch bloß einen Einstieg in das komplexe Feld der Nachbildung humanoider Laufbewegungen. Es werden daher weitere Arbeiten notwendig sein, um das System weiter zu verbessern und robuster zu gestalten. In dieser Arbeit wurden die meisten Parameter für das Modell experimentell bestimmt. Der hier gezeigte Algorithmus lässt sich durch Optimierung dieser Größen sicherlich verbessern. Ein Ausblick auf eine zukünftige Arbeit kann also die Optimierung der Pendelparameter sein. Des Weiteren bietet der Algorithmus die Möglichkeit, omnidirektionale Bewegungen auszuführen. So werden bei dem Ansatz auch seitliche Schrittweiten und die Möglichkeit berücksichtigt, Kurven zu gehen. Eine Umsetzung ist aus zeitlichen Gründen in dieser Arbeit allerdings nicht erfolgt, sodass eine weitere mögliche Arbeit die Erweiterung des Modelles auf omnidirektionale Bewegungen darstellen könnte. Des Weiteren wurde ein neuartiger Ansatz zur Neuberechnung der Pendelparameter entwickelt, der im Simulator schon erfolgreich getestet werden konnte, in dieser Form allerdings noch nicht auf dem realen Roboter vollständig umgesetzt werden konnte. Eine Weiterentwicklung dieses Ansatzes oder die Kombination mit einigen Komponenten des ursprünglichen Verfahrens bieten sich ebenfalls als zukünftige Projekte an. Neben der reinen Weiterentwicklung und Verbesserung des Laufalgorithmus existieren noch die Aufgabenfelder, den Roboter möglichst exakt von einer Position an eine bestimmte andere Position zu bewegen. Ein Ansatz für eine Pfadplanung wurde bereits vorgestellt und implementiert, allerdings kann derzeit nicht überwacht werden, wie genau der Roboter dem vorgegebenen Pfad folgt. Eine Unterstützung hierbei kann eine Selbstlokalisierung im Raum bieten, die durch visuelle Verarbeitung der Umgebung durch den Roboter erlangt werden kann.
Im Hinblick darauf, den Roboter in internationalen Wettbewerben teilnehmen zu lassen, werden zudem Entwicklungen im Bereich der Ballerkennung, Schusstechniken und Spielstrategien notwendig sein.

Insgesamt bietet sich daher ein enorm weites Feld für zukünftige Arbeiten mit dem *Nao*-Robotiksystem.

A. Hardwareübersicht

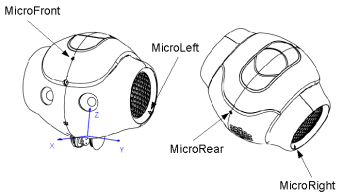
Körpereigenschaften	
Größe	$\sim$ 58cm
Gewicht	$\sim$ 5kg

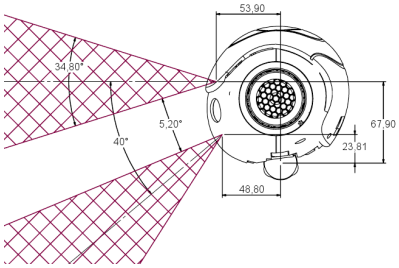
Batterie	
Batterietyp	Lithium Ionen
Gewicht	$\sim$ 350g
Kapazität	2 Ah (21,6V Nominalspannung)
Ladezeit	$\sim$ 2h
Betriebsdauer	$\sim$ 90min



Lautsprecher	
Anzahl	2
Position	linkes Ohr, rechtes Ohr

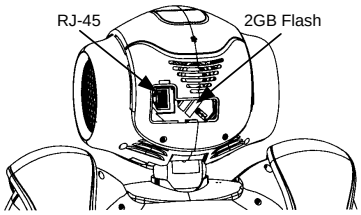
Mikrofone	
Anzahl	4
Position	linkes Ohr, rechtes Ohr, Stirn, Hinterkopf
Bandbreite	300Hz - 8kHz



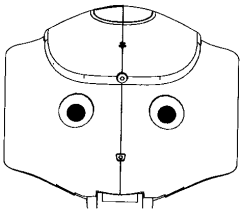


Kameras	
Anzahl	2
Position	Stirn und Mund
Auflösung	640×480
Framerate	30fps
Fokus	30cm - ∞
Fokustyp	Fixfokus

Netzwerk	
Ethernet	RJ-45 (Hinterkopf)
Wireless	Wi-Fi (IEEE 802.11g)

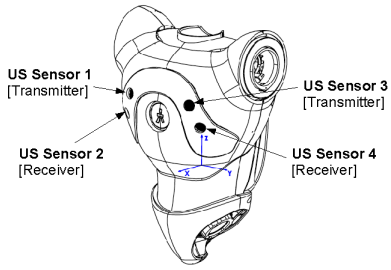


Motherboard	
CPU	AMD GEODE 500MHz
Speicher	256MB SDRAM / 2GB Flash



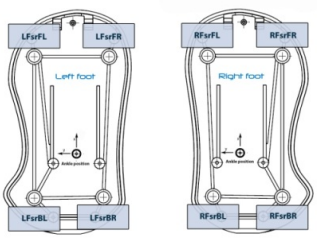
Infrarot	
Anzahl	2
Position	Augen
Wellenlänge	940nm
Abstrahlwinkel	±60°
Leistung	8mW/sr

Inertiales Navigationssystem	
Position	Torso
Komponenten	2× einachsiges Gyroskop 1× dreiachsiger Beschleunigungssensor
Genauigkeit	Gyroskop: 5% Beschleunigungssensor: 1%



Ultraschall	
Kanäle	2
Position	Torso
Frequenz	40kHz
Sensitivität	-86dB
Auflösung	9mm
Messbereich	0,2 - 1,2m
Messkegel	60°

Resistive Kraftsensoren	
Anzahl	8
Position	Fuß (je 4)
Messbereich	0-25N
Genauigkeit	∼ 20%



Aktoren/Freiheitsgrade	
Anzahl	25
Aufteilung	Kopf: 2 Arm: je 4 Bein: je 5 Hüfte: 1 Hand: je 2
Ansteuerung	dsPICS Mikrocontroller
Motorart	kernloser Gleichstrommotor

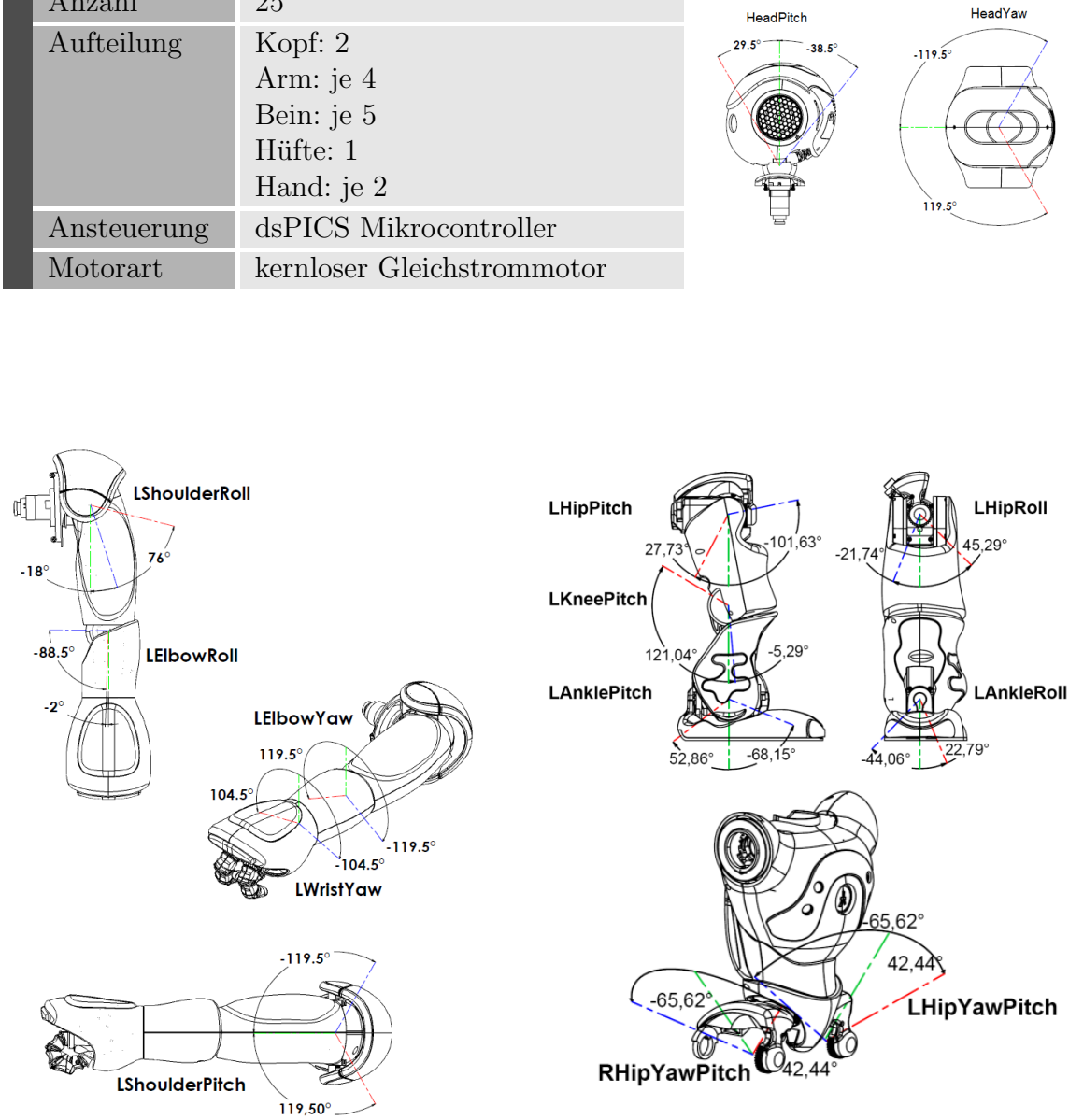
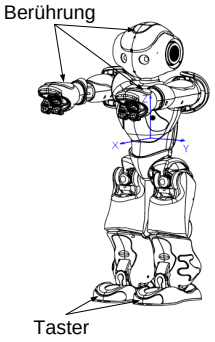


Abb. A.1.: Übersicht über die Aktoren des Roboters

Aktorsensorik	
Typ	Hallsensor
Anzahl	36
Aufteilung	je zwei pro Freiheitsgrad in der unteren Körperhälfte (Hüfte, Beine) je einer pro Freiheitsgrad in der oberen Körperhälfte (Arme, Hände, Kopf)
Auflösung	12bit $\widehat{=}$ 0,1°



Berührungs-/Tastsensoren	
Anzahl	7
Position	Kopf: 3 Handrücken: je 1 Fuß: je 1
Art	Kopf und Hand: Kapazitive Berührungssensoren Fuß: Taster

LEDs	
Anzahl	51
Art	19× RGB Farbwechselnd 32× 16 Blaustufen
Position	Berührungssensor Kopf: 12× Blau Augen: 2 × 8 Farbwechselnd Ohren: 2 × 10 Blau Torso: 1× Farbwechselnd Fuß: 2 × 1 Farbwechselnd

B. Inhalte der CD

- Bericht,
- Literatur,
- Quellcode.

Literaturverzeichnis

[Ald11] Aldebaran Robotics (Hrsg.): *Nao User Guide*. 1.10.52. Aldebaran Robotics, 2011

[CKU10] Czarnetzki, S. ; Kerner, S. ; Urbann, O.: Applying Dynamic Walking Control for Biped Robots. In: Baltes, J. (Hrsg.) u. a.: *RoboCup 2009: Robot Soccer World Cup XIII*. Springer-Verlag Berlin Heidelberg, 2010, S. 69–80

[GHB+08] Gouaillier, David ; Hugel, Vincent ; Blazevic, Pierre ; Kilner, Chris ; Monceaux, Jérôme ; Lafourcade, Pascal ; Marnier, Brice ; Serre, Julien ; Maisonnier, Bruno: The NAO humanoid: a combination of performance and affordability. In: *CoRR* abs/0807.3223 (2008)

[GHRL09] Graf, Colin ; Härtl, Alexander ; Röfer, Thomas ; Laue, Tim: A Robust Closed-Loop Gait for the Standard Platform League Humanoid. In: Zhou, Changjiu (Hrsg.) ; Pagello, Enrico (Hrsg.) ; Menegatti, Emanuele (Hrsg.) ; Behnke, Sven (Hrsg.) ; Röfer, Thomas (Hrsg.): *Proceedings of the Fourth Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS International Conference on Humanoid Robots*. Paris, France, 2009, S. 30 – 37

[GLP01] Gienger, M. ; Löffler, K. ; Pfeiffer, F.: Towards the Design of a Biped Jogging Robot. In: *IEEE International Conference on Robotics & Automation*, 2001, S. 4140–4145

[GR10] Graf, C. ; Röfer, T.: A Closed-loop 3D-LIPM Gait for the RoboCup Standard Platform League Humanoid. In: Zhou, C. (Hrsg.) ; Pagello, E. (Hrsg.) ; Behnke, S. (Hrsg.) ; Menegatti, E. (Hrsg.) ; Röfer, T. (Hrsg.) ; Stone, P. (Hrsg.): *Proceedings of the Fourth Workshop on Humanoid Soccer Robots in conjunction with the 2010 IEEE-RAS International Conference on Humanoid Robots*, 2010

[GR11] Graf, C. ; Röfer, T.: A Center of Mass Observing 3D-LIPM Gait for the RoboCup Standard Platform League Humanoid. In: Röfer, T. (Hrsg.) ; Meyer, N.M. (Hrsg.) ; Savage, J. (Hrsg.) ; Saranli, U. (Hrsg.): *RoboCup 2011: Robot Soccer World Cup XV*, 2011

[KKK+02a] Kajita, S. ; Kanehiro, F. ; Kaneko, K. ; Fujiwara, K. ; K. Yokoi, H. H.: A Realtime Pattern Generator for Biped Walking. In: *IEEE International Conference on Robotics & Automation*, 2002

[KKK+02b] Kaneko, K. ; Kajita, S. ; Kanehiro, F. ; Yokoi, K. ; Fujiwara, K. ; Hirukawa, H. ; Kawasaki, T. ; Hirata, M. ; Isozumi, T.: Design of Advanced Leg Module for Humanoid Robotics Project of METI. In: *IEEE International Conference on Robotics & Automation*, 2002, S. 38–45

[KKK+03] Kajita, S. ; Kanehiro, F. ; Kaneko, K. ; Fujiwara, K. ; Harada, K. ; Yokoi, K. ; Hirukawa, H.: Biped Walking Pattern Generation by using Preview Control of Zero-Moment Point. In: *International Conference on Robotics & Automation*, 2003

[KOIK85] Katayama, T. ; Ohki, T. ; Inoue, T. ; Kato, T.: Design of an optimal controller for a discrete-time system subject to previewable demand. In: *International Journal of Control* 41 (1985), Nr. 3, S. 677–699

[LaV06] LaValle, S. M.: *Planning Algorithms*. Cambridge, U.K. : Cambridge University Press, 2006. – Available at http://planning.cs.uiuc.edu/

[Lun10] Lunze, J.: *Regelungstechnik 2: Mehrgrößensysteme, Digitale Regelung*. Bd. 6. Springer-Verlag Berlin Heidelberg, 2010

[PKLO06] Park, I.W. ; Kim, J.Y. ; Lee, J. ; Oh, J.H.: Online Free Walking Trajectory Generation for Biped Humanoid Robot KHR-3(HUBO). In: *IEEE International Conference on Robotics & Automation*, 2006, S. 1231–1236

[SK08] Siciliano, B. (Hrsg.) ; Khatib, O. (Hrsg.): *Handbook of Robotics*. Springer-Verlag Berlin Heidelberg, 2008. – ISBN 978–3–540–23957–4

[SKM+04] Sugahara, Y. ; Kawase, M. ; Mikuriya, Y. ; Hosobata, T. ; Sunazuka, H. ; Hashimoto, K.: Support Torque Reduction Mechanism for Biped Locomotor with Parallel Mechanism. In: *IEEE/RSJ Interational Conference on Intelligent Robots Systems*, 2004, S. 3213– 3218

[Str09] Strom, J.H.: *Dynamically Balanced Omnidirectional Humanoid Robot Locomotion*. An Honors Paper for the Department of Computer Science, 2009

[VB04] Vukobratović, M. ; Borovac, B.: Zero-Moment-Point - Thirty Five Years Of Its Life. In: *International Journal of Humanoid Robotics* 1 (2004), Nr. 1, S. 157–173

[WB06] Welch, Greg ; Bishop, Gary: An Introduction to the Kalman Filter. In: *In Practice* 7 (2006), Nr. 1, S. 1–16