technische universität
dortmund

Endbericht der Projektgruppe 560

Fußballspielende humanoide Roboter

Emanuel Gaisenkersting, Jens Große-Allermann, Daniel Hüsmert, Sascha Kwiatkowski, Dino Menges, Lars Ostwald, Fabian Pawlowski, Robert Rapczynski, Michael Schmidt, Ingmar Schwarz, Sebastian Sudholt, Max Vallender

September 2012

Betreuer:
Prof. Dr.-Ing. Uwe Schwiegelshohn
Dipl.-Inf. Oliver Urbann
Dipl.-Inf. Stefan Tasse

Technische Universität Dortmund
Institut für Roboterforschung
Lehrstuhl für Datenverarbeitungssysteme
http://www.irf.tu-dortmund.de

FUSSBALLSPIELENDE HUMANOIDE ROBOTER

EMANUEL GAISENKERSTING, JENS GROSSE-ALLERMANN, DANIEL HÜSMERT, SASCHA KWIATKOWSKI, DINO MENGES, LARS OSTWALD, FABIAN PAWLOWSKI, ROBERT RAPCZYNSKI, MICHAEL SCHMIDT, INGMAR SCHWARZ, SEBASTIAN SUDHOLT, MAX VALLENDER

Endbericht der Projektgruppe 560

Lehrstuhl für Datenverarbeitungssysteme
Institut für Roboterforschung
Technische Universität Dortmund

September 2012

Emanuel Gaisenkersting, Jens Große-Allermann, Daniel Hüsmert, Sascha Kwiatkowski, Dino Menges, Lars Ostwald, Fabian Pawlowski, Robert Rapczynski, Michael Schmidt, Ingmar Schwarz, Sebastian Sudholt, Max Vallender:
Fußballspielende humanoide Roboter,
Endbericht der Projektgruppe 560, September 2012

BETREUER:
Prof. Dr.-Ing. Uwe Schwiegelshohn
Dipl.-Inf. Oliver Urbann
Dipl.-Inf. Stefan Tasse

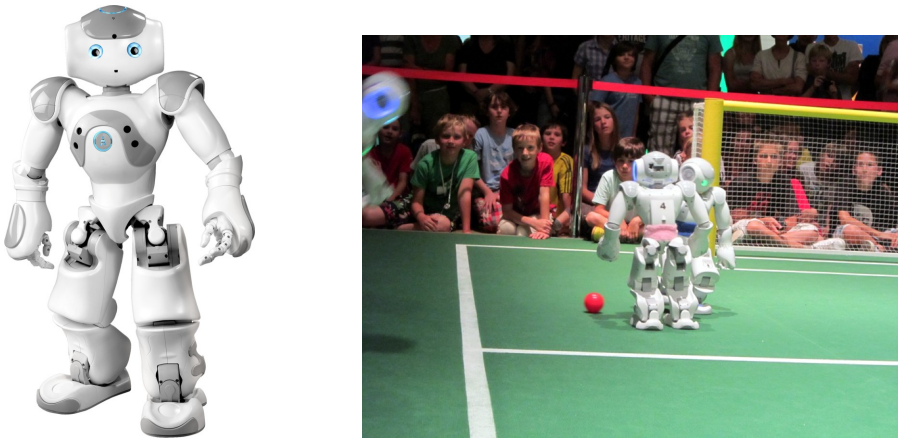
INHALTSVERZEICHNIS

1 EINLEITUNG

1.1 MOTIVATION

Die Möglichkeit an aktuellen Themen in aktiven Forschungsgebieten zu arbeiten bietet sich nicht jeder Projektgruppe. Dies erlaubt neben dem Sammeln erster Erfahrungen in selbst organisierter Projektarbeit einen persönlichen Einblick in die Arbeit als Forscher. Zudem erlaubt die Einbettung der in dieser Projektgruppe gestellten Aufgaben in den Roboterfußball in spielerischer Weise einen Überblick über verschiedenste Aspekte.

Roboterfußball



(a) NAO von Aldebaran Robotics

(b) Roboterfußballspiel

Abbildung 1.1: SPL Liga

Die Teilnahme an Turnieren der Standard-Platform-League (SPL), in der baugleiche Roboter vom Typ NAO (Abb.: 1.1) gegeneinander Fußball spielen, erlaubt die Ergebnisse der Arbeit nicht nur in der Praxis, sondern direkt mit anderen Forschern am aktuellen Stand der Forschung zu messen und auszutauschen. Dies gibt einen tieferen Einblick in die Thematik und die Möglichkeit direkt am Puls der aktuellen Forschung mitzuwirken.

1.2 ZIELE

Das Ziel der Projektgruppe ist die (semi-)automatisierte Lösung von mindestens zwei Problemstellungen sowie deren Optimierung. Für diese Aufgabe kommen Black-Box-Verfahren in Frage, wie etwa evolutionäre Algorithmen. Die zu optimierenden Problemstellungen sind aus dem Bereich der Wahrnehmung und der Bewegungsplanung zu

Black-Box-Optimierung

wählen. Neben den vorgeschlagenen Zielen ist es des Weiteren auch möglich, weitere Aufgaben mit den Betreuern abzusprechen.

Kamerakalibrierung

Im Bereich der Wahrnehmung dient die automatische Kalibrierung der Kamera dazu, eventuelle Fehlstellungen und Fehljustierungen, die über die Zeit entstehen, zu beheben. Dies hat zum Ziel, die Lokalisierung, welche zu einem großen Teil auf der Bildanalyse beruht, zu verbessern.

Zudem ist die Optimierung der Farbräume für die zur Bilderkennung genutzte Farbtafel zu automatisieren, um die an unterschiedlichen Orten vorhandenen Farbschwankungen und Lichtverhältnisse zeitnah zu kompensieren.

Gelenkwinkelkalibrierung

Im Bereich der Bewegung erlaubt es die automatische Gelenkwinkelkalibrierung die Gelenkwinkel zu justieren, was in besonderem Maße die Korrektheit der auf den Winkeln beruhenden Berechnungen des Laufs verbessern kann. Darüber hinaus wurden weitere Aufgaben und Interessengebiete bearbeitet.

2 GRUNDLAGEN DER OPTIMIERUNG

Hauptthema der Projektgruppe sind Optimierungsprobleme, für die geeignete Optimierungsstrategien gewählt werden müssen. Im Rahmen der Aufgaben wurden drei Verfahren implementiert und eingesetzt. Die Wahl fiel hierbei auf den Nelder-Mead-Simplex Algorithmus, einen $(\mu + \lambda)$-EA und die CMA-ES. Diese Optimierungsstrategien werden im folgenden Kapitel grundlegend erläutert, sowie ihre Vor- und Nachteile aufgezeigt.

2.1 NELDER-MEAD-SIMPLEX ALGORITHMUS (SIMPLEX ALGORITHMUS)

Der Simplex Algorithmus (vgl. [NM65]) gehört zur Kategorie der direkten Suchverfahren. Er kommt im Gegensatz zu den meisten Verfahren allerdings ohne Ableitung aus. Obwohl er nicht randomisiert ist und damit in lokalen Minima hängen bleiben kann, gilt er als relativ robust. Grundgedanke für den Einsatz dieses Verfahrens bei den gestellten Optimierungsaufgaben waren zum einen die einfache Implementierung und die Möglichkeit eines sogenannten Warmstarts. Damit wird die Möglichkeit bezeichnet den Algorithmus von einem Punkt nahe eines Optimums zu starten und schnell zu diesem Optimum zu Konvergieren. Praktisch würde dies z.B. bei der Kameraoptimierung (vgl. Kap. 3.1) bedeuten, dass ein guter Parametersatz eines Roboters auf einen zweiten Roboter Übertragen werden könnte und von diesem aus eine Optimierung für diesen Roboter durchgeführt wird. Ausgehend davon, dass die Lösungen für beide Roboter sehr nahe beieinander liegen und sich nur durch kleine Unterschiede zwischen den Robotern unterscheiden, wäre so eine sehr schnelle Kalibrierung möglich.

ohne Ableitung

Warmstart

2.1.1 *Algorithmus*

Ein Simplex ist ein Volumen in einem N-Dimensionalen Raum, dass von $N + 1$ Punkten aufgespannt wird. Beispielsweise im Zweidimensionalen ein Dreieck, im Dreidimensionalen ein Tetraeder, usw.

Ausgehend von einem Simplex im Suchraum des Optimierungsproblems, wird allen Punkten des Simplex mittels einer Fitnessfunktion ein Kostenwert zugewiesen. Mittels dieser Werte wird der beste und schlechteste Punkt zum Zeitpunkt t ermittelt. Im nächsten Iterationsschritt $t + 1$ wird der beste Wert als aktuelles Optimum stets beibehalten, der schlechteste Punkt jedoch durch einen vermeintlich

schlechtester Punkt wird ersetzt

besseren ersetzt, indem je nach Situation Reflexion, Expansion, Kontraktion und Komprimierung (vgl. Abb.: 2.1) durchgeführt werden. Beeinflusst wird der Algorithmus hierbei durch die Strategieparameter $\alpha \in \mathbb{N}$, $\gamma > 1$ und $\beta \in [0, 1]$. Insgesamt ergibt sich der folgende Ablauf für ein Problem der Dimension N.

I. Man wähle $N + 1$ Startpunkte $P_0, \ldots, P_N$ auf beliebige Weise

II. Die $N + 1$ Punkte werden mittels einer Fitnessfunktion bewertet. P_w bezeichnet im folgenden den am schlechtesten bewerteten Punkt des Simplex, P_b den am besten bewerteten. $\overline{P}$ bezeichnet den Mittelpunkt des Simplex.

III. Reflexion, Expansion, Kontraktion und Komprimierung

1. Wenn P_b gemäß eines Kriteriums gut genug ist, wird das Verfahren beendet.

2. Der schlechteste Punkt P_w wird am Mittelpunkt des Simplex mit Hilfe des Strategieparameters α zu $P' = (1 + \alpha)\overline{P} - \alpha P_w$ reflektiert (Reflexion Abb.: 2.1a).

   a) Wenn P' besser als P_w und schlechter als P_b bewertet wird, so ersetze P_w durch P' und beginne die nächste Iteration $\rightarrow$ III.

   b) Wenn P' besser bewertet wird als P_b, so wird mit dem Strategieparameter γ zu $P'' = (1 + \gamma)\overline{P} - \gamma P'$ expandiert. (Expansion Abb.: 2.1b)

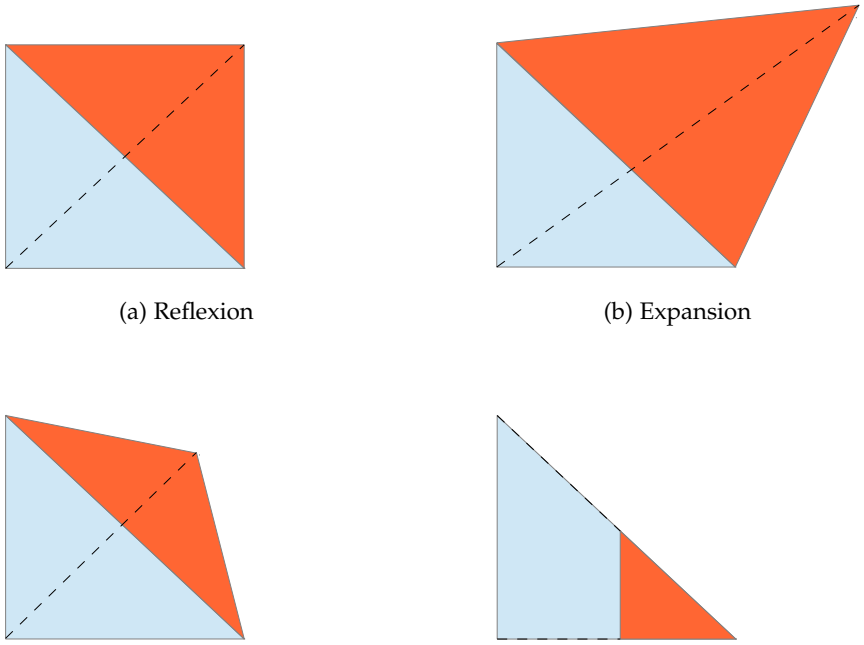
      i. Wenn auch P'' besser als P_b ist, so wird P_w durch P'' ersetzt $\rightarrow$ III.

      ii. Wenn P'' nicht besser als P_b bewertet wird, so wird P_w durch P' ersetzt $\rightarrow$ III..

   c) Wenn P' schlechter bewertet wird als alle $P_i \neq P_w$, so wählt man aus P', P_w den besser bewerteten Punkt aus und definiert diesen als P_w. Anschließend wird P_w um den Faktor β an den Mittelpunkt des Simplex herangezogen $P''' = (1 + \beta)\overline{P} - \beta P_w$ (Kontraktion Abb.: 2.1c).

      i. Ist P''' besser als der schlechteste, beginne die nächste Iteration $\rightarrow$ III.

      ii. Ist P''' schlechter als P_w, ersetze alle Punkte P_i des Simplex durch die Punkte $P_i' = (P_i + P_b/2)$. Es werden also alle Punkte um die Hälfte ihrer Distanz in Richtung des besten Punktes verschoben. (Komprimierung Abb.: 2.1d) Danach beginnt die nächste Iteration $\rightarrow$ III.



(a) Reflexion (b) Expansion

(c) Kontraktion (d) Komprimierung

Abbildung 2.1: Skizzen der Schritte des Simplex-Algorithmus im Zweidimensionalen. Blau vor dem jeweiligen Schritt, Orange nach der Durchführung.

2.2 EVOLUTIONÄRE ALGORITHMEN

Evolutionäre Algorithmen (EA) sind *Black-Box-Verfahren* zur Optimierung verschiedenster Probleme. Das bedeutet, dass der Algorithmus ohne zusätzliches Wissen über das Problem auskommt. Sie gehören, wie der zuvor beschriebene *Simplex-Algorithmus*, zu den direkten Suchverfahren und benötigen keine Ableitung. Außerdem ist es analog zum *Simplex-Algorithmus* möglich, einen Warmstart durchzuführen. Allerdings sind im Gegensatz zum *Simplex-Algorithmus* die EAs randomisiert und besitzen Vorteile beim Verlassen von lokalen Minima.

ohne Ableitung

Warmstart

randomisiert

Wie der Name schon andeutet, sind EAs an die biologische Evolution angelehnt. Die Idee dabei ist, dass sich hochkomplexe Lebensformen durch Veränderung des Erbgutes an ihre Umwelt anpassen. Eine möglichst gute Anpassung an die Umwelt kann als Optimierungsproblem verstanden werden. EAs greifen diese Idee auf und versuchen durch zufällige Veränderung der Parameter (dem *Phänotyp*) das Optimum im Suchraum (die beste Anpassung an die Umwelt) zu finden. Die Kodierung der Parameter (binär, reell, usw.) bildet den *Genotyp*, während der darauf basierende Parametersatz den *Phänotyp* darstellt. Letzterer stellt ein Individuum in einer Population, einer Menge von Individuen, dar. Jede Population erzeugt Nachkommen, eine neue

Phänotyp

Genotyp

Generation von Individuen, die sich ihrer Umwelt weiter angepasst haben. Beim EA geschieht das durch eine Abfolge von *Rekombination*, *Mutation*, *Evaluation* und *Selektion*, wobei die Rekombination in vielen EAs ausgelassen wird.

REKOMBINATION mischt die Daten (*Genome*) ausgewählter Individuen miteinander. Eine mögliche Form der Rekombination, bei binär kodiertem *Genotyp*, wäre eine **AND**, **OR** oder **XOR** Verknüpfung der *Genome*.

MUTATION erzeugt eine neue Generation von Individuen. Hierbei werden die Individuen (der *Phänotyp*) der aktuellen Population zufällig verändert.

EVALUATION beurteilt die Individuen mittels einer Fitnessfunktion. Die Fitnessfunktion bewertet dabei, wie nahe die einzelnen Individuen dem Optimum (der perfekten Anpassung an ihre Umwelt) sind.

SELEKTION wählt aus den vorhandenen Individuen die am besten geeigneten aus (survival of the fittest).

μ *Individuen,* λ *Nachkommen*

Die Größe der Populationen ergibt sich aus dem EA. Wird von einem $(\mu + \lambda)$-EA gesprochen, bedeutet dies, dass jede Population aus μ vielen Individuen besteht. Die Anzahl der erzeugten Kinder wird durch λ gegeben, während $+$ die Art und Weise der Selektion beschreibt.

Dabei gibt es verschiedene Arten von EAs. Sie unterscheiden sich in *genetische Algorithmen*, *evolutionäre Strategien* und *evolutionäre Programmierung*, wobei die Grenzen immer weiter verschmelzen. An dieser Stelle liegt der Fokus auf *evolutionären Strategien*. Der Suchraum, auf dem *evolutionäre Strategien* arbeiten, ist meistens eine Teilmenge des $\mathbb{R}^n$.

2.2.1 *Algorithmus*

Zu Beginn wird eine zufällige Population erzeugt. Diese initialen Individuen werden anhand einer Fitnessfunktion bewertet. Anschließend beginnt eine Schleife aus *Rekombination*, *Mutation*, *Evaluation* und *Selektion*, die zyklisch durchlaufen wird, bis ein Abbruchkriterium erfüllt ist (vgl. Abb. 2.2). Der in späteren Kapiteln genutzte EA läuft allerdings wie folgt ab:

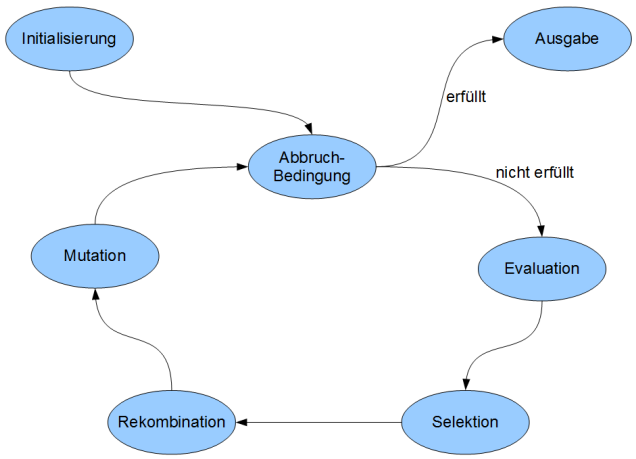


Abbildung 2.2: Allgemeine Schematische Darstellung eines evolutionären Algorithmus

1. Erzeuge $\mu + \lambda$ Individuen zufällig.
2. Solange Abbruchkriterium nicht erfüllt:
   a) Evaluation
   b) Selektion
   c) Mutation

Der hier verwendete Algorithmus verzichtet auf die oben genannte Rekombination. Die Veränderung der Individuen findet ausschließlich während der Mutation statt. *keine Rekombination*

SELEKTION Bei der Selektion werden die Individuen für die nächste Population ausgewählt. Es gibt verschiedene Möglichkeiten, die Individuen zu selektieren. Unterschieden wird in die Selektionsoperatoren „,“ und „+“.

Bei dem „,“-Selektionsoperator werden die besten μ Individuen aus den λ mutierten Individuen selektiert. Dabei muss $\lambda \geqslant \mu$ gelten, wobei $\lambda = \mu$ einem „random walk“ mit ggf. mehreren Wegen entspricht. Es gilt zu beachten, dass die neue Population ausschließlich aus den mutierten Kindern der Elternpopulation besteht [Wei07]. Somit werden auch Individuen verworfen, die bessere Fitnesswerte haben, als manche Kinder. *„,“-Selektion* *„+“-Selektion*

Bei dem „+“-Selektionsoperator werden die μ Individuen für die nächste Generation sowohl aus den μ Individuen der Eltern, als auch aus den λ Individuen der mutierten Kinder gewählt [Wei07]. Der Vorteil dieser Selektion ist, dass ein für gut befundenes Individuum so lange behalten wird, bis μ viele Individuen gefunden wurden, die bessere Fitnesswerte haben. Ein Nachteil hingegen besteht darin, dass

dadurch lange in lokalen Minima verharrt werden kann, falls diese überhaupt wieder verlassen werden, da die Mutation immer in der Umgebung der Elter-Individuen stattfindet. Es gibt weitere Ansätze, die versuchen die guten Eigenschaften beider Selektionsoperatoren zu vereinen. Eine dieser Ideen wählt den „+"-Selektionsoperator, der eine Höchstlebensdauer für Individuen angibt. Dadurch bleiben gute Individuen zunächst erhalten. Haben sie eine Höchstanzahl an Generationen überdauert, werden sie nicht erneut selektiert. Individuen können in diesem Fall, wie beim „,"-Selektionsoperator, sterben.

Höchstlebensdauer

Für beide Möglichkeiten der Selektion gibt es verschiedene Selektionsstrategien. Diese Selektionsstrategien besagen, wie die μ geeignetsten Individuen bestimmt werden. Es ist dabei zu beachten, dass die Selektionsopratoren „+" und „," lediglich die Menge generieren, auf der die Selektionsstrategien angewendet werden. In den späteren Anwendungen der EAs wird eine *Fitness-proportionale* Selektionsstrategie verwendet. Das bedeutet, dass die Individuen mit den μ besten Fitnesswerten für die neue Population selektiert werden. Eine andere mögliche Strategie ist beispielsweise die „Q-stufige Turnierselektion" [Wei07].

Selektionsstrategien

MUTATION Während der Mutation werden neue Individuen erzeugt. Diese basieren auf den besten Individuen der aktuellen Population. Mit Hilfe eines Mutationsoperators werden die *Genome* der neuen Individuen bestimmt. In späteren Kapiteln wird hierzu eine Zufallszahl bei reellem *Genotyp* genutzt. In diesem Fall berechnet sich der neue Wert des *Genom* als Addition aus dem alten Wert und einer Zufallszahl, die aus einer Normalverteilung um 0 gezogen wird:

$$x + s \cdot \mathcal{N}(0, \sigma^2), \tag{2.1}$$

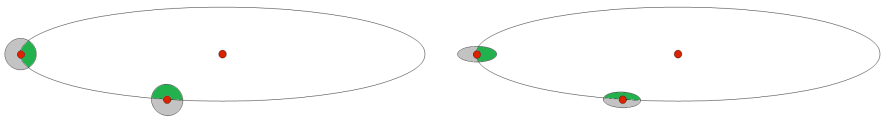
mit x als altem Wert, der Standardabweichung σ und der Schrittweite s. Letztere ist ein sogenannter „Strategieparameter" des EA. Sie bewirkt, dass bei einer kleineren Schrittweite die Werte nicht zu stark variieren. Während der Ausführung des Algorithmus wird die Schrittweite dynamisch angepasst. Hierfür existieren verschiedene Verfahren. Sowohl Rechenberg ($\frac{1}{5}$-Regel) als auch Schwefel (selbstadaptiv) haben dazu Regeln aufgestellt [Wei07].

Strategieparameter

2.2.2 *CMA-ES*

Ein weiterer verwendeter Algorithmus ist die Covariance Matrix Adaptation Evolution Strategy (CMA-ES). Dabei handelt sich dabei um einen modifizierten EA. Der Vorteil einer CMA-ES liegt im verbesserten Mutationsoperator. Bei einem mehrdimensionalem Optimierungsproblem (es werden mehrere Parameter optimiert) würde im Falle eines einfachen EA anhand von $\mathcal{N}(\boldsymbol{x}, \sigma^2 \cdot \boldsymbol{I}_n)$ mutiert werden (vgl. 2.2.1), während die neuen Individuen beim CMA-ES durch $\mathcal{N}(\boldsymbol{x}, \boldsymbol{C})$ aus

verbesserter Mutationsoperator



(a) Mutation beim EA (b) Mutation bei der CMA-ES

Abbildung 2.3: Vergleich der Mutation beim EA und der CMA-ES. Die grüne Färbung zeigt den Bereich, in dem eine Verbesserung erzielt wird, während der grau gefärbte Bereich eine Verschlechterung bedeuten würde.

dem jeweiligen Individuum $\boldsymbol{x}$ generiert werden. $\boldsymbol{I}_n$ bezeichnet dabei die Einheitsmatrix mit Rang n und mutiert in einem rotationssymmetrischem Raum (vgl. Abb. 2.3a) um $\boldsymbol{x}$, während $\boldsymbol{C}$ die Kovarianzmatrix mit Rang n ist, die den Raum, in dem mutiert wird, mit jeder Iteration weiter an den Suchraum anpasst (vgl. Abb. 2.3b) [Rud92]. Falls kein Vorwissen über den Suchraum vorhanden ist, existieren auch noch keine guten Varianzen für einzelne Dimensionen. In diesem Fall kann für jede Dimension dieselbe Varianz gewählt werden. Die Kovarianzmatrix wird dann durch die Einheitsmatrix initialisiert. Der Vorteil der Kovarianzmatrix ist, dass sie mit jeder Iteration weiter an den Suchraum angepasst wird. Das geschieht durch eine Neuberechnung der Kovarianzmatrix anhand der Lage und den Fitnesswerten der neu erzeugten Individuen. Der Vorteil, die Mutation basierend auf der Kovarianzmatrix durchzuführen, ist, dass mit einer höheren Wahrscheinlichkeit eine Verbesserung gefunden wird (vgl. Abb. 2.3) [Rud92].

Kovarianzmatrix

3 WAHRNEHMUNG

Die korrekte Wahrnehmung der Umgebung ist eine der Grundvoraussetzungen, um erfolgreich in Spielen agieren zu können. Im Wesentlichen spielt hierbei das Kamerabild die wichtigste Rolle. Den größten Optimierungsbedarf hat hierbei zum einen die Wahl geeigneter Kameraparameter und zum anderen der Ausgleich von Ungenauigkeiten basierend auf dem Einbau der Kamera. Der erste Teil dieses Kapitels behandelt ein Verfahren zur Optimierung der Kameraparameter, der zweite Korrekturmaßnahmen, um Verschiebungen der Kamera auszugleichen.

3.1 OPTIMIERUNG DER KAMERAPARAMETER ZUR FARBKLASSIFIKATION

Die einzelnen Komponenten des Spielfeldes (Ball, Linien, Tore, etc.) werden von den Robotern nicht nur aufgrund ihrer Form unterschieden, sondern vor allem anhand ihrer Farbe. Somit ist die sichere Erkennung der einzelnen Farben eine wichtige Grundvoraussetzung für eine korrekte Wahrnehmung des Spielgeschehens. Während der Dauer der Projektgruppe wurden zwei Klassifikationsverfahren eingesetzt. Zum einen ein Verfahren basierend auf einer Color Look-Up Table (LUT) (vgl. [QCM03]), zum anderen ein Verfahren basierend auf einer Analyse des Farbhistogramms des Kamerabildes. Der Erfolg beider Verfahren hängt direkt von der Qualität des Kamerabildes ab, insbesondere von der Unterscheidbarkeit der Farben innerhalb des Kamerabildes. Es wird davon ausgegangen, dass Farben in einem Kamerabild besonders gut zu unterscheiden sind, wenn sie im zum Kamerabild zugehörigen Farbraum eine möglichst große Distanz aufweisen.

Im Folgenden wird ein semi-automatisches Verfahren vorgestellt, welches mit Hilfe der im Kapitel Grundlagen erläuterten Optimierungsalgorithmen die Kameraparameter optimiert. Die Qualität des resultierenden Kamerabildes wird mit Hilfe von Fitness-Funktionen bewertet, die große Distanzen der einzelnen Farben im Farbraum positiv bewerten (vgl. Abb.: 3.3c, 3.3d). Eingabe des Verfahrens sind hierbei von Hand markierte Flächen des Kamerabildes, die exemplarisch eine Farbe darstellen.

3.1.1 *Kameraparameter*

Die Kameraparameter definieren den Suchraum der eingesetzten Optimierungsalgorithmen. Ihre Einstellung bestimmt die Lage der einzelnen Farben im Farbraum. Die folgenden Parameter sind bei der Nao Version V3.x einstellbar.

BELICHTUNG bezeichnet die Lichtmenge, welche auf den Sensor der Kamera fällt. Da der Durchmesser der Blende nicht angepasst werden kann, bezeichnet die Belichtung hier im speziellen die Belichtungszeit. Eine hohe Belichtungszeit sorgt für sehr gute Ergebnisse bei statischen Szenen, indem Sie Intensität und Helligkeit erhöht sowie Bildrauschen reduziert. Leider ist eine zu große Belichtungszeit für dynamische Spielszenen ungeeignet, da sie zu Schlieren bei bewegten Objekten führt.

FARBSÄTTIGUNG beschreibt die Intensität einer bestimmten Farbe. Es gibt die Möglichkeit, die Intensität der Farben Blau und Rot zu verändern. Durch eine Anpassung der Intensität von Rot und Blau lässt sich ein Weißabgleich erzielen, da mit Hilfe der Anpassung der Intensität Unterschiede in der Beleuchtung ausgeglichen werden. Des weiteren lassen sich Fehler in der Wahrnehmung von Schatten minimieren.

FARBTON beschreibt die Möglichkeit den "puren" Farbton, unabhängig von Helligkeit und Sättigung zu verändern. In Farbräumen wie *Hue-Saturation-Intensity* (HSI) ist der Farbton eine der drei Charakteristiken, um eine bestimmte Farbe zu spezifizieren. In dem von uns verwendeten Optimierungsverfahren wird eine Anpassung des Farbtons nicht berücksichtigt. Geometrisch würde es lediglich die Farben im gleichen Abstand zueinander innerhalb des Farbraums rotieren und hätte keinen Einfluss auf die verwendeten Fitness-Funktionen

GAIN bezeichnet die Verstärkung des Kamerasignals. Die Anpassung des Gains ist ein Kompromiss zwischen dem negativen Effekt der Verstärkung des Rauschens und positiven Effekten auf Farbsättigung und Helligkeit des Kamerabildes

HELLIGKEIT ist ein Attribut der Farbwahrnehmung und beschreibt, wie hell oder dunkel ein Körper erscheint. Dieser Parameter ist kein Hardwareparameter, sondern eine Korrektur von Über-/Unterbelichtung durch Anpassung der Mitteltöne im Bild. Da es sich nicht um einen Hardwareparameter handelt, wird dieser Parameter ignoriert.

KONTRAST bezeichnet den Abstand zwischen dem hellsten Punkt und dem dunkelsten Punkt im Bild. Genau wie die Helligkeit ist dies kein Hardwareparameter und wird daher nicht optimiert.

3.1.2 *Farbräume*

Die Wahl des Farbraumes hat einen großen Einfluss auf die Optimierung. Die einzelnen Farben haben je nach Farbraum einen unterschiedlich großen Abstand zueinander. Obwohl das Kamerabild stets für den nativen Farbraum der Kamera (YUV) optimiert werden muss, ist dies nicht notwendigerweise der beste Farbraum für die Optimierung. Es wurden drei verschiedene Farbräume (vgl. [FR94]), die den Definitionsbereich und das Ergebnis der Fitness-Funktionen auf unterschiedliche Weise beeinflussen, getestet.

YUV ist ein Farbraum, der ursprünglich entwickelt wurde, um die Farbinformation von einem Schwarz-Weiß-Signal zu trennen. Es wurde bspw. in Standards zur Übertragung analogen Fernsehens (PAL, NTSC) verwendet. Außerdem kann das menschliche Auge grundsätzlich besser zwischen verschiedenen Helligkeiten als zwischen verschiedenen Farbtönen unterscheiden. Diesen Effekt nutzt YUV, um Bandbreite bei Übertragungen einzusparen. Die drei Komponenten bezeichenen die Helligkeit Y (Luma) und zwei Komponenten des Farbtons UV (Chroma).

YUV ist der native Farbraum der NAO Kamera. Aufgrund mangelnder Rechenresourcen ist eine Konversion in andere Farbräume während des Spiels nicht möglich.

RGB ist ein sehr verbreiteter Farbraum. Eine Farbe wird in ihm als Kombination der drei Basisfarben Rot *(R)*, Grün *(G)* und Blau *(B)* repräsentiert. Der Wert der jeweiligen Farbe bestimmt ihre Intensität. Alle anderen Farben werden durch additives Mischen der Basisfarben gebildet, Weiß würde bspw. als Kombination des Maximums aller drei Farben dargestellt.

HSI beschreibt Farben mit Hilfe der drei Komponenten Farbton (*Hue*), Sättigung *(Saturation)* und Intensität *(Intensity)*. Geometrisch ist der Farbraum als Zylinder beschrieben. Der Farbton wird durch einen Winkel auf dem Farbkreis repräsentiert. Die Höhe des Zylinders beschreibt die Intensität und die Sättigung wird durch den Abstand zum Mittelpunkt beschrieben.

3.1.3 *Fitness-Funktionen*

Zur Bewertung der Individuen während der Optimierung werden zwei verschiedene Fitness-Funktionen genutzt. Eingabe der Fitness-Funktionen sind jeweils n Cluster von Punkten $\mathfrak{X}_1 \dots \mathfrak{X}_n$ eines bestimmten Farbraumes. Jeder Cluster besteht aus Punkten einer bestimmten Farbe. Jeder einzelne Punkt wird repräsentiert durch einen dreidimensionalen Vektor $\boldsymbol{X}_i = (c_1^i, c_2^i, c_3^i)$. Grundidee der Fitness-

Funktionen ist, dass Farben mit paarweise größerer Distanz leichter zu klassifizieren sind.

Die erste Fitness-Funktion f_1 berechnet aus den Clustern zunächst einen Durchschnittswert (Mittelpunkt) für jede Farbe. Anschließend wird die euklidische Distanz für jedes Paar von Durchschnittswerten berechnet und die Summe aus allen Distanzen gebildet. Da die Summe mit durchschnittlich größeren Distanzen zwischen den Farben streng monoton wächst, bevorzugt diese Fitness-Funktion Individuen (Kameraparameter), die zu weit im Farbraum auseinanderliegenden Farben führen.

$$f_1(X_1 \dots X_n) = \sum_{i=1}^{n-1} \sum_{j=i}^{n} \left| \text{avg}(X_i) - \text{avg}(X_j) \right|, \tag{3.1}$$

$$\text{mit } \text{avg}(X) = \frac{\sum_{i=1}^{m} \boldsymbol{X}_i}{m} \tag{3.2}$$

Die zweite Fitness-Funktion unserer Optimierung f_2 soll sicherstellen, dass alle Farben paarweise unterschieden werden können. Sie betrachtet daher den paarweisen Minimalabstand aller Farben und nicht deren Summe. Hierdurch wird schrittweise der derzeitig kleinste Abstand maximiert.

$$f_2(X_1 \dots X_n) = \min(\text{dist}(i,j)),\ i \neq j, \tag{3.3}$$

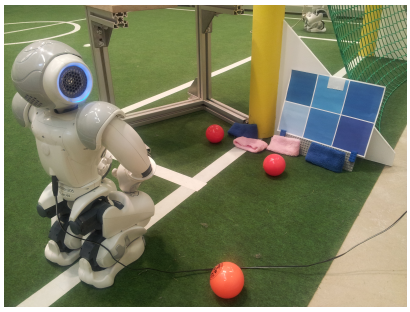
$$\text{mit } \text{dist}(i,j) = \left| \text{avg}(X_i) - \text{avg}(X_j) \right| \tag{3.4}$$

Aufgrund begrenzter Rechenleistung der Roboter sind andere Ansätze, wie z.B. die Bestimmung des „closest pair“ der beiden Punktwolken, leider nicht durchführbar.

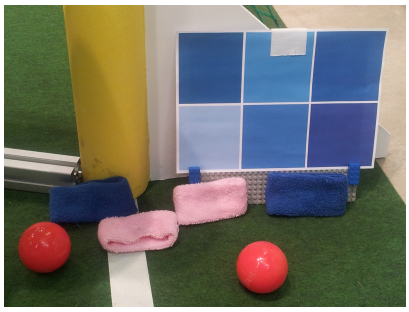
3.1.4 *Aufbau*

Als Versuchsaufbau wird eine statische Szene verwendet, bei der ein Roboter vor Beispielen der einzelnen zu unterscheidenden Farben platziert wird (Abb.: 3.1a). Die Beispiele der einzelnen Farben werden im Kamerabild des Roboters (Abb.: 3.1b) markiert. Aus den markierten Bereichen werden die Farbcluster, die die Eingabe der Fitness Funktionen sind, extrahiert.

Auf dem Roboter wird nun ein Programm gestartet, das iterativ mit Hilfe der Optimierungsalgorithmen die Kameraparameter ändert. Nach einer Wartezeit von ca. 2 Sekunden zur Übernahme der Kameraparameter durch die Hardware werden erneut Farbcluster aus den markierten Bereichen extrahiert, wobei sich das Kamerabild entsprechend der Kameraparameter geändert hat und sich ein neuer Fitnesswert ergibt.



(a) Roboter vor Farbbeispielen



(b) Kamerabild des Roboters

Abbildung 3.1: Setup

Auf diese Weise werden 54 verschiedene Testkombinationen aus 3 verschiedenen Optimierungsalgorithmen mit jeweils 3 unterschiedlichen Parametrisierungen, 3 Farbräumen und 2 Fitness Funktionen getestet. Um eine große Zeitersparnis gegenüber der Kalibrierung von Hand zu erzielen, ist ein Durchlauf auf 500 Aktualisierungen der Kameraparameter begrenzt.

3.1.5 *Anwendung*

Um mit dem aktuellen Framework kompatibel zu bleiben, ist die Benutzung der Optimierung in den vorhanden Simulator, *SimRobot*, integriert.

Der erste Schritt der praktischen Anwendung besteht daraus, per SimRobot eine Verbindung zu dem Roboter aufzubauen und im Kamerabild des Roboters die Farben, anhand derer optimiert werden soll, zu markieren (vgl. Abb.: 3.1b).

Dazu sollte zunächst die zu diesem Zweck erstellte Simulatorszene *RemoteRobotCamSetOpt* geladen werden, die sich im Ordner *Config/Scenes/* befindet.

Zur einfachen Benutzung des Verfahrens sind nun einige neue Konsolenbefehle in SimRobot integriert, die hier kurz beschrieben werden sollen.

- **cc on | off.** Schaltet das Markieren von Bildbereichen ein beziehungsweise aus.
- **cc <color>.** Stellt die zu markierende Farbe ein.
- **cc undo.** Macht die vorherige Aktion rückgängig.
- **cc send.** Macht dem Roboter die aktuellen Markierungen aller Farben bekannt.
- **cc clear.** Löscht alle bisherigen Markierungen.

Die Farben werden markiert, in dem der Benutzer mit der Maus ein Rechteck in dem Kamerabild des Roboters zieht. Die so markierten

Pixelpositionen werden automatisch lokal gespeichert und sind nun der zuvor gewählten Farbe zugeordnet.

Für einen Überblick über die bisherigen Markierungen ist eine Zeichnung über das Kamerabild eingebaut, die mit dem Befehl
vid image <Bildname> representation:ColorCluster [on | off]
an- und ausgeschaltet werden kann. Wenn alle Farben wie gewünscht markiert und dem Roboter die Markierungen bekannt gemacht worden sind, kann zum nächsten Schritt übergegangen werden.

Als zweiter Schritt kann jetzt optional zwischen verschiedenen Optimierungsalgorithmen, Fitnessfunktionen und Farbräumen gewählt werden, voreingestellt sind der $(3+9)$-EA, die Fitnessfunktion f_1 und der YUV-Farbraum.

Die Auswahl des Optimierungsalgorithmus erfolgt mit dem Befehl *set module:CameraSettingsOptimizer:optimizationAlgorithm <Nummer des Algorithmus>*.

Der Benutzer hat die Wahl zwischen folgenden Algorithmen:

- 0 : $(3+9)$-EA,
- 1 : Nelder-Mead-Simplex
- 2 : und CMA-ES .

Die verfügbaren Fitnessfunktionen werden mit
set module:CameraSettingsOptimizer:optimizationAlgorithm <Nummer der Fitnessfunktion>
gesetzt, wobei *0* für Fitnessfunktion f_1 steht und *1* für Fitnessfunktion f_2.

Über den Befehl
set module:CameraSettingsOptimizer:maxSteps <Anzahl>
kann die maximale Anzahl an Schritten innerhalb des jeweiligen Algorithmus begrenzt werden, bevor dieser Abbricht. Ein Schritt ist dabei eine Fitnessfunktionsauswertung und voreingestellt ist ein Wert von 500.

Für den $(\mu+\lambda)$-EA kann zusätzlich die initiale Schrittweite und die Varianz eingestellt werden. Die Schrittweite wird mit dem Befehl
set module:CameraSettingsOptimizer:stepSize <Schrittgröße>
verändert (der Standardwert ist 20,0), die Varianz mit dem Befehl
set module:CameraSettingsOptimizer:eaVariance <Varianz>,
der Standardwert hier ist 0,01.

Die Optimierung an sich kann schließlich mit dem Konsolenbefehl
set module:CameraSettingsOptimizer:optimize true|false
an- und ausgeschaltet werden.

Der Verlauf der Optimierung kann in dem *Scene Graph* unter *views* in der 3D-Darstellung *ColorCluster* (siehe Abb.: 3.3d) verfolgt werden. Außerdem verfügbar sind Plots über die besten Fitnesswerte (*bestFitness*) und die neuesten Fitnesswerte (*actualFitness*) im Plot-Unterbaum in dem *Scene Graph*.

3.1.5.1 *TestBench*

Die TestBench ermöglicht einen automatisierten Ablauf der Optimierung und besteht aus einer oder mehreren Kombinationen von Algorithmen, Farbräumen und Fitnessfunktionen (im folgenden bezeichnet als Konfigurationen) und wurde entwickelt, um diese Konfigurationen zu testen und vergleichbar zu machen. Die Ergebnisse der TestBench werden in CSV-Dateien abgespeichert und können mit dem dafür vorgesehenen Python-Skript gezeichnet werden. Die Namen der CSV-Dateien ergeben sich dabei aus dem Namen des Algorithmus, der Nummer der Fitnessfunktion, der Nummer des Farbraumes und der Nummer des Testdurchlaufs. Das Skript und eine kurze Beschreibung zu diesem findet sich im Ordner *Util/cameraSettingsPlotter/*.

Die TestBench wird wie auch die Optimierung in SimRobot gestartet. Dazu muss sie zuerst mit
dr module:CameraSettingsOptimizer:initTestBench once
initialisiert werden, dann kann sie mit
dr module:CameraSettingsOptimizer:startTestBench once
gestartet werden.

Falls die TestBench unterbrochen werden muss, kann sie später mit dem Befehl
dr module:CameraSettingsOptimizer:loadTestBench once
geladen werden und wird ab dem zuletzt beendeten Durchlauf eines Algorithmus weitergeführt.

Verfügbar und auch voreingestellt für die TestBench sind die Algorithmen

- $(1+1)$-EA,
- $(3+9)$-EA,
- $(6+9)$-EA,
- Nelder-Mead-Simplex mit Strategieparameter 6 und Abbruchschranke $1/100$,
- Nelder-Mead-Simplex mit Strategieparameter 6 und Abbruchschranke $3/100$,
- Nelder-Mead-Simplex mit Strategieparameter 3 und Abbruchschranke $3/100$,
- CMA-ES mit $\lambda = 5$,
- CMA-ES mit $\lambda = 9$ und
- CMA-ES mit $\lambda = 13$.

Außerdem sind beide Fitnessfunktionen und die drei beschriebenen Farbräume verfügbar und auch voreingestellt.

Falls der Benutzer eine andere Konfiguration nutzen will, kann er mit dem Befehl
dr module:CameraSettingsOptimizer:clearLists once
die Voreinstellungen löschen und mit
dr module:CameraSettingsOptimizer:add:<Alg./Fitnessfunktion/Farbraum> once
die gewünschten Teile der Optimierung hinzufügen um beispielsweise nur einzelne Algorithmen, Fitnessfunktionen oder Farbräume zu testen.

Eine weitere Voreinstellung ist die Anzahl der Durchläufe der verschiedenen Konfigurationen die mit
set module:CameraSettingsOptimizer:noRuns <Anzahl>
geändert werden kann, Standardwert ist hier 10. Für jeden Durchlauf mit einer Fitnessfunktion zu Farbraum Kombination erhält jeder Algorithmus die gleichen, zufällig gewählten, Startkameraparameter um eine gute Vergleichbarkeit zu erzielen.

3.1.6 *Evaluation*

Die Evaluation hat gezeigt, dass jeder eingesetzte Algorithmus über Vor- und Nachteile verfügt. Grundsätzlich haben die Algorithmen die besten Ergebnisse, die die Belichtungszeit maximiert haben. Zum einen liefern sie die höchsten Fitness Werte, zum anderen konvergieren sie sehr häufig. Außerdem ist es sehr wichtig den Gain auf den mittleren Bereich seines Definitionsbereichs zu stellen. Abgesehen von den Fitnesswerten ist es für die Varianz der Fitness Werte zwischen den Iterationen sehr wichtig, dass diese beiden Parameter stabil bleiben (vgl. Abb.: 3.2d).

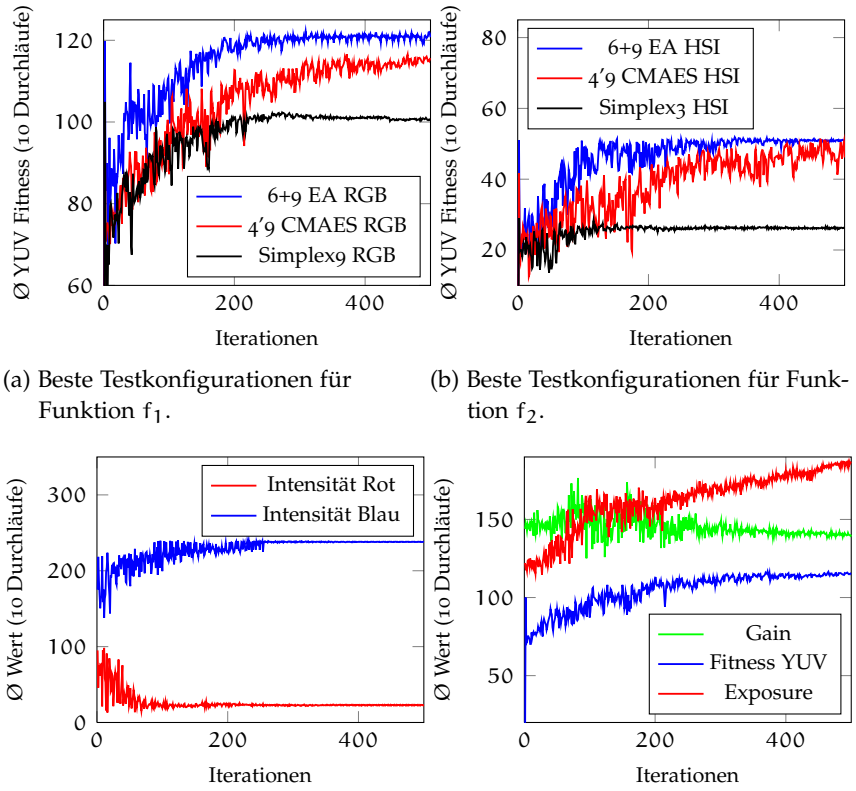
Die Farbsättigung sollte für die Unterscheidbarkeit der Farben stets maximal sein.

Jede der getesteten Kombinationen führte zu einer Anpassung der Farbintensitäten gegenüber den von Hand optimierten Werten, wobei die Intensität von Blau stets deutlich höher war, als die von Rot (vgl. Abb.: 3.2c).

Unabhängig von der Wahl des Algorithmus und der Fitness Funktion hat die Wahl des Farbraumes einen starken Einfluss auf den maximal erreichten Fitness Wert. Bei Fitness Funktion f_1 ist es von Vorteil während der Optimierung RGB zu verwenden (vgl. Abb.: 3.2a), obwohl die abschließende Bewertung stets in YUV gemessen wird. Bei Fitness Funktion f_2 liefert der HSI Farbraum die besten Ergebnisse (vgl. Abb. 3.2c). Damit liefert der native Farbraum der Kamera (YUV) die schlechtesten Ergebnisse.

Der (6+9)-EA erzielte bei beiden Fitness Funktionen die besten Resultate. Allerdings ist der (μ, λ)-CMAES im Vergleich zum (6+9)-EA das robustere Verfahren und konvergiert häufiger zu guten Fitness-

werten. Der Simplex-Algorithmus liefert stets die schlechtesten Werte.



(a) Beste Testkonfigurationen für Funktion f_1.

(b) Beste Testkonfigurationen für Funktion f_2.

(c) Optimierung der Intensitäten von Rot und Blau.

(d) Belichtung und Gain im Vergleich zum Fitnesswert.

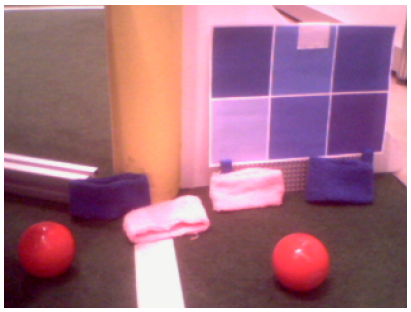
Abbildung 3.2: Beispiele der Optimierung.

3.1.7 *Fazit und Ausblick*

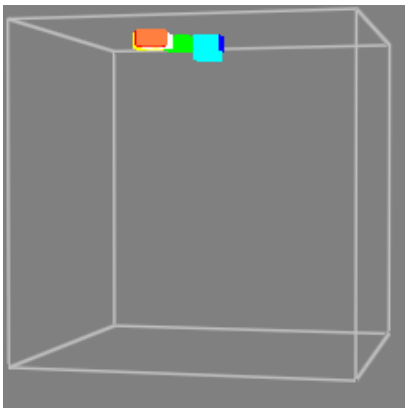
Die vorgestellte Optimierungsstrategie ist in der Lage Kameraparameter zu finden, die für die Klassifikation von Farben innerhalb des Kamerabildes von Vorteil sind. Es kann gezeigt werden, dass mit Hilfe der eingesetzten Fitness-Funktionen und Algorithmen ausgehend von randomisierten Kameraparametern deutlich bessere Einstellungen gefunden werden (Abb.: 3.3). Man erkennt deutlich die Verbesserung zwischen dem Kamerabild vor der Optimierung (Abb.: 3.3a) und dem optimierten Bild (Abb.: 3.3b). Die Abstände der einzelnen Farben werden ebenfalls signifikant vergrößert (vgl. Abb.: 3.3c, 3.3d).

Wie bereits in der Evaluation beschrieben, konvergiert die Intensität von Blau stets zu hohen Werten, während die Intensität von Rot eher niedrig gewählt wird. Dies liegt darin begründet, dass das Optimierungsverfahren einen Weißabgleich vornimmt, der das warme Licht der Deckenlampen ausgleicht.

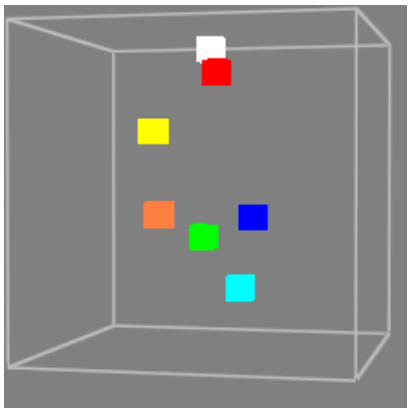
(a) Kamerabild vor der Optimierung



(b) Optimiertes Kamerabild



(c) Farbverteilung vor der Optimierung



(d) Optimierte Farbverteilung

Abbildung 3.3: Optimierung der Kameraparameter

Hauptproblem des eingesetzten Verfahrens ist die hohe Dauer der Optimierung. Während Wettkämpfen ist die Zeit für die Optimierung sehr begrenzt. Obwohl dieses Verfahren zwar die Kalibrierung der Kamera deutlich beschleunigt, wäre ein noch schnelleres Verfahren von großem Vorteil. Hauptproblem ist hierbei die Hardware. Die Kamera benötigt ca. 2 Sekunden um ein neues Parameterset zu übernehmen und daher wird eine Verbesserung der Geschwindigkeit kaum mehr möglich sein.

Wie bereits am Beispiel der Belichtung beschrieben, sind gute Parameter für statische Szenen zudem nicht immer auch für dynamische Szenen optimal. Leider ist eine Optimierung dynamischer Szenen mit dem verfolgten Ansatz jedoch nicht zu realisieren.

3.2 AUTOMATISCHE KALIBRIERUNG DER KAMERAMATRIX

3.2.1 *Motivation*

Neben Problemen, die sich durch die Variabilität der Umwelt ergeben, existieren auch Probleme aus der Produktion. Der verwendete Roboter NAO verfügt über zwei Kameras im Kopf. Der Einbau die-

ser entspricht bei der Mehrzahl unserer Roboter nicht exakt den vom Hersteller angegebenen Daten. Hinzu kommt noch eine leichte Variabilität, welche durch das Verwenden von Heißkleber im Produktionsprozess entstehen kann. Gerade in der Wahrnehmung spielt der genaue Einbau der Kamera eine entscheidende Rolle. Ist diese bereits leicht rotiert oder geneigt, so ist das Kamerabild - in diesem Fall die gesamte Wahrnehmung der Außenwelt - um genau diesen Faktor nicht stimmig.

Um diese Ungenauigkeit und die daraus permanente Verzerrung der Umwelt zu reduzieren, gibt es bereits im Framework die Möglichkeit durch bestimmen von Offsets auf die Lage der Kamera im Roboter diesen Fehler entgegenzuwirken. Mit Hilfe dieser werden so eventuelle Rotations- oder Translationsfehler korrigiert. Bisher war zum Einstellen dieser Offsets eine manuelle Kalibrierung vorgesehen. Dazu wird der Roboter meist auf Höhe der Mittellinie an der Seitenauslinie platziert, um dann durch betrachten des Kamerabildes, im Vergleich zur Wirklichkeit, die Offsets richtig zu kalibrieren. Hierzu gibt es eine Szene (`/Config/Scenes/CameraCalibration.con`), welche ein gedachtes Spielfeld als Overlay dem Kamerabild hinzufügt. Durch diese Einzeichnung lassen sich jetzt die Linien des Gedachten Spielfeldes mit denen des Kamerabildes angleichen (Abb.: 3.6a).

Ziel der Projektgruppe ist es eine vollautomatische Kalibrierung zu erstellen, um die manuelle zu ersetzen. Dazu sollte der Roboter bestimmte Punkte auf dem Spielfeld identifizieren und mit seiner Erwartung über die Umwelt und dessen Lage abgleichen können. Aus deren Abweichung heraus werden möglichst naheliegende Offsets berechnet, welche demnach zu einer verbesserten Wahrnehmung führen.

Durch diese Korrektur bei der Wahrnehmung, welche ein genaueres Kamerabild zur Folge hat und zweifellos zu einer weniger verzerrten Sicht auf die Umwelt führt, wird z.B. ein präziseres Anspielen des Balles ermöglicht.

3.2.2 *Orientierungspunkte*

Um eine Optimierung durchführen zu können, werden Anhaltspunkte aus der Umgebung des Roboters benötigt. Hinzu kommt die Vorgabe, dass diese Anhaltspunkte nach Möglichkeit lokal begrenzt, sowie flexibel sein sollen. Ein ganzes Spielfeld als Beispiel kommt hierfür nicht in Frage, da bei Teilnahmen an Turnieren wie dem RoboCup 2012 (in Mexico-City) (RoboCup 2012) dies immer nur eingeschränkt zur Verfügung steht. Es mussten also einzelne Punkte, in einem möglichst geringen Umkreis, welche zudem variabel sind, verwendet werden. Eine Möglichkeit besteht darin den Roboter auf unterschiedliche Punkte auf dem Spielfeld bzw. drei Punkte innerhalb einer Szene schauen zu lassen, was das Mindestmaß für eine voll-

ständige Kalibrierung ist. Der Nutzer würde dann im Kamerabild bestimmte Punkte per Mausklick markieren. Diese Punkte würden als Vergleichspunkte für eine Optimierung weiterverwendet werden. Nachteil dieses Verfahrens ist, dass zu jedem markierten Punkt das exakte Pendant in der realen Welt gefunden werden, muss um eine einwandfreie Kalibrierung zu gewährleisten.

Eine andere Möglichkeit besteht darin, statt diese Punkte aus einer Szene zu abstrahieren, diese Szene einfach direkt zu stellen, z.B. durch Marker auf dem Spielfeld. Hier ergeben sich natürlich Probleme mit der Platzierung, sowie durch Behinderungen anderer Objekte.

Die Idee bei dem Lösungsansatz, den die Projektgruppe hier wählt, ist die Verwendung eines Schachfeldmusters. Durch dieses lässt sich bei der Erkennung der einzelnen Kreuzungspunkte eine genauere Information ableiten, welche dann wieder Rückschlüsse auf die Korrektur der Kamera-Matrix erlaubt. Zudem sind Schachfelder von der Handhabung sehr einfach und sie lassen sich überall (z.B. Boden, Wand) anbringen. Die Erkennung der Kreuzungspunkte wird mittels einer bereits schon existierenden Bibliothek Open Source Computer Vision (OpenCV) von Statten gehen. Zusätzlich bietet diese auch Funktionen der Kamerakalibrierung an. Im optimalen Fall lässt sich so das Schachbrett als Raster für die Kamera nutzen.

3.2.3 *Hauptteil*

Die Bestandteile der Eingaben für die Optimierung sind hierbei die Aufnahmen der Schachfelder, so wie deren Koordinaten aus Sicht des Roboters. Mithilfe von OpenCV lässt sich im aufgenommenen Bild ein Schachfeldmuster finden, sowie die einzelnen Kreuzungspunkte ermitteln (Abb.: 3.4a). Somit werden Informationen, die als Input der Vision, mit der der Roboter seine Umwelt wahrnimmt, gezielt und eindeutig erkannt. Durch die Möglichkeit und das Wissen über die Platzierung der Schachfelder ist genaueres Wissen über die wirkliche Welt, welche den Roboter umgibt, vorhanden.

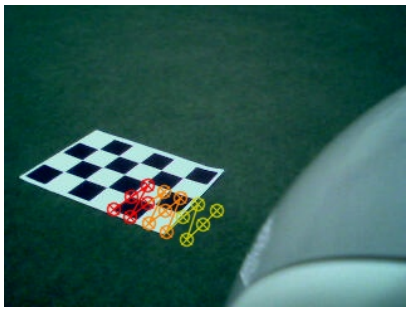
Die Basis der Optimierung ist der Vergleich zwischen diesen beiden Welten. Mithilfe einer Funktion, die die Distanz der einzelnen Punkte zwischen Robotersicht und der wirklichen Welt berechnet, wird ein Maß für die Abweichung der Welten gegeben. Um dies zu veranschaulichen, lassen sich die Punkte für das Schachfeld in das Kamerabild des Roboters einzeichnen (Abb.: 3.4).

3.2.3.1 *OpenCV*

Bevor die zugrundeliegenden Funktionen aus OpenCV und die daraus erstellte Funktionsreihenfolge genauer betrachtet werden, soll hier zunächst die OpenCV-Library kurz vorgestellt werden.

OpenCV ist ist eine Open Source C/C++ Bibliothek. Der vollständige Name lautet: „Intel Open Computer Vision Library“. Zunächst

(a) Die von OpenCV erkannten Schachkreuzungen



(b) Die Schachkreuzungen wie sie im Bild vermutet werden

Abbildung 3.4: Kalibration

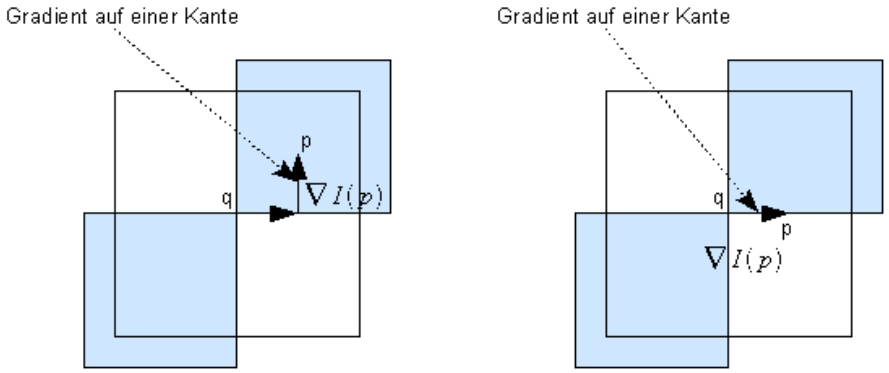
von Intel entwickelt, wird die Bibliothek heute quelloffen unter einer BSD-Lizenz weiterentwickelt. Die Bibliothek stellt komplexe Funktionen aus dem Bereich der Computer Vision zur Verfügung. So lässt sich OpenCV für Gesichtserkennung, Gestenerkennung oder dem Tracking von Bewegungen einsetzen. Ebenso finden sich Funktionen zur Segmentierung, Objekt-Identifikation und eine Reihe verschiedener Filter. Alle diese Funktionen zeichnet aus, dass diese auf aktuellen Forschungen und optimierten Implementationen der Algorithmen fußen.

3.2.3.2 *Ablauf der Kamerakalibrierung*

Die Kamerakalibrierung mit Hilfe von Schachfelderkennung läuft in drei Phasen ab. In der ersten Phase werden mit dem Roboter Bilder von verschiedene Blickwinkeln auf das Spielfeld aufgenommen. In jedem dieser Bilder ist ein Schachfeld platziert. Diese Schachfelder werden dann mit den dafür vorgesehenen OpenCV-Methoden erkannt. In Phase II werden die Schachfelder in das Feldkoordinatensystem abgebildet und mit einer vorher erstellten Feldmaske abgeglichen. Die Maske wird anhand der Koordinaten des Standpunktes des Roboters und der Entfernungen der Schachfelder zum Roboter individuell für jedes Schachfeldbild so erstellt, dass dieses die idealen Punkte vorgibt. Ziel der in Phase III durchgeführten Optimierung ist es, die Summe aller Fehler der einzelnen Vergleiche zwischen Schachfeld und Schachfeldmaske zu minimieren. Denn es wird angenommen, dass ein minimaler Fehler dazu führt, dass in jeder Blickrichtung die Abweichung der Ist-Position möglichst gering bzgl. der Soll-Position ist.

3.2.3.3 *Phase I*

Diese Phase behandelt die Aufnahme von Bildern und das Erkennen von Schachfeldern in diesen Bildern.
Die einzelnen Aufnahmen der Bilder werden mit der schon imple-



(a) p innerhalb eines Schachfeldes (b) p auf einer Kante

Abbildung 3.5: Lage eines erkannten Schachfeldeckpunktes (Abbildung nach [GBo8])

mentierten Log Funktion des Simulators gemacht und gespeichert. Dies geschieht unter Verwendung der OpenCV-Methode *cvSaveImage*. Sind alle benötigten Aufnahmen gesichert, kann mit der Erkennung der Schachfelder begonnen werden. Dazu werden die Bilder mit der Methode *cvLoadImage* in das Programm geladen und in eine *Struktur* integriert, welche alle weiteren Ground-Truth Informationen, wie z.B. die Lage des Schachfeldes relativ zum Roboter oder die Anzahl der Schachfelder innerhalb einer Zeile bzw. Spalte des Schachbrettes, enthält. Mit der OpenCV-Methode *cvFindChessboardCorners* wird ein Layout der Schachfeldeckpunkte erstellt, welches alle Bildkoordinaten dieser enthält. Nach der Konvertierung des Bildes in ein Grauwertbild, welches die anschließende Methode benötigt, wird die Genauigkeit der Schachfeldeckpunktkoordinaten - bekannt aus der Erkennung mit *cvFindChessboardCorners*- mit der Methode *cvFindCornerSubPix* erhöht. „Dies ist erforderlich, da es selten der Fall ist, dass eine Schachfeldecke genau im Zentrum eines Bildpixels liegt. *cvFindChessboardCorners* liefert allerdings nur diese Genauigkeit. Um die Subpixelgenauigkeit zu berechnen wird ein mathematischer Trick angewandt. Voraussetzung dabei ist, dass das Vektorprodukt zweier orthogonal zueinander stehender Vektoren gleich o ist. Falls der gefundene Punkt nicht mit der Schachfeldecke übereinstimmt, gibt es zwei Möglichkeiten wo dieser liegen kann. Innerhalb eines Schachfeldes oder auf einer Kante eines Schachfeldes. Bezeichne q den Punkt der Schachfeldecke und p den subpixelgenauen Punkt. Liegt p innerhalb eines Schachfeldes, hat der Gradient im Punkt p einen Wert von o, da der Punkt sich in einer gleichförmigen Umgebung befindet (Abbildung 3.5). Liegt der Punkt p auf einer Kante des Schachfeldes, so gibt es dort einen Sprung von weiß nach schwarz und der Gradient[Anmerkung des Autors: Gemeint ist hier der Sobeloperator] des Punktes p steht senkrecht zum Vektor $q - p$. In beiden Fällen ist das Vektorprodukt $< \nabla p, q - p >= 0$. Aus mehreren solcher Gradient-

Vektor-Paarung lässt sich ein Gleichungssystem erstellen, dessen Lösung zu einen genaueren Punkt p führt.“ [GA10] nach [GB08]

3.2.3.4 *Phase II*

In dieser Phase werden die Ist-Layouts mit den Soll-Layouts verglichen um die Abweichung zwischen den Layouts zu ermitteln. Die Soll-Layouts werden aufgrund der Informationen, der Schachfeldgröße und der Position des Schachfeldes relativ zum Roboter, erstellt. Allerdings sind die Koordinaten der Ist-Layouts in Bildkoordinaten und die der Soll-Layouts in Feldkoordinaten gegeben. Um diese vergleichbar zumachen, werden die Soll-Koordinaten in das Bildkoordinatensystem transformiert. Um Fehler beim Vergleich mit der Soll-Schachfeldmaske zu vermeiden, wird die Soll-Schachfeldmaske mittels der Methode *sortingEstimatedPoints* in eine einheitliche Reihenfolge gebracht. Dies ist notwendig, da die Methode *cvFindChessboardCorners* die Schachfelder immer in der Reihenfolge zeilenweise beginnend links oben erkennt. Dies führt zu einer unterschiedlichen Sortierung, wenn das Schachfeld längs bzw. quer zum Roboter liegt. Als Maß für den Fehler zwischen den Layouts wird der Euklidische Abstand verwendet.

3.2.3.5 *Phase III*

Die Optimierung wird mit einem (1+1)-EA durchgeführt. Als Fitnessfunktion dient der zuvor berechnete Abstand zwischen den Soll- und Ist-Layouts. Ist ein Individuum geeigneter als das zuvor berechnete, wird es gespeichert. Dabei besteht ein Individuum aus den Parametern: *BodyRoll*, *BodyTilt*, *CameraRoll*, *CameraTilt* und *HeadTilt*. Diese sind Offsets, welche Fehler aus dem Fertigungsprozess des NAOs ausgleichen sollen. Während der Optimierung wird die Kameramatrix, welche alle Informationen zur Berechnung des Roboterbildes enthält, mit den Parametern eines neu erzeugten Individuums rotiert und die Transformation der Soll-Layouts mit der rotierten Kameramatrix durchgeführt. Anhand dieser neu generierten Maske wird erneut die Fitnessfunktion ausgewertet. Dann beginnt die Optimierung mit der Erzeugung eines neuen Individuums von vorne. Das beste Individuum soll die Offsets enthalten, die am besten dazu geeignet sind die Produktionsfehler, die sich auf das Roboterbild auswirken, auszugleichen.

3.2.4 *Aufbau*

Im Grundsatz kann das Modul in sehr vielen Variationen genutzt werden.

Jedoch hat sich ein Standardaufbau etabliert, bei der der Roboter möglichst exakt auf das T-Linienkreuz der Seiten- und Mittellinie

platziert wird. Der NAO nimmt hierbei eine aufrecht stehende Haltung ein. Die genaue Haltung wird durch eine Ansteuerung in der Szene `CameraCalibration.con` vorgegeben (siehe Kapitel: 3.2.5) Sein Blickfeld ist dabei Richtung Spielfeld ausgerichtet und es sollte auf ein möglichst symmetrische Körperhaltung geachtet werden.

Die Schachfelder sollten möglichst groß ausgedruckt, z.B. auf DIN-A4 mit 6×4 Kästchen und nicht weiter als 1 m entfernt niedergelegt werden. Bei Entfernungen darüber hinaus kommt es zu Problemen mit der Erkennung des Schachfeldmusters. In dem verwendeten Code durch OPENCV können sich durch alleiniges nicht erkennen einer einzigen Schachfeldkreuzung große Probleme ergeben. Deshalb muss auch darauf geachtet werden, dass der Roboter das gesamte Schachfeld im Blick hat. Ansonsten können die Schachfelder frei auf dem Feld platziert werden. Diese Implementierung erlaubt es ebenfalls diese hochkant zu stellen. In der im Folgenden verwendeten Szene existieren darüber hinaus schon Werte für eine gültige Platzierung. Wegen der bereits erwähnten Problematik empfiehlt es sich diese vorerst beizubehalten.

3.2.5 *Softwarebenutzung*

Zur Benutzung des Moduls kann die Szene `CameraCalibration.con` verwendet werden. Die Nutzung dieser erlaubt eine einfache Vorbereitung sowie Durchführung der Kalibrierung durch fortlaufendes Drücken der *Enter*-Taste.

- Im ersten Block werden einige Anweisungen vorgegeben, die zum Arbeiten mit dem Roboter notwendig sind. So wird der Roboter angesprochen, das Kamerabild hinzugezogen und der Roboter in eine Stehhaltung gebracht.
- Darauf folgt einem Block der einen gewissen Konfigurationsspielraum lässt. So kann z.B. der Roboter an einer anderen Stelle platziert werden oder eine andere Größe für die Schachbretter vorgegeben werden.
- In den nächsten drei Blöcken werden jetzt jeweils die Bilder, sowie die zugehörigen Parameter eingespeist:
  Dazu erfolgt zunächst die Ausrichtung des Kopfes, sowie die Aufnahme des aktuellen Bildes unter `logs/cameraCalibration_direction`. Anschließend geben zwei zweidimensionale Vektoren die beiden Eckpunkte des Schachfeldes an. Zum Schluss wird das erzeugte Bild, sowie die dazugehörigen Daten zu eine Datenstruktur hinzugefügt.
- Wenn dies alles geschehen ist, kann die Optimierung gestartet werden. (Es existiert darüber hinaus noch ein Flag, welches die

Option bereitstellt, das Ergebnis jedes Optimierungsschritts in ein Bild einzuzeichnen.)

Die Optimierung läuft vollständig auf dem externen Rechner ab, so dass auf den Roboter verzichtet werden kann. Es wäre also denkbar, jeden Roboter kurz an die dafür vorgesehene Position zu stellen, mit ihm Bilder aufzunehmen und in einem zweiten Schritt, zu einem späteren Zeitpunkt die Optimierungen starten zu lassen. Dieses Vorgehen kann auf Veranstaltungen wie dem RoboCup 2012 von Vorteil sein. Wird bei der Optimierung eine Verkleinerung des Fehlers erreicht, so werden die aktuellen Parameter, deren Fitness, sowie das aktuelle Ergebnis im Bild eingezeichnet. Dadurch wird fortan die beste erstellte Kalibrierung festgehalten. Nach der Kalibrierung lassen sich dann die Werte der Korrektur-Matrix in die `jointCalibration.cfg` übertragen.

3.2.6 *Evaluation*

Betrachtet man die Ergebnisse der Optimierung, so lässt sich eine leichte Verringerung der Abweichung zu den Orientierungspunkten feststellen. Es fällt die exakt Auslegung der gedachten Mittellinie parallel neben der eigentlichen auf. Siehe dazu Abb: 3.6. Während sich die Abweichung zu Orientierungspunkten welche über 1 m entfernt sind verringert, gilt dies nicht für weiter vorne liegende Punkte. Hier nimmt die Abweichung enorm zu, verdoppelt sich sogar.

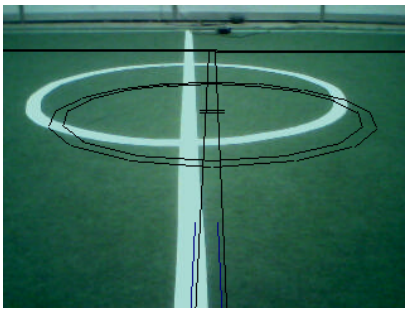
Entscheidend ist welche Auswirkung die Optimierung auf das reale Verhalten des Roboters haben wird. Der Roboter mag bessere Informationen über weit entfernte Objekte erhalten. Doch gerade der Nahbereich ist für das Spielgeschehen ausschlaggebend.

In der Annahme der Roboter soll zu einem 2 m entfernen Ball laufen und diesen Anspielen. So ist eine genauere Ballposition zwar zu begrüßen, jedoch bei weitem nicht so relevant. Insbesondere wenn man die Abweichungen beim Laufen selbst hinzuzieht (siehe Kapitel: 4.1). Je näher der Roboter dem Ball ist, desto wichtiger sind exakte Positionsangaben. Befindet sich nun der Roboter in unmittelbarer Ballnähe, so hat die Zunahme der Abweichung eine stark negative Auswirkung. Es wird zu mehr Fehlern beim Anspielen des Balles führen.

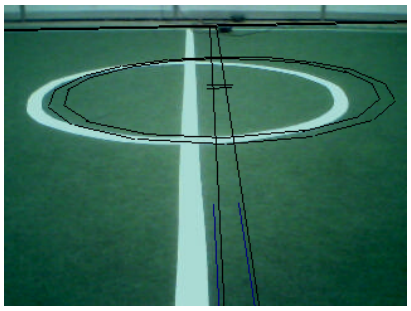
Darin sah die Projektgruppe einen gravierenden Nachteil bei den durchgeführten Optimierungen der Kameramatrix.

3.2.7 *Fazit*

Gegen Ende der Arbeit fiel - anhand von Kamerabildern - auf, dass die Abweichung der Kamera meist einem Winkel von 2-3 Grad auf der horizontalen entsprach. Dies traf auf Tests mit mehreren Robo-

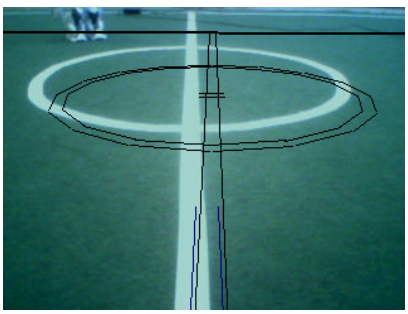


(a) Vor der Optimierung

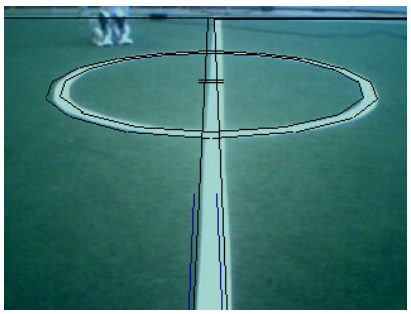


(b) Ergebnis der Optimierung

Abbildung 3.6: Ergebnis der Optimierung



(a) Ohne jegliche Korrektur



(b) Korrektur des Pan um 3 und des Tilt um 2 Grad

Abbildung 3.7: Standart Pan/Tilt-Korrektur

tern zu. Um dies zu verifizieren wurden die Köpfe nach links, geradeaus und rechts ausgerichtet und Bilder aufgezeichnet. Gerade beim Blick zur Seite kann gut verglichen werden zu welchem Anteil das Schulterpolster im Bild ist. Anhand dieses Merkmals wird dann der Offset für den *HeadPan* (die horizontale Kopfbewegung) auf einen passenden Winkel eingestellt. Ebenfalls ließ sich auch der *HeadTilt* (die vertikale Kopfbewegung) nach menschlichem Ermessen einstellen. Diese manuelle Korrektur allein erzielt eine erhebliche Verbesserung (Abb.: 3.7). (Dies ließ sich nur mit einigen Robotern testen, da zu diesem Zeitpunk die Hälfte der Roboter bereits zum Austausch gegen eine neuere Version eingeschickt worden war.)

Das nochmalige Ausführen einer Optimierung, mittels des neuen Offsets, bringt dagegen minimale Veränderungen. Und es bleibt fraglich wie sehr dadurch eine Verbesserung eintritt. Zu den Optimierungsalgorithmen lässt sich feststellen, das diese korrekt auf dem gegebenen Problem arbeiten. Es sind vielmehr die falschen Parameter die optimiert werden, wodurch das Ergebnis unbrauchbar wird.

Das Optimierungsverfahren ist dahin gehend umgeschrieben und bezieht jetzt sowohl den *HeadPan*, als auch den *HeadTilt* ein. Zu diesem Zeitpunkt wurden die NAOs V3.x gegen neuere Modelle des Roboters ausgetauscht. Bei diesen zeigte sich, dass beim Einbau der Kamera/des Kopf sorgfältiger gearbeitet wurde und meist nur eine sehr geringe Abweichungen festzustellen war. Es wurde bisher nicht weiter versucht diese zu optimieren.

4 LAUFOPTIMIERUNG

Als eines der Mindestziele, stellte die Laufoptimierung einen zentralen Punkt in der Arbeit der Projektgruppe dar. Ziel war es, den Lauf in Hinblick auf Geschwindigkeit und Genauigkeit zu verbessern. Dieses Kapitel beschreibt die drei unterschiedlichen Lösungsansätze, die von der Projektgruppe zur Verbesserung des Laufs untersucht wurden.
In allen Vorexperimenten konnte beobachtet werden, dass der NAO bei einem gewünschten Geradeauslaufen eine Linkskurve einschlug. Diese Eigenart galt für alle Roboter, unabhängig von der Version. Das erste Unterkapitel beschreibt den Ansatz diesen Fehler zu messen und während des Laufs stetig zu korrigieren.
Im Verlauf der Projektgruppe wurde festgestellt, dass ein Entgegenwirken dieser Laufungenauigkeit nicht so effektiv ist, wie deren Ursache zu beseitigen. Zu diesem Zweck wurde eine automatische Optimierung der Laufparameter untersucht, welche im zweiten Unterkapitel genauer beschrieben wird.
Als vielversprechendste der drei Lösungen zeigte sich die automatische Kalibrierung der Gelenke, welche im dritten Unterkapitel erläutert wird.

4.1 ODOMETRIEKORREKTUR

Dieses Kapitel beschäftigt sich mit einem Lösungsansatz zur Behebung des fehlerhaften Geradeauslaufens, der Odometriekorrektur.
Im ersten Unterkapitel werden zunächst die Grundlagen der Odometrie erläutert, insbesondere in Hinblick auf den Unterschied zwischen Odometrie bei Robotern mit Rädern und humanoiden Robotern.
In den folgenden Abschnitten wird beschrieben, wie eine Odometriekorrektur für den NAO realisiert werden kann. Zuerst werden Vorexperimente beschrieben, aus dessen Beobachtungen die Realisierung einer Odometriekorrektur abgeleitet werden soll. Im nächsten Abschnitt wird die Aufgabenstellung der Odometriekorrektur skizziert und anschließend der Versuchsaufbau erläutert, sowie die Resultate und Beobachtungen aus diesen Versuchen verbildlicht.
Der letzte Abschnitt diskutiert die Relevanz und Eignung der Vorgestellten Odometriekorrektur in Bezug zur automatischen Kalibrierung der Walking-Engine-Parameter und der Kalibrierung der Gelenkwinkel-Offsets.

4.1.1 *Odometrie*

Der Name Odometrie stammt aus dem griechischen und bedeutet etwa so viel wie *Wegmessung*. Es bezeichnet die Lokalisierung des Roboters an Hand eigener Sensormessungen. Bei Robotern, die sich mit Rädern fortbewegen, ist die Odometrie recht einfach über die Umdrehungen der Räder zu bestimmen: Abhängig von Durchmesser der Räder und deren Umdrehungen lässt sich sowohl die zurückgelegte Strecke, als auch die Drehung des Roboters bestimmen, die während einer Fahrt aufgetreten ist. Diese Messwerte sind bei heutigen Robotern mit Rädern sehr genau.
Im Gegensatz hierzu gestaltet sich die Odometrie bei humanoiden Robotern ein wenig schwieriger. Grundsätzlich stehen zur Messung erst einmal nur die jeweils angesteuerten Gelenkwinkel zur Verfügung. An Hand dieser Gelenkwinkel muss zunächst die Position der Füße im Raum bestimmt werden. Dieses Problem nennt sich *Vorwärtskinematik*. Es behandelt das Problem zu einem gegebenen Roboter mit bekannten Abmessungen eine Abbildung zu finden, die vom Gelenkwinkelraum in den Arbeitsraum abbildet, also zu einer gegeben Gelenkwinkelkonfiguration die Position und Rotation des Endeffektors zu bestimmen. Die Endeffektoren sind im Fall des NAOs die beiden Füße. Das Problem der Vorwärtskinematik ist bereits gut untersucht. Mit Hilfe von homogenen Transformationen lässt sich die Position im Arbeitsraum einfach bestimmen.

Vorwärtskinematik

Ist zu den aktuellen Gelenkwinkeln die Fußposition bestimmt, lässt sich der gelaufene Weg eines humanoiden Roboters nachvollziehen und es kann hierüber die Odometrie bestimmt werden. Auf diese Weise bestimmt auch der NAO seine aktuelle Odometrie.
Es treten bei dieser Art der Odometrieberechnung jedoch systematische Fehler auf. So ist es praktisch unvermeidbar, dass der NAO beim Aufsetzen eines Fußes nicht mit der ganzen Sohle, sondern mit der Fußspitze oder der Hacke zuerst Bodenkontakt herstellt. Hierbei rutscht der NAO. Dieses Rutschen wird von der Odometrie nicht erfasst und resultiert in einem Positions- und Rotationsfehler. Ein ähnliches Problem stellt sich, wenn der Roboter vom normalen Teppichboden auf eine Feldlinie tritt. Die Feldlinien weisen wesentlich geringere Reibung auf als der Teppich und auch hier kommt es häufig zu einem Rutschen des Roboters. Die Odometrie wird zudem von der Bauweise des NAOs beeinflusst. Da die Gewichtsverteilung im Roboter asymmetrisch ist, neigt er dazu beim Laufen stärker eine Richtung zu schwingen als in die andere. Hierdurch rotiert der Roboter in Richtung des Schwerpunkts, was ebenfalls nicht von der aktuellen Odometrie des NAOs berücksichtigt wird.

4.1.2 *Vorexperimente*

Bevor die Arbeit an der Odometriekorrektur der NAOs beginnen kann, müssen einige Vorexperimente und Beobachtungen durchgeführt werden, um das fehlerhafte Verhalten der Roboter zu dokumentieren. Um Rückschlüsse auf das Fehlverhalten ziehen zu können, sollen zu erst einige Daten der Läufe gesammelt werden. Für diese Rückschlüsse soll ein Verhalten geschrieben werden, welches den NAO zum Mittelpunkt des Feldes ausrichtet und in einer zuvor definierten Geschwindigkeit eine festgelegte Länge laufen lässt. Anschließend wird das Verhalten wiederholt, um Testdaten von mehreren Läufen aufnehmen zu können.
Aus den Beobachtungen dieses einfachen Verhaltens wurde früh ersichtlich, dass der NAO dazu tendierte, nach links abzudriften. Zur Aufnahme der Daten, der Position des Roboters und seiner Ausrichtung wird die Selbstlokalisation des NAO verwendet, sowie eine Lokalisierung über eine Deckenkamera.
Früh konnte mit diesem Aufbau beobachtet werden, dass die Lokalisierung des Roboters über die Deckenkamera besser funktioniert, als über die Selbstlokalisierung des Roboters. Der Roboter selber hatte oft starke Abweichung bezüglich seiner Position auf dem Feld (in x- und y-Richtung), die Position der Deckenkamera stimmte in fast allen Experimenten mit der tatsächlichen Position des Roboters überein. Allerdings erwies sich die Bestimmung der Ausrichtung des Roboter, also seiner Rotation auf dem Feld bei der Deckenkamera als fehleranfällig. Das Versuchsverhalten war so geschrieben, dass der NAO seinen Kopf abwechselnd nach den beiden weit-entferntesten Feldeckpunkten ausrichtete, um so möglichst viele Feldinformationen in Blick zu haben. Da die Deckenkamera den NAO über einen Marker, der auf seinem Kopf befestigt ist, lokalisiert, war es schwer, die Bewegung des Kopfes und damit auch des Markers mit seiner tatsächlichen Ausrichtung zu synchronisieren. Aus weiteren Beobachtungen ging hervor, dass die Bestimmung der Ausrichtung des Roboters auf dem Feld aus der Selbstlokalisierung des NAO oft richtig war.
Aufgrund dieser Beobachtungen wurde die Entscheidung getroffen, für die späteren Versuche die Position des NAO aus den Daten der Deckenkamera zu entnehmen, sowie die Daten über seine Ausrichtung aus der Selbstlokalisierung des Roboters.

4.1.3 *Aufgabe*

Aufgabe der Odometriekorrektur sollte sein, dass fehlerhafte Verhalten der Roboter auszugleichen. Dazu sollte die Systematik des fehlerhaften Geradeauslaufens über verschiedene Testdaten aufgedeckt werden, um aus diesen Testdaten eine Look-Up-Tabelle zu erzeugen,

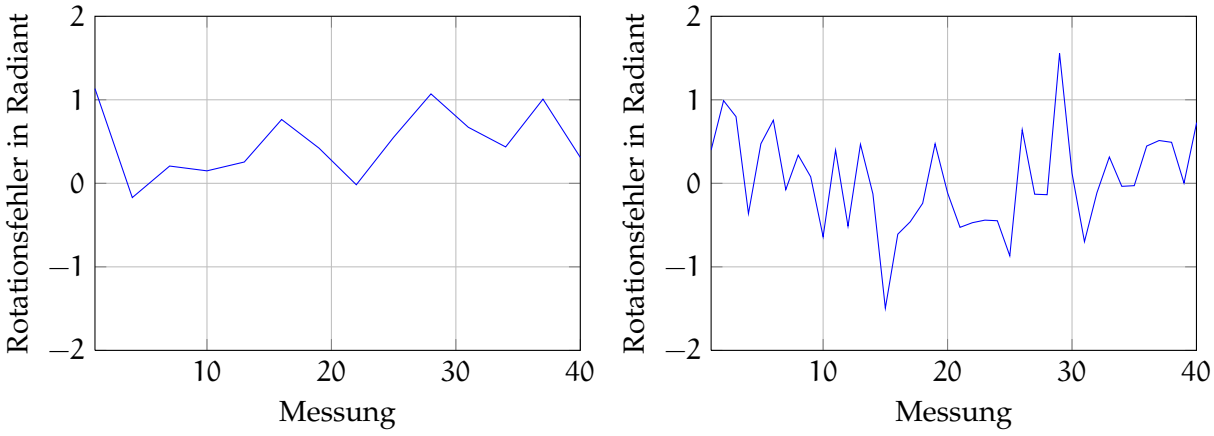
welche zur Korrektur der Odometrie hätte verwendet werden können.

4.1.4 *Versuchsaufbau*

Wie bereits zuvor angedeutet, sollten Messdaten über die Position und die Ausrichtung des Roboters erfasst werden. Um der Aufgabenstellung zu genügen, reichte es logischerweise nicht aus, den Roboter schlicht einen Lauf durchführen zu lassen, sondern musste Daten aus vielen Läufen sammeln, um eine gute Wahl für die Korrektur der Odometrie treffen zu können. Hierzu wurde das Verhalten geschrieben, welches zuvor erwähnt wurde.
Der NAO, ausgerüstet mit einem Marker auf seinem Kopf, wechselte die Blickrichtung stets zwischen den beiden weit-entferntesten Feldeckpunkten, um möglichst viele signifikante Informationen erblicken zu können. Vor jeder Messung steuerte der Roboter den Penalty-Punkt einer Feldseite an, richtete sich zum Mittelpunkt des Feldes aus und begann einen Testlauf für eine zuvor gewählte Geschwindigkeit und eine zuvor definierte Lauflänge. In unseren Tests wurden stets Läufe für einen Meter Länge durchgeführt mit vier unterschiedlichen Geschwindigkeiten (anhand der Informationen aus der Selbstlokalisierung des Roboters, welche für die Rotation des NAOs verwendet wurden, und den Daten aus der Deckenkamera für die Position des Roboters konnte nach jedem Lauf die Abweichung von der erwarteten Position und Ausrichtung errechnet werden.
Nachdem der NAO einen Lauf ausgeführt hatte, versuchte er sich durch das gegebene Verhalten in zehn Sekunden zu lokalisieren. Diese abschließende Lokalisierung sollte dazu dienen, die angenommene Position der Selbstlokalisierung des Roboters der tatsächlichen Position anzunähern, damit somit weniger Fehler in die Datensätze durch Fehllokalisierungen aufgenommen werden.
Hat der Roboter seine Lokalisierung nach zehn Sekunden verbessert, begann er wieder damit, sich zu einem Penalty-Punkt zu bewegen und zum Mittelpunkt des Feldes auszurichten, um den nächsten Datensatz aufzunehmen. Mit diesem Verhalten wurden für die vier erwähnten Geschwindigkeiten jeweils 40 Datensätze erzeugt. Das Aufnehmen einer solchen Testaufnahme dauerte circa eine Stunde.

4.1.5 *Resultate*

Die Abbildung 4.1 zeigt die jeweiligen Rotationsfehler in den unterschiedlichen Messungen. Tabelle 4.1 zeigt den kleinsten und größten Rotationsfehler, sowie den Durchschnitt und den Median der Rotationsfehler für die einzelnen Geschwindigkeiten.



(a) Rotationsfehler bei 50 mm/s

(b) Rotationsfehler bei 100 mm/s

(c) Rotationsfehler bei 150 mm/s

(d) Rotationsfehler bei 200 mm/s

Abbildung 4.1: Die Abbildungen zeigen grafisch den Rotationsfehler der einzelnen Läufe abhängig von den angesteuerten Geschwindigkeiten.

Geschwindigkeit	min	max	mit	med	std
50 mm/s	−0.171	1.134	0.485	0.429	0.403
100 mm/s	−1.494	1.554	0.045	−0.012	0.572
150 mm/s	−0.867	1.162	0.523	0.602	0.395
200 mm/s	0.426	1.239	0.702	0.631	0.213

Tabelle 4.1: Die Tabelle zeigt jeweils den kleinsten (min), größten (max), durchschnittlichen (mit) sowie den Median (med) der Rotationsfehler der Messungen bei den unterschiedlichen Geschwindigkeiten. Zusätzlich wird die Standardabweichung (std) der einzelnen Messungen mit aufgeführt. Die Einheit ist jeweils rad.

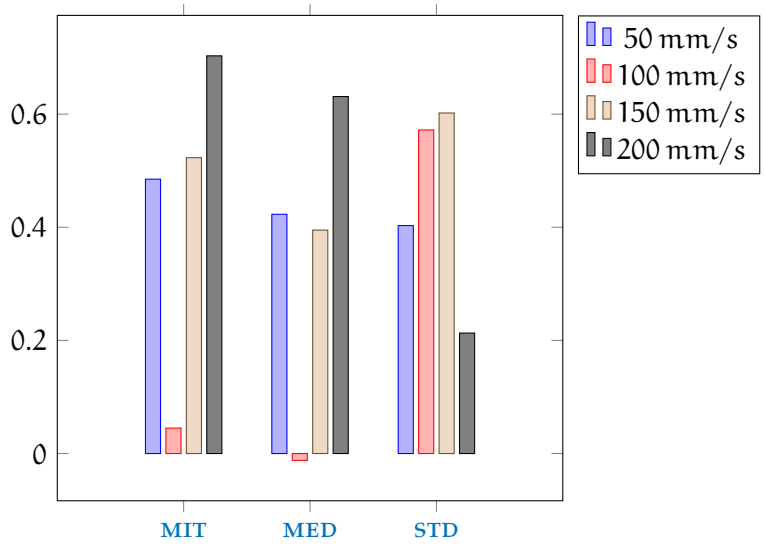


Abbildung 4.2: Die Abbildung visualisiert Durchschnitt (mit), Median (med) und Standardabweichung (std) der Messwerte aus Tabelle 4.1.

4.1.6 *Beobachtungen*

Bei 50 mm/s konnte beobachtet werden, dass der Roboter sich fast immer nach links drehte, wenn ein gerader Lauf angesteuert wurde. Im Schnitt wich die Endposition des Naos ca. 27.79° von der erwarteten Position ab, im Median 24.58°.
Bei 100 mm/s streute der Rotationsfehler des Roboters stark. Dadurch dass in beide Richtungen abgewichen wurde, zeigte sich im Durchschnitt und im Median keine große Abweichung, 2.58° bzw. −0.69°.
Bei 150 mm/s konnte wieder eine Tendenz zur Abweichung nach links beobachtet werden. Im Schnitt wich der Roboter bei dieser Geschwindigkeit 29.97° und im Median 34.42° nach links ab. Die größten Abweichungen konnten bei einer Geschwindigkeit von 200 mm/s gemessen werden. Sie lagen bei hier bei durchschnittlich 40.22° und

36.15° im Median.
Aus der Abbildung 4.1 und der Tabelle 4.1 geht hervor, dass, mit Ausnahme von 100 mm/s, der Rotationsfehler zunimmt, je schneller der NAO sich bewegt. Bemerkenswert ist jedoch, dass die Standardabweichung abnimmt, je schneller der Roboter sich bewegt. Das bedeutet, je schneller gelaufen wird, desto genauer kann für die untersuchten Geschwindigkeiten der Rotationsfehler bestimmt werden.

4.1.7 *Diskussion*

Die anfängliche Annahme, dass ein schnelleres Laufen des NAOs zu einem größeren Rotationsfehler führt, konnte durch die gewonnen Stichproben bestätigt werden. Bemerkenswert war außerdem, dass die Standardabweichung für die Stichproben bei einer Geschwindigkeit von 200 mm/s am geringsten war, was bedeutet, dass der Rotationsfehler bei dieser Geschwindigkeit am besten bestimmt werden kann. Eine spätere Korrektur der Odometrie wäre für diese Geschwindigkeit daher einfach im Gegensatz zu 100 mm/s. Bei diesem Tempo konnten Abweichungen in beide Richtungen festgestellt werden, eine Korrektur wäre daher durch dieses Verfahren nicht sicher zu bestimmen.

Das vorgestellte Verfahren zur Odometriekorrektur wurde bei der Optimierung des Roboterlaufs auf Grund mehrerer Nachteile nicht eingesetzt. Zum einen ist das Verfahren sehr zeitaufwändig, da für jeden Roboter eine individuelle Look-Up-Tabelle erstellt werden müsste und daher jeder Roboter zunächst eine große Anzahl von Stichproben erzeugen müsste. Diese Prozedur müsste wiederholt werden, sobald sich der Untergrund ändert oder die Hardware des Roboters sich verändert, beispielsweise durch Reparatur. Im Kontext des ROBOCUP 2012s wäre dieser Ablauf nicht möglich, da der benötigte Platz auf dem Feld zur Aufnahme der Stichproben fast niemals gegeben ist.
Des weiteren hängt die komplette Aufnahme der Daten von der eingesetzten Deckenkamera ab, die auf dem ROBOCUP 2012 ebenfalls nicht zur Verfügung steht. Es wäre denkbar hier die Selbstlokalisierung des NAOs zu benutzen. In den durchgeführten Vortests stellte sich jedoch heraus, dass dieses Vorgehen sehr große Ungenauigkeiten birgt, sodass es zur Bestimmung der Stichproben ungeeignet ist.
Zusätzlich zu den bereits beschriebenen Problemen stellt sich bei dieser Art der Optimierung des Laufs auch eine konzeptionelle Frage: Mit dem Erzeugen einer Look-Up-Tabelle und der nachträglichen Korrektur des Laufs wird nicht die Ursache des Fehlers bekämpft, sondern nur die Auswirkung. Es wurde sich dazu entschieden anstatt dieser Symptombekämpfung die Ursachen des fehlerhaften Laufs zu beseitigen, weswegen dieses Verfahren im Vergleich zur Optimierung

der Walking-Engine-Parameter und der Gelenkwinkelkalibrierung nicht weiter berücksichtigt wurde.

4.2 OPTIMIERUNG DER WALKING-ENGINE-PARAMETER

Dieses Kapitel beschäftigt sich mit der Optimierung der Parameter der Walking-Engine des NAO. Wie in dem vorherigen Kapitel bereits erwähnt, wird diese Idee aufgegriffen, da ein Verfahren für die Odometrie-Korrektur nur die Symptome und nicht die Ursache des Problems behandelt. Damit das Problem erst gar nicht entstehen kann, soll ein Optimierungsverfahren für die Walking-Engine Parameter eingesetzt werden, aus welchem ein gerader Lauf resultieren soll. Dieses Kapitel stellt dazu nur eine kurze Erläuterung dar, da das Vorhaben zu Gunsten einer besseren Idee nach dem ROBOCUP 2012 verworfen wurde.

4.2.1 *Vorgehen*

Die eingesetzte Walking-Engine beinhaltet mehr als 70 einstellbare Parameter. Führt man sich die Komplexität der vorgestellten Optimierungsverfahren aus Kapitel 2 vor Augen, wird schnell ersichtlich, dass ein Optimierungsverfahren für über 70 Parameter zu komplex ist. Es musste also zu erst eine geeignete Auswahl von zu optimierenden Parametern getroffen werden. Diese waren die *Arrays legJointHardness*, *heightPolygon*, *polygonLeft*, *polygonRight*, sowie die Variablen *footYDistance*, *doubleSupportRatio*, *xOffset*, sowie *stepDuration*.

LEGJOINTHARDNESS ist ein Array, dass aus sechs Werten besteht. Die sechs Werte bestimmen die Härte der einzelnen Gelenke im Bein. Angefangen mit den drei Hüftgelenken über das Gelenk im Knie, hin zu den beiden Gelenken im Knöchel. Es werden zur Optimierung allerdings nur die ersten fünf Werte gewählt, weil bereits bekannt ist, dass das letzte Gelenk sehr weich eingestellt sein muss.

HEIGHTPOLYGON ist ein Array der Größe fünf. Es beschreibt, welche Höhe der Fuß während eines Schritts erreicht. Die fünf Werte unterteilen den Schritt in einzelne Phasen, an dessen Ende der Fuß die angegebene Höhe erreicht. Jeder Wert in diesem Array ist ein Multiplikator, der an eine Maximalhöhe zur Ermittlung der resultierenden Höhe multipliziert wird. Die Werte sind daher begrenzt auf das Intervall $(0, 1)$.

POLYGONLEFT bestimmt den Verlauf des Zero Moment Point (ZMP) im linken Fuß während eines Schritts mit dem rechten Fuß. Dieses Array besteht aus vier Elementen, die den Verlauf des ZMP

während eines Schritts nach links und rechts verschieben. Sie unterteilen den Schritt ähnlich wie das *heightPolygon* in einzelne Phasen.

POLYGONRIGHT stellt die Funktion von *polygonLeft* für den rechten Fuß bereit und wird daher nicht eigenständig optimiert. Die Werte werden von *polygonLeft* spiegelverkehrt (mit -1 multipliziert) übernommen.

FOOTYDISTANCE bestimmt wie weit die Füße beim Laufen voneinander entfernt sind (breitbeinig).

DOUBLESUPPORTRATIO gibt an wie lange der NAO während eines Doppelschritts (linker und rechter Fuß) auf beiden Füßen steht.

XOFFSET ist ein Wert der beschreibt, wie weit die Hüfte vorgeschoben wird. Bei einer schnelleren Beschleunigung müssen Hüfte und Oberkörper weiter vorgeschoben werden, damit der Roboter nicht nach hinten umfällt. Auf der anderen Seite dürfen Hüfte und Oberkörper nicht zu weit nach vorne geneigt werden, damit der Roboter nicht vornüber kippt.

STEPDURATION bestimmt die Dauer eines Doppelschritts in Sekunden. Die Dauer wurde auf vier Werte (0.4, 0.6, 0.8, 1.0) begrenzt.

Damit ein Lauf optimiert wird, der in sich stabil ist, sollte ein Verfahren entwickelt werden, welches auf unterstützende Sensorkontrollen des Laufes verzichten. Zur Optimierung wurden zunächst zwei verschiedene Fitnessfunktionen betrachtet. Unabhängig von der Fitnessfunktion wurden Werte bei jeweils zehn sekündlichem geradeaus Laufen aufgenommen. Da die erste Fitnessfunktion, bei der die Werte der Gyroskopsensoren summiert werden, schwer zu interpretieren ist, wurde eine Funktion gewählt, welche über die Fußdrucksensoren berechnet, wann die Füße vom Boden abheben und wieder aufkommen. Aus den Werten der Fußdrucksensoren, die den IST-Zustand aufzeigen und dem berechneten SOLL-Zustand lässt sich somit eine einfachere Fitnessfunktion bestimmen.

4.2.2 *Abschließende Worte*

Bei diesem Verfahren traten wie bei der Odometrie-Korrektur einige Probleme auf. Die Datenaufnahme muss länger dauern als der Lauf, um signifikante Stellen für den Start und das Ende des Laufs entdecken zu können. Dadurch ergibt sich eine Zeitverschiebung in der Aufnahme, welche zuvor ausgeglichen werden muss. Dabei stellte sich heraus, dass es nicht trivial war, aus den aufgenommenen Daten automatisch zu erkennen, wann der Lauf startete und endete.

Ein weiteres Problem war, dass zu der Zeit, in der dieser Idee nachgegangen wurde, keine Roboter zur Verfügung standen, da diese gegen aktuellere Modelle ausgetauscht wurden. Der Simulator bot nur unzureichende Funktionen, um die Datenaufnahme zu simulieren, deshalb wurde vor dem RoboCup 2012 an den Fitnessfunktionen und einem Laufgestell gearbeitet, welche für die Optimierung genutzt werden sollte. Mit diesem Laufgestell hätte die Optimierung automatisiert stattfinden können. Eine Alternative dazu wäre, dass zwei Personen an der Optimierung arbeiten, einer arbeitet vor dem Computer und ein Zweiter verhindert, dass der Roboter umfällt.
Ein Laufgestell hätte wiederum auch einige Probleme hervorgerufen. Die Idee, dass der Roboter über ein Band unter den Armen gesichert ist, welches durch eine Schiene über ihm verläuft hätte die Optimierung verändern können, wenn dadurch Beeinträchtigungen für den Lauf entstehen würden. Wie aus der Odometrie-Korrektur schon zu sehen, läuft der Roboter allerdings gerade anfangs nichts geradeaus, was eine sinnvolle Konstruktion eines Laufgestells erschwert hätte. Aus diesem Grund wurde sich gegen das Laufgestell entschieden.
Wie bereits eingangs erwähnt, wurde die Idee der Optimierung der Walking-Engine Parameter allerdings generell für eine günstigere Lösung verworfen. Genauso wie bei der Odometrie-Korrektur, hätte jeder Roboter viele Optimierungsdurchläufe benötigt, um geeignete Parameterwerte zu finden. Dadurch wäre die Optimierung rein zeitlich äußerst komplex geworden. Wie bei der Odometrie-Korrektur, hätte man die Optimierung bei jedem neuen Untergrund oder jeder Reparatur wieder von neuem Ausführen müssen. Nachfolgend wird die Idee beschrieben, welche als bessere Lösung erschien, da sie automatisierbar einsetzbar ist.

4.3 KALIBRIERUNG DER GELENKWINKEL-OFFSETS

Der Nao hat in der Regel im Auslieferungszustand, wie auch im weiteren Gebrauch, eine Abweichung zwischen der über die Sensoren ermittelten und der tatsächlichen Winkelstellung. Durch Gelenkverschleiß und andere Beschädigungen am Nao sowie den Austausch von Teilen bei einer Reparatur ist es notwendig, die Gelenke regelmäßig wieder neu zu kalibrieren. Ohne diese Kalibrierung kann es zu einem unruhigen Lauf kommen, was diesen instabil macht und zu Abweichungen der Laufrichtung führt.
Von Hand ausgeführt ist die Kalibrierung recht aufwendig, da man durch viele Testläufe die optimale Offset-Einstellung finden muss. Wird ein einzelner Nao von Hand kalibriert, so beschäftigt dies drei Personen für gut eine Stunde, wobei ein gewisses Maß an Erfahrung gefordert ist, um durch scharfes Beobachten zu erkennen, welche Gelenke eine Nachjustierung erfordern.
Aufgabe des Teams war es, ein halbautomatisches Verfahren zu ent-

wickeln, das möglichst einfach und schnell gute Offset-Einstellungen findet.

4.3.1 *Grundidee*

Der NAO wird mit den Füßen in eine eigens dafür entwickelte Schablone fixiert. Diese gibt die Positionen der Beine und somit die Werte der Gelenke vor, die der NAO im aufrechten Stand haben soll. Für den Stand wurde die SpecialAction *standStraight* verwendet, da sie einem aufrechten Stand mit durchgestreckten Knien entspricht. Liegen Abweichungen bei den Gelenkwinkeln vor, lassen sich diese anhand der Sensorenwerte und dem Stromverbrauch der jeweiligen Gelenke erkennen.
Es wurden zwei Verfahren umgesetzt, die jeweils ausgehend von den Sensoren oder dem Stromverbrauch versuchen diese Abweichungen zu minimieren. Die beiden ausgewählten Verfahren werden in Abschnitt 4.3.3 genauer beschrieben.

4.3.2 *Schablone*

Es bieten sich verschiedene Möglichkeiten an, eine Schablone zu entwerfen, die die gewünschten Anforderungen erfüllt. Zum einem können die Fußpositionen eines zuvor per Hand kalibrierten NAOs verwendet werden. Zum anderen kann anhand seiner Geometrie eine Idealstellung umgesetzt werden.
Da die sechs NAOs aber nicht vollkommen identisch sind und die manuelle Kalibrierung des ursprünglichen nicht zwangsläufig bei allen zu guten Ergebnissen führen muss, wird die erste Variante nicht verwendet. So ist die Schablone anhand der Geometrie des NAOs angefertigt. Abbildung 4.3 zeigt in Bild 4.3a die Schablone von oben und in Bild 4.3b wie die Schablone zur Kalibrierung eingesetzt wird.

4.3.3 *Verfahren*

Hier wird auf die Verfahren eingegangen, die umgesetzt sind, um die Abweichungen der Gelenke auszugleichen. Es handelt sich um zwei grundverschiedene Ansätze. In Abschnitt 4.3.3.1 betrachten wir zunächst ein recht intuitives Verfahren, dass über die Sensoren der Gelenke die Differenzen zu ermitteln und auszugleichen versucht. In Abschitt 4.3.3.2 wird ein Verfahren beschrieben, dass die Ströme der Servomotoren der Gelenke misst und anhand dessen die Differenzen zu minimieren versucht.

(a) Die Schablone aus der Vogelperspektive

(b) Nao während einer Kalibrierung eingeschnallt in die Schablone

Abbildung 4.3: Die Schablone im Überblick und im Einsatz

4.3.3.1 *Ist-Soll-Vergleich*

Die Schablone wurde so angefertigt, dass die Beines des NAOs, der mit seinen Füßen darin fixiert wird, symmetrisch ausgerichtet sind und die Winkel der Gelenke denen eines kalibrierten NAOs entsprechen, der sich in der *standStraight*-Haltung befindet. Demzufolge sollten bei einem kalibrierten NAO alle von den Sensoren ermittelten den angesteuerten Gelenkwinkeln entsprechen. Eine Differenz wird als Fehler der Gelenkkalibrierung interpretiert und muss ausgeglichen werden. Dafür wird der Wert der Abweichung mit umgedrehten Vorzeichen als Offset des betreffenden Gelenks gesetzt.
Der Vorteil dieses Verfahrens ist die schnelle Ausführung innerhalb weniger Sekunden, jedoch erwiesen sich die Sensoren als zu ungenau, um gute Ergebnisse erzielen zu können.

4.3.3.2 *Messen der Ströme*

Dieses Verfahren betrachtet nicht die Winkel der Gelenke, sondern den Stromverbrauch der jeweiligen Servomotoren. Je weiter der aktuelle Winkel des betrachteten Gelenks vom angestrebten Wert abweicht, desto höher der Strom, mit dem der Motor versucht diese Position anzufahren. Diesen Stromverbrauch gilt es zu minimieren, um einen möglichst stabilen Lauf des NAO zu erreichen.
Bei der eigentlichen Kalibrierung wird der Strom in den Gelenken der Beine minimiert. Die Gelenke werden einzeln nacheinander kalibriert. Die Standardreihenfolge startet mit den oberen Rumpfgelenken und geht dann weiter nach unten.

Die Kalibrierung eines Gelenks erfolgt dabei nach folgendem Schema. Das Gelenk fährt äquidistante Stellungen innerhalb eines Bereiches um den aktuellen Winkel schrittweise an und verharrt kurze Zeit, um den gemittelten Stromverbrauch zu ermitteln. Die Gelenke haben etwas Spiel durch z.B. nicht perfekt ineinander verkeilende Zahnräder, der durch das Abfahren der Stellungen die Messung nicht so stark beeinflusst, wie es während einer Messung in ruhender Stellung der Fall wäre. Nach einer Sequenz wird diese in umgekehrter Reihenfolge ausgeführt, um produktionsbedingte lokale Optima für nur eine Richtung, die globale Optima überdecken könnten, nicht zu bevorzugen. Dieser Vorgang wird wiederholt bis eine festgelegte Anzahl von Iterationen erreicht wurde. Nach Vollendung wird die Gelenkwinkelstellung übernommen, die den geringsten Stromverbrauch hat.
Dies wird für alle Gelenke, die kalibriert werden sollen, durchgeführt. Ist ein Durchlauf aller Gelenke absolviert, wird für ein weiteres Feintuning der Umgebungsbereich, der um den aktuellen Winkel gelegt wird um die Winkel für die Kalibrierung zu erheben, verkleinert. Dies wird gemacht um das Minimum, welches zwischen zwei während des ersten Durchlaufs gemessenen Gelenkwinkeln liegen könnte, genauer zu approximieren. Wird der Umgebungsbereich nach einem Durchlauf kleiner als ein vorgegebener Schwellwert, wird die Kalibrierung beendet.
Bei dieser Form der Kalibrierung kann es vorkommen, dass der Oberkörper nach einer vollständigen Kalibrierung eine starke Neigung nach vorne aufweist. Diese Neigung wird mit Hilfe des Gyroskops behoben. Eine auf dem RoboCup 2012 entwickelte Variante dieser Methode optimiert nicht alle einzelnen Ströme nacheinander, sondern die Summe aller Ströme zu jedem Zeitpunkt. Dies entspricht einer Skalarisierung des mehrkriteriellen Optimierungsverfahrens. Der Ablauf des Verfahrens ändert sich dabei nicht, es werden lediglich die Ströme aller Gelenke erhoben und gemittelt.

4.3.4 *Theorie - Mehrkriterielle Optimierung*

Um die Vor- und Nachteile beider Lösungsansätze zu erschließen, ist eine genauere Betrachtung der Theorie der mehrkriteriellen Optimierung notwendig.

Wie der Name, mehrkriterielle Optimierung, bereits vermuten lässt, werden bei dieser Form der Optimierung mehrere Zielsetzungen verfolgt. Diese stehen nicht selten in Konflikt miteinander. Vorstellbar wäre z.B. das Szenario eines Getränkeherstellers, der ein möglichst erfrischendes Getränk zu möglichst geringen Kosten produzieren möchte. Dabei stehen die hohen Preise für Zutaten hoher Qualität, die eine größtmögliche Erfrischung garantieren, im Konflikt zu der Kostenminimierung in der Produktion. Hinzu kommt, dass viele Ziele inkommensurabel zueinander sind. Niemand kann sagen wie viel Er-

frischung einen Preis von z.B. 1 Euro rechtfertigt. Aus diesem Grund ist es oftmals nicht möglich, ein einziges globales Optimum zu finden, das allen Zielen einer mehrkriteriellen Optimierung gerecht wird. Um trotzdem zumindest teiloptimale Lösungen zu generieren wird, wie im Folgenden beschrieben, verfahren. Zunächst einige Definitionen von verwendeten Begrifflichkeiten.
Definition nach [DAVV]: Ein **mehrkriterielles Optimierungsproblem** ist: $(f_1(x), f_2(x), ..., f_m(x))'$: min!, bzgl. $x \in U$, wobei U ein beliebiges Universum sei. Gegeben die Nebenbedingung $g_i(x) \geqslant 0, i = 1, ..., m$.
Definition nach [DAVV]: Ein Vektor $u = (u_1, ..., u_m)$ **dominiert** einen Vektor $v = v_1, ..., v_m$, wenn gilt $\forall i \in 1, ..., m$, $u_i \geqslant v_i$ und $\exists i \in 1, ..., m$: $u_i > v_i$.
Definition nach [DAVV] und [Han98]: Eine Lösung $x_u \in U$ heißt **Pareto-optimal**, falls $\nexists\, x_v \in U$, so dass $v = f(x_v)$ dominiert $u = f(x_u)$. Wenn x_u Pareto-optimal ist, dann heißt $f(x_u)$ **effizient**. Die Menge aller Pareto-optimalen Punkte S^* heißt **Paretomenge**. Die Menge aller effizienten Punkte F^* wird **Paretofront** genannt.

Bei dem in Abschnitt 4.3.3.2 vorgestellten ursprünglichen Ansatz, läuft die Optimierung sukzessiv ab. Der Strom wird nacheinander in den einzelnen Gelenken minimiert. Durch dieses vorgehen, wird eine Pareto-optimale Lösung errechnet. Die Bedingung für eine Pareto-optimale Lösung wird dabei trivialerweise durch das Verfahren erreicht. Die erweiterte Variante ist die Umsetzung eines klassischen Ansatzes zur mehrkriteriellen Optimierung. Dieser Ansatz wird „Gewichtete Summe“genannt und stellt eine Skalarisierung der mehrkriteriellen Optimierung dar. Es gilt

$$\min_{x \in X} \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_m) \end{pmatrix} \Rightarrow \min_{x \in X} \sum_{i=1}^{m} w_i f_i(x), \qquad w_i > 0. \tag{4.1}$$

Für diesen Ansatz ist bekannt, dass die Lösung eines mehrkriteriellen Optimierungsproblems Pareto-optimal ist, falls für alle Kriterien die Gewichte positiv sind. In beiden Versionen wird demzufolge eine Pareto-optimale Lösung erreicht, da für die zweite Version eine Gleichgewichtung mit Gewichten größer Null gewählt wird.
Vorteil dieser Methode ist, dass nicht nur die Auswirkungen der Gelenkwinkeländerung auf das eine Gelenk, sondern die Korrelation mit den Strömen der anderen Gelenkwinkel berücksichtigt wird. Dies ist von Vorteil, wenn das Gewicht des Roboters durch eine Änderung eines Winkels so verlagert wird, dass die unveränderten Gelenke mehr Arbeit verrichten müssen als vorher. Ein solches Verhalten könnte sogar dazu führen, dass das Gelenk, in welchem die Änderung stattfindet, entlastet wird und somit der erste Ansatz eine solche Situation als Verbesserung ansehen könnte, obwohl sich der Abstand zum Ziel der Optimierung vergrößert.

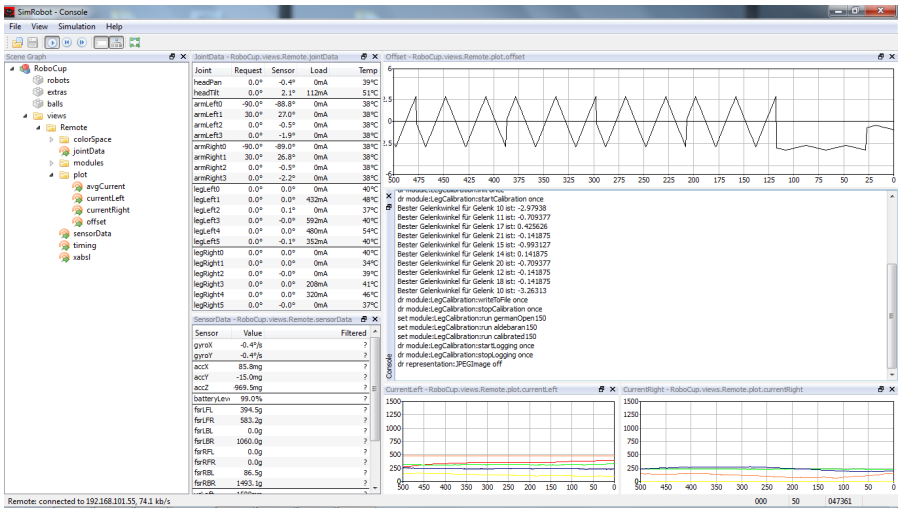


Abbildung 4.4: Die Abbildung zeigt die Szene im Simulator während einer Kalibrierung.

4.3.5 *Softwarebenutzung*

Um die Kalibrierung so schnell wie möglich zu gestalten, gibt es eine Leg Joint Calibration Szene, die nach dem Starten des Simulators geladen werden kann. In der Szene (Abbildung 4.4) sind die wichtigsten Befehle schon in der Reihenfolge aufgelistet, in der diese für eine erfolgreiche Kalibrierung aufgerufen werden müssen. Zusätzlich gibt es einige Fenster, die zur Überwachung und Auswertung der Kalibrierung dienen.

Es werden einerseits die „JointData“angezeigt, welche Informationen über die momentane Winkelpositionen und Offsets der Gelenkwinkel liefern. Hier wird auch die Temperatur der einzelnen Gelenke angezeigt, steigt diese über 80°C muss die Kalibrierung abgebrochen werden und der NAO einige Zeit abkühlen, da dieser sonst Hitzeschäden erleiden könnte. Im Fenster „SensorData“können die Werte der Fußdrucksensoren eingesehen werden. Dadurch ist eine Beurteilung der Gewichtsverlagerung des Roboters möglich. Während der Kalibrierung werden im Fenster „Offsets“die Offsets nacheinander eingetragen, die während der Kalibrierung an den Gelenken angelegt werden. Gleichzeitig lässt sich der Strom der Gelenke in den Fenstern „CurrentLeft“bzw. „CurrentRight“beobachten, dabei wurde zwischen linker - und rechter Hälfte des NAO aus Gründen der Übersichtlichkeit unterschieden.

Um eine Kalibrierung zu starten, bedarf es nichts weiter als die in der Szene vorhandenen Befehle der Reihe nach mit der „Enter“-Taste zu bestätigen. Die Funktionalität der einzelnen Befehle sei hier dennoch kurz erläutert.

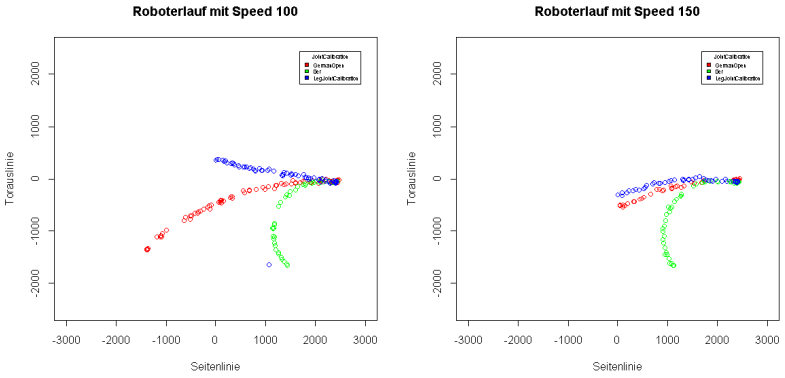
Die ersten drei Befehle „dr represantation:SensorData“, „dr represantation:JointRequest“und „dr represantation:JointData “ dienen der Anforderung der Daten, die in einigen der oben beschriebenen Fenstern angezeigt werden. Der folgende „set represantation:MotionRequest“ Befehl führt nach Bestätigung zum Aufrichten des Roboters, der sich dann in der specialAction „standLC“ befindet. Diese ist eine aus der specialAction „standStraight“ hervorgegangene specialAction, bei der der Roboter keine Härte in den Armen aufweist, um diese vor Überhitzung zu schützen. Der eigentlich Kalibrierungsteil beginnt jetzt, dazu sollte der NAO sich mittlerweile in der Schablone befinden. Mit Ausführung der Zeile „dr module:LegCalibration:init once“ werden alle Offsets der an der Kalibrierung beteiligten Gelenke auf Null gesetzt. Gestartet wird die Kalibrierung mit „dr module:LegCalibration:-startCalibration“. Um die erzeugten Offsets später auch komfortabel nutzen zu können, kann mit dem Befehl „dr module:LegCalibration :writeToFile“ eine .cfg Datei erzeugt werden, die dann nur noch an der richtigen Stelle auf den NAO aufgespielt werden muss. Vorher ist es noch möglich den Oberkörper des Roboters mit „dr module:LegCalibration:gyroCalibration“senkrecht aufzurichten, was bei einigen Kalibrierergebnissen zu einem deutlich stabileren Lauf führt.

Im Falle einer Überhitzung des Roboters lässt sich die Kalibrierung mit Hilfe der Zeile „dr module:LegCalibration:stopCalibration once anhalten“. Es wäre möglich die Kalibrierung wieder mit „dr module:LegCalibration:startCalibration“ fortzusetzen, doch sind die Ergebnisse danach nicht immer vorbehaltlos als optimal zu betrachten, da sich durch abkühlen der Gelenke die Voraussetzungen unter denen kalibriert wird ändern. Denkbar ist auch, dass der NAO z.B. um Abzukühlen in eine andere specialAction gebracht wird, was zu einer Veränderung der Position der Gelenke und der Ströme führen kann. Deswegen ist es ratsam, sollte eine Kalibrierung unterbrochen werden, diese neu zu starten.

4.3.6 *Evaluation*

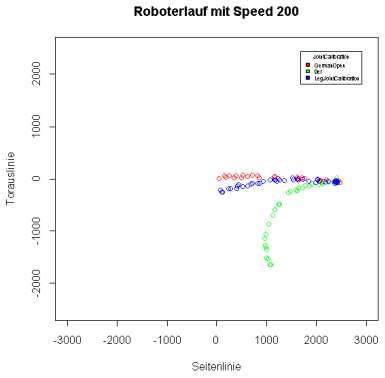
Um die Ergebnisse der Kalibrierung zu beurteilen, soll experimentell bestimmt werden, in wie weit der NAO mit den neuen Gelenkwinkeloffsets geradeaus läuft. Zu diesem Zweck kommt eine Deckenkamera zum Einsatz, mit der es möglich ist, die Position des Roboters auf dem Spielfeld zu bestimmen. Dem NAO wird dazu ein bestimmter Marker auf dem Kopf angebracht. Anschließend wird der NAO an eine markierte Position auf dem Spielfeld positioniert und der Lauf gestartet. Den zurückgelegten Pfad zeichnet die Deckenkamera anhand von Feldkoordinaten auf. So lässt dieser sich grafisch auswerten.

Abbildung 4.5 zeigt Pfade von je 3 verschieden Kalibrierungen bei je 3 verschiedenen Laufgeschwindigkeiten. In Grün dargestellt, sind



(a)

(b)



(c)

Abbildung 4.5: Aufzeichnung mehrerer Roboterläufe bei unterschiedlicher Kalibrierung und Geschwindigkeit. Das Koordinatensystem gibt die Position auf dem Spielfeld wieder. So ist der Punkt (0,0) der Spielfeldmittelpunkt und das Quartett (-2000,-3000),(2000,-3000),(-2000,3000) und (2000,3000) bildet die Spielfeldeckpunkte. Die Koordinaten sind in *mm* und die Geschwindigkeit in *mm/s* angegeben.

die Kalibrierungen vom Hersteller, in Rot die per Hand durchgeführten und in Blau die Resultate der semi-automatischen. Die Daten sind vor dem RoboCup 2012 erhoben worden und mit dem ursprünglichen Verfahren aus 4.3.3.2 kalibriert. Um eine einfache Übersicht zu ermöglichen, sind die x- und y-Achse so gewählt, dass diese im Verhältnis des Spielfeldes stehen. Der eingezeichnete Lauf spiegelt somit den wirklichen Lauf des Roboters wieder und ermöglicht den gleichen Eindruck des Laufs wie in der Realität.

Es wird deutlich, dass die semi-automatische Kalibrierung Winkeloffsets so liefert, dass ein Lauf ähnlich dem handkalibrierten erzeugt wird. Ebenso deutlich wird die starke Verbesserung im Vergleich zu der Kalibrierung des Herstellers.

Zu beachten ist, dass die Daten auf einem Spielfeld aufgenommen sind, welches nicht exakt eben ist. Kleine Unebenheiten können den Lauf des Roboters beeinflussen. Sie bewirken geringfügige Richtungsänderungen, so dass selbst für zwei Läufe mit der selben Kalibrierung nicht immer das gleiche Ergebnis zu erwarten ist. Die Pfade liegen allerdings so deutlich, dass sich die oben erwähnten Tendenzen auch unter Berücksichtigung der Unebenheiten feststellen lassen.

Um ein statistisch relevanteres Ergebnis zu erhalten, wird der oben beschriebene Versuch, in leicht abgewandelter Form - mit unterschiedlichem Startpunkt - , mehrmals hintereinander ausgeführt. Einen Eindruck des Versuchsaufbaus liefert die Abbildung 4.6. Diese zeigt in Bild 4.6a den Nao mit dem Marker für die Deckenkamera und im Hintergrund den Monitor mit dem das erkannte Bild der Deckenkamera überwacht werden kann. Als Startposition ist ein Punkt am Feldrand gewählt und der Roboter läuft einmal die Breitseite des Spielfeldes ab, illustriert in Bild 4.6b. Exemplarisch sind 10 Wiederholungen der Läufe mit einer jeden Kalibrierung in Abbildung 4.7 abgebildet. Hierbei ist zu erwähnen, dass die händische Kalibrierung schon etwas veraltet ist und durch Verschleiß und Transport der Roboter die zugehörigen Läufe nicht mehr die höchste Güte einer Handkalibrierung besitzen. Dennoch sind diese ausreichend um darzulegen, dass das Ziel, eine semi-automatische Kalibrierung zu verwirklichen, die eine ähnliche Güte wie die Handkalibrierung garantiert, erreicht ist.

Die Güte der Kalibrierung lässt sich schon bei Betrachtung der Abbildung 4.7 abschätzen. Klar ersichtlich ist, dass die Läufe mit der Kalibrierung des herstellers (Bild 4.7a) eine deutliche Linkskurve aus Sicht des Roboters aufweisen. Die händische Kalibrierung liefert schon eine sichtliche Verbesserung (Bild 4.7b). Die bzgl. eines geraden Lauf des Naos besten Ergebnisse kann die semi-automatische Kalibrierung (Bild 4.7c) vorweisen.

Dieser optische Eindruck lässt sich anhand von Zahlen verifizieren. Um die Ergebnisse zu vergleichen, wird die Kennzahl Abweichung [°/mm] erhoben. Dazu werden die Wegstrecken eines jeden

(a) Versuchsaufbau mit Kamerabild auf dem Monitor im Hintergrund

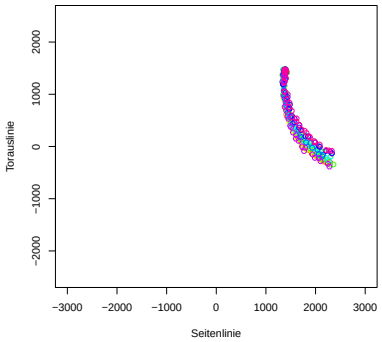
(b) Versuchsaufbau mit Blick aufs Spielfeld

Abbildung 4.6: Darstellung des Versuchsaufbaus zur Evaluierung des Roboterlaufs

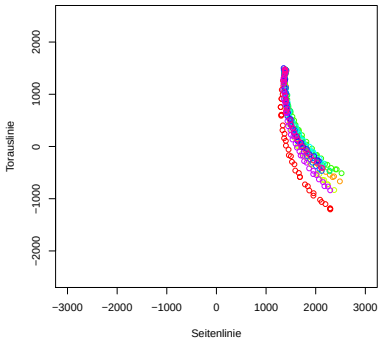
Laufes anhand der Summe der Abstände zwischen den Approximationspunkten des Laufes bestimmt. Die absolute Abweichung eines bestimmten Laufs in Bezug auf die gerade Laufstrecke wird in Relation zu dieser Länge des Laufs gesetzt. Das Ergebnis liefert die o.g. Kennzahl. Die Ergebnisse zu den Läufen aus Abbildung 4.7 sind in den Tabellen 4.2, 4.3 sowie 4.4 dargestellt. Die Auswertung der Mittelwerte der Abweichung [°/mm] bestätigt den oben gewonnen Eindruck. Im Mittel weichen die Läufe mit der Kalibrierung des Herstellers 0.0118 °/mm ab. Die handkalibrierten Läufe weisen eine mittlere Abweichung von 0.0091 °/mm auf. Die eindeutig geringste mittlere Abweichung besitzt die semi-automtische Kalibrierung mit einer Abweichung von nur 0.0019 °/mm.

Abschließend soll ein Vergleich zwischen den zwei entwickelten Kalbrierungsverfahren angestellt werden. Hierzu wurde obiges Experiment mit jeweils zwei Kalibrierungen jedes Verfahrens durchgeführt.

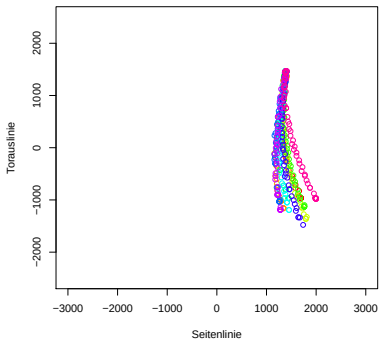
Abbildung 4.8 zeigt, dass es mit beiden Verfahren möglich ist eine Kalibrierung hoher Güte zu erzeugen. Ebenso wird deutlich, dass nicht jede Kalibrierung einen optimalen Lauf garantiert, siehe 4.8c. In dem durchgeführten Experiment liefert die sequentielle Kalibrierung bessere Ergebnisse. Die jeweiligen mittleren Abweichungen des ersten Ansatzes von 0,019 *[°/mm]* (4.8a)und 0,045 *[°/mm]* (4.8b) sind beide besser als die beste mittlere Abweichung des zweiten Ansatzes, die 0,0573 *[°/mm]*(4.8d) beträgt.



(a) Läufe mit Kalibrierung des Herstellers

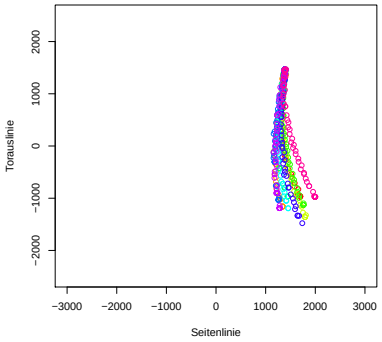


(b) Läufe mit händischer Kalibrierung

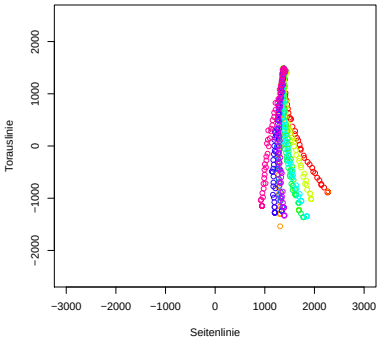


(c) Läufe mit semi-automatischer Kalibrierung

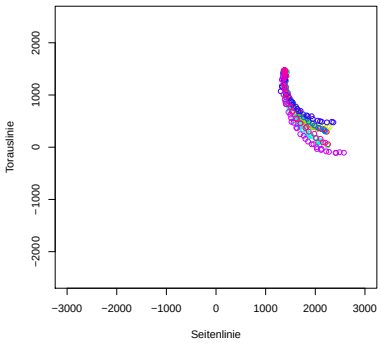
Abbildung 4.7: Abbildung verschiedener Evaluierungsläufe mit 3 unterschiedlichen Kalibrierungen.



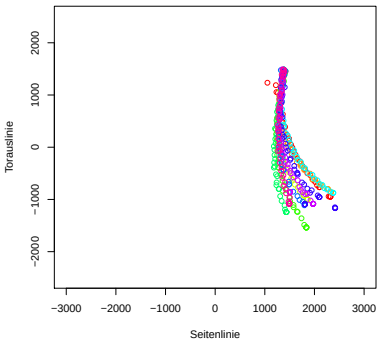
(a) Lauf mit 1. Kalibrierung des sequentiellen Verfahrens



(b) Lauf mit 2. Kalibrierung des sequentiellen Verfahrens



(c) Lauf mit 1. Kalibrierung des summierten Verfahrens



(d) Lauf mit 2. Kalibrierung des summierten Verfahrens

Abbildung 4.8: Vergleich der zwei entwickelten Verfahren anhand zweier Kalibrierungen. Bei einer Geschwindigkeit von 150 *mm/s*. Das Koordinatensystem entspricht den Abmessungen des Spielfeldes in *mm*.

Startpunkt x,y-Wert	Endpunkt x,y-Wert
1388, 1472	2288, -300
1392, 1464	2242, -321
1384, 1466	2119, -245
1385, 1470	2361, -344
1385, 1468	2302, -194
1386, 1465	2268, -261
1387, 1474	2135, -189
1385, 1465	2328, -131
1389, 1466	2280, -384
1388, 1471	2326, -83

Länge [mm]	Abweichung gesamt [°]	Abweichung [°/mm]
2351.28	-26.93	-0.0115
2294.04	-25.46	-0.0111
2191.93	-23.25	-0.0106
2488.62	-28.28	-0.0114
2425.90	-28.89	-0.0119
2234.90	-27.07	-0.0121
2231.54	-24.22	-0.0109
2190.01	-30.58	-0.0140
2463.53	-25.72	-0.0104
2210.01	-31.12	-0.0141

Tabelle 4.2: Lauf mit Kalibrierung des Herstellers

4.3.7 *Fazit und Ausblick*

Da das bisherige Verfahren keine eindeutigen, sondern nur Pareto-optimale Lösungen findet, sind die Ergebnisse der Kalibrierung variierend. Ein mögliches Verfahren, welches alle Punkte der Paretomenge generiert und eine Entscheidung darüber fällen kann welche Lösung in Bezug auf die Laufstabilität von Vorteil ist, würde zu besseren Ergebnissen führen. Allerdings stünde der exponentiell erhöhte Zeitaufwand einer solchen Berechnung nicht im Verhältnis zum Mehrnutzen. Vor allem wenn der Nao weiterhin aktiv in die Kalibrierung einbezogen werden muss, kann die erzeugte Hitze in den Gelenken ein schwer in den Griff zu bekommendes Problem darstellen.
Aus diesem Grund bildet die vorgestellte Lösung einen guten Kompromiss zwischen Güte und Zeitaufwand der Kalibrierung. Denn der

STARTPUNKT X,Y-WERT	ENDPUNKT X,Y-WERT
1383, 1470	2292, -1188
1391, 1456	2487, -668
1385, 1469	2373, -834
1388, 1469	2521, -512
1381, 1471	2280, -475
1388, 1469	2159, -326
1391, 1465	2121, -468
1383, 1464	2061, -277
1386, 1452	2292, -839
1381, 1472	2134, -410

LÄNGE [mm]	ABWEICHUNG GESAMT [°]	ABWEICHUNG [°/mm]
3249.11	-18.88	-0.0058
2745.28	-27.29	-0.0099
2898.25	-23.22	-0.0080
2660.88	-29.77	-0.0112
2333.81	-24.80	-0.0106
2344.08	-23.24	-0.0099
2340.83	-20.69	-0.0088
2152.26	-21.28	-0.0099
2771.75	-21.58	-0.0078
2188.98	-21.81	-0.0100

Tabelle 4.3: Lauf mit händischer Kalibrierung

Nutzer kann in einem überschaubaren Zeitrahmen mehrere Kalibrierungen durchführen, so dass es möglich ist aus unterschiedlichen Lösungen der Paretomenge zu wählen. Dabei ist die Validierung der Güte einer Lösung meist mit dem Auge erkennbar, indem der NAO eine Linie entlang laufen gelassen wird. Läuft der NAO bei mehrmaliger Ausführung vermehrt einen nicht tolerierbaren Bogen, der nicht durch Unebenheiten oder anderer Fehler des Untergrundes erklärt werden kann, sollte die Kalibrierung wiederholt werden. Eine zufriedenstellende Lösung sollte so in einer verträglichen Zeit gefunden werden.

Startpunkt x,y-Wert	Endpunkt x,y-Wert
1387, 1457	1696, -966
1386, 1459	1343, -1159
1383, 1462	1794, -1361
1384, 1462	1760, -1128
1387, 1456	1443, -967
1385, 1470	1454, -1192
1388, 1456	1260, -1032
1382, 1465	1738, -1478
1387, 1461	1280, -1189
1387, 1457	1997, -974

Länge [mm]	Abweichung gesamt [°]	Abweichung [°/mm]
2666.83	-7.27	-0.0027
2959.43	0.94	0.0003
3100.44	-8.28	-0.0027
2833.78	-8.26	-0.0029
2651.20	-1.32	-0.0005
2811.97	-1.48	-0.0005
2795.55	2.95	0.0011
3325.33	-6.90	-0.0021
3001.52	2.31	0.0008
2780.19	-14.09	-0.0051

Tabelle 4.4: Lauf mit semi-automatischer Kalibrierung

5 FEED-FORWARD-CONTROLLER

Um den Lauf eines Roboters weiter zu verbessern und Gelenkwinkel schneller anzufahren, können die gewohnten PID-Regler innerhalb der Gelenkservos zu Feed-Forward PID (FFPID) Reglern erweitert werden. Jedoch bietet der NAO keine Möglichkeit zur Modifikation der verwendeten Regelungsalgorithmen an, weswegen im Folgenden auf käufliche Standardservos zurückgegriffen wird. Da die Implementierung direkt in die Firmware der Servoplatine eingreift, die Servohersteller jedoch aus verständlichen Gründen weder Informationen zur Schaltung, noch zur darauf ausgeführten Software bereitstellen, musste eine alternative Elektronik zur Ansteuerung des Motors verwendet werden. Die Wahl fiel auf eine bereits vollständig bestückt käufliche Platine, die im Rahmen des OpenServo Projektes [Ope12] entwickelt wurde (Abbildung 5.1b). Auf die Entwicklung einer eigenen Platine wurde bewusst verzichtet, da die Entwicklung mühsam und langwierig ist und so den Rahmen dieser Projektgruppe sprengen würde. Für mehr Details zur Entwicklung einer eigenen Elektronik sei auf die Studienarbeit von Daniel Hauschildt verwiesen [Hau08].

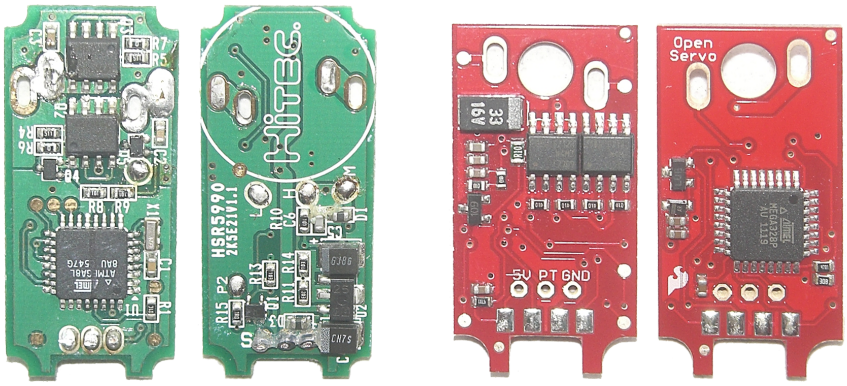
Umbau eines Servos

OpenServo Platine

Dieses Kapitel behandelt zunächst die Grundlagen des theoretischen Modells vom FFPID. Anschließend folgt die Dokumentation der Hard- und Softwarekomponenten, die für den Betrieb erforderlich sind, mit besonderer Berücksichtigung, dass die Versuche auch durch Dritte (Hausarbeiten, Projektgruppen) ohne großen Aufwand reproduzierbar sind. Schlussendlich werden die Ergebnisse, die im Rahmen dieser Projektgruppe erreicht wurden evaluiert.

5.1 FEED-FORWARD-PRINZIP

In diesem Abschnitt werden die zum Verständnis einer Feed-Forward-Control benötigten Grundlagen ausgeführt. Zunächst werden dynamische Systeme genauer erläutert und gezeigt, wie diese kontrolliert werden können. Anschließend wird das Konzept eines PID-Reglers vorgestellt, während im letzten Teil die Eigenschaften und Vorteile einer Feed-Forward-Control erklärt werden. Grundlage für dieses Unterkapitel sind die Arbeiten von Furlan [Fur08], Lunze [Lun08] und Unbehauen [Unb07].



(a) Original (b) OpenServo

Abbildung 5.1: Elektronik zur Regelung des Servomotors. Links das Original des Herstellers, rechts die OpenServo Platine

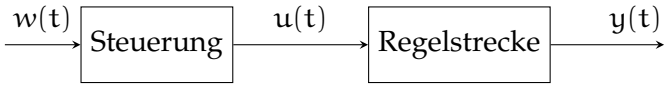


Abbildung 5.2: Die Abbildung zeigt die generelle Struktur einer Steuerung oder eines Open-Loop-Systems. Die Steuerung bekommt ein Eingangssignal $w(t)$, welches in ihr in eine Stellgröße $u(t)$ übersetzt wird. Die Stellgröße bedingt den Ausgang $y(t)$ der Regelstrecke.

5.1.1 *Dynamische Prozesse und Regler*

dynamisches System

Das Problem der Steuerung eines Elektromotors fällt in den Bereich der Regelungstechnik. Dieser Zweig der Ingenieurswissenschaften beschäftigt sich mit der Untersuchung realer Systeme in Hinblick auf ihre *Dynamik* und wie diese bestmöglich kontrollierbar ist. Dynamische Systeme sind generell Systeme, deren Kenngröße sich als zeitlich veränderbare Funktionen darstellen lassen. Sie stellen Funktionseinheiten dar, die Signale annehmen, verarbeiten und anschließend eine entsprechende Auswirkung wieder zurückgeben. Bei einem Elektromotor handelt es sich genau um ein solches dynamisches System. Als Eingangssignale werden an den Motor elektrische Ströme angelegt, die im Motor in Drehmomente übersetzt werden, wobei diese Drehmomente sich durch eine entsprechende Stellung des Motors zeigen. Das Eingangsignal *Strom* wird also durch das dynamische System in ein Ausgangssignal *Motorstellung* überführt.

Steuerung und Regelung

Um dynamische Systeme kontrollieren zu können, gibt es generell zwei Verfahren in der Regelungstechnik: Die *Steuerung* und die *Regelung*. Die Abbildung 5.2 veranschaulicht das Prinzip einer Steuerung, Abbildung 5.3 das einer Regelung. In beiden Systemen existiert ein Signal $w(t)$, welches die *Führungsgröße* oder *Sollwert* repräsentiert. Ge-

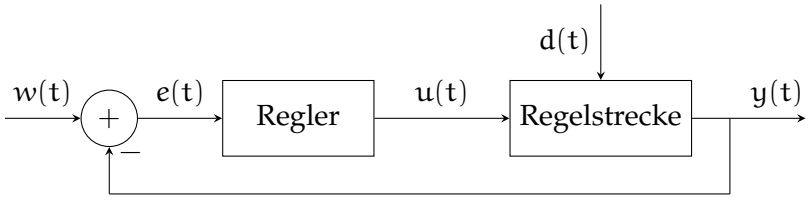


Abbildung 5.3: Die Abbildung zeigt die generelle Struktur einer Regelung oder eines Closed-Loop-Systems. Das Führungssignal $w(t)$ wird mit dem aktuellen Ist-Wert $y(t)$ der Regelstrecke verglichen. Der Fehler $e(t)$ zwischen Soll- und Ist-Wert wird anschließend an einen Regler weitergeleitet, der aus dem Fehler die Stellgröße $u(t)$ für die Regelstrecke erzeugt.

meint ist hiermit die gewünschte Ausgabe, die das System annehmen soll. Die Steuerung entscheidet unabhängig vom derzeitigen Systemzustand dann, welche *Stellgröße* $u(t)$ für das System geeignet ist, um den Sollwert zu erreichen.

Steuerung ignoriert den Systemzustand

Im Unterschied zur Steuerung, benutzt die Regelung zusätzlich zum Sollwert noch den derzeitigen *Ist-Wert* des Systems. Dieser wird am Ausgang des Systems abgegriffen und vom Sollwert abgezogen. Der so berechnete Fehler $e(t)$ wird dem Regler zugeführt, der ausschließlich an Hand dieses Fehlers die geeignete Stellgröße berechnet.

Regelung benutzt den Systemzustand

Durch die Eigenschaft der Steuerung, dass ausschließlich der Sollwert betrachtet wird, kann die Steuerung nicht auf Änderungen des dynamischen Systems einwirken. Die Steuerung würde beispielsweise eine gewünschte Stellung des Elektromotors in einen geeigneten Eingangsstrom als Stellgröße übersetzen. Hängt nun am Elektromotors aber ein Gewicht, würde der Strom nicht ausreichen, um die gewünschte Position anzufahren.

Im Gegensatz hierzu bezieht ein Regler den aktuellen Ist-Wert des dynamischen Systems mit ein. Beim obigen Beispiel würde der Regler merken, dass der berechnete Strom nicht ausreicht, um die gewünschte Position anzufahren, und würde daher die Stellgröße entsprechend anpassen.

Korrektur von Störgrößen

Der Vorteil einer Steuerung gegenüber einer Regelung besteht darin, dass eine Steuerung im allgemeinen schneller arbeitet. Der Vorteil einer Regelung liegt in ihrer Eigenschaft auf unvorhersehbare Störgrößen reagieren und diese ausgleichen zu können.

5.1.2 *PID-Regler*

Für den Einsatz bei einem Elektromotor ist ein Regler besser geeignet, als eine Steuerung. Es existieren eine Reihe von Reglertypen, die für unterschiedliche dynamische Systeme ausgelegt sind. Ein geeigneter Typ für den Elektromotor ist der *PID-Regler*.

PID-Regler

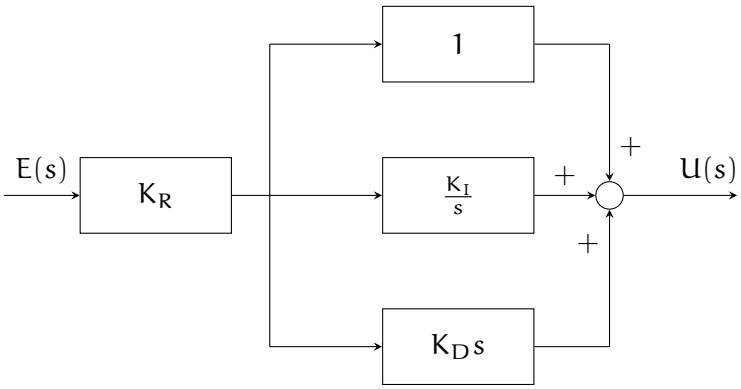


Abbildung 5.4: Die Abbildung zeigt den Aufbau eines PID-Reglers. Der Fehler $E(s)$ wird zunächst durch den Faktor K_R verstärkt. Anschließend werden zusätzliche Integral- und Differentialanteile des Signals hinzugefügt, um die Stellgröße $U(s)$ zu erzeugen.

Seinen Namen verdankt dieser Regler seinen drei Komponenten, mit denen aus dem Fehlersignal $e(t)$ die Stellgröße $u(t)$ berechnet wird. Die Abbildung 5.4 verdeutlicht das Prinzip eines PID-Reglers. Der Fehler $E(S)$ wird zunächst über mit einem Faktor proportional verstärkt. Danach wird das Signal aufgeteilt und in drei Blöcke weitergeleitet. Der oberste Block gibt das Signal einfach nur weiter, während im mittleren Block das Signal integriert und im unteren differenziert wird. Der Differential- und Integralteil werden danach zusätzlich noch mit den Faktoren K_I bzw. K_D verstärkt. Anschließend werden alle drei Signale wieder zusammengeführt. Die Summe entspricht der Stellgröße $U(s)$. Der vorgezogene Verstärkungsfaktor K_R gehört eigentlich zum oberen Zweig, dem Proportionalteil. Es ist jedoch üblich in vor die drei Zweige zu ziehen, wodurch er auch den Integral- und Differentialteil beeinflusst. Die drei Teile (Proportionalteil, Integralteil und Differentialteil) geben dem PID-Regler seinen Namen.

Differentiation, Integration

Zum besseren Verständnis sei erwähnt, dass in der Abbildung 5.4 die jeweiligen Signale mit Hilfe der *Laplace-Transformation* in den Bildbereich überführt wurden. Die Laplace-Transformation ist eine Integraltransformation, die einer Funktion $f(t)$ im Zeitbereich eine Funktion $F(s)$ im Spektral- oder Bildbereich zuordnet. $F(s)$ wird über die Formel

Laplace-Transformation

$$F(s) = \int_0^\infty f(t)e^{-st}dt$$

aus $f(t)$ erzeugt. Dieses Verfahren findet häufige Anwendung in der Regelungstechnik, da viele Systeme im Bildbereich einfacher zu lösen sind, als im Zeitbereich.

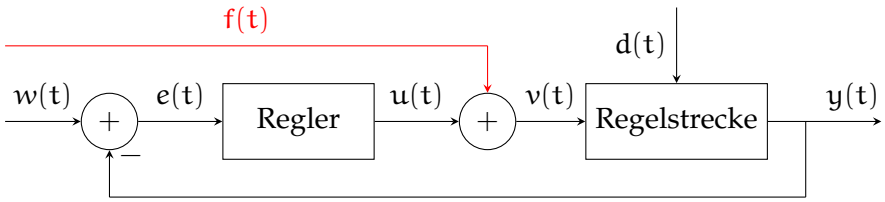


Abbildung 5.5: Struktur eines Feed-Forward-Controllers. Der Regler unterscheidet sich nur durch die zusätzliche Stellgröße $f(t)$ (rot gekennzeichnet) vom Standardregler.

5.1.3 *Feed-Forward-Control*

Über die drei Parameter K_R, K_I und K_D kann ein PID-Regler eingestellt werden. Die Wahl der Parameter bedingt das Verhalten des dynamischen Systems, welches gesteuert werden soll. So kann es sein, dass bei falscher Parameterwahl das Gesamtsystem aus Regler und dynamischem System instabil wird. Am Beispiel des Elektromotors würde das bedeuten, dass sich der Motor aufschwingt, wenn er versucht eine gewünschte Position anzufahren. Der Regler für einen Elektromotor sollte daher nicht zu viel Strom auf den Motor geben dürfen.

ungünstige K_R, K_I, K_D ⇒ Instabilität

Das Problem was sich nun ergibt ist folgendes: der Elektromotor soll eine Position möglichst schnell anfahren, während der Regler aber nicht zu viel Strom einstellen darf. Zur Lösung des Problems, müssen die Eigenschaften des PID-Reglers betrachtet werden. Er arbeitet allein auf dem Fehler zwischen Soll- und Ist-Wert, hat also kein eingebautes Vorwissen, über das System, was er Regeln soll.

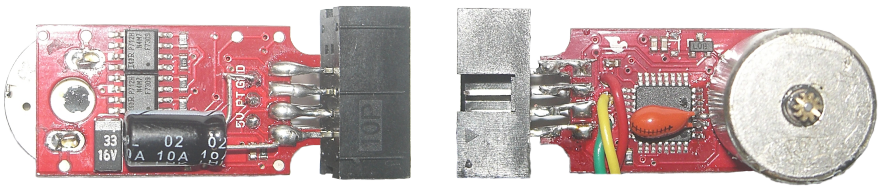
Hier eignet sich das Prinzip der *Feed-Forward-Control*. Bei dieser Erweiterung des klassischen PID-Reglers wird zusätzliches Systemwissen mit in den Regelungsprozess eingebaut. Abbildung 5.5 veranschaulicht dieses Konzept. Zusätzlich zu der aus Abbildung 5.3 bekannten Regelung, wird ein Signal $f(t)$ eingeführt. Dieses Signal steuert direkt das dynamische System an. Es handelt sich hierbei also um eine Mischform aus Steuerung und Regelung. Über den neuen Zweig kann bestehendes Systemwissen mit in die Regelung eingeführt werden.

Feed-Forward-Control

zusätzliches Systemwissen

Bei einem Elektromotor kann die Feed-Forward-Control Wissen über die Drehmomente am Motor mit einbeziehen. Die Position des Motors wird allein durch die Drehmomente bestimmt, die an ihm anliegen, also das Drehmoment, was der Motor selbst ausübt, und das Drehmoment, welches durch die Last am Motor ausgeübt wird. Es existiert Systemwissen darüber, welches Drehmoment angelegt werden muss, um eine bestimmte Position zu erreichen. Wenn also das Lastdrehmoment ungefähr abschätzbar ist, kann direkt ein Strom angelegt werden, der den Motor in die Nähe der gewünschten Position

Vorwissen über Drehmomente



(a) Oberseite (b) Unterseite

Abbildung 5.6: Erweiterte OpenServo Platine um einen Eletrolytkondensator direkt am Eingang der Versorgungsspannung (links), um einen Tantalkondensator direkt über den Versorgungsspannungsbeinchen des Mikrocontrollers (rechts), sowie um einen Steckverbinder mit Verpolschutz.

fährt. Der PID-Regler regelt danach den Unterschied zwischen Soll- und Ist-Wert genauer aus. Das Konzept vereint die Geschwindigkeit einer Steuerung mit der Eigenschaft einer Regelung auf Störgrößen Einfluss nehmen zu können und diese ausgleichen zu können.

5.2 HARDWAREKOMPONENTEN

Dieser Abschnitt widmet sich den verwendeten Hardwarekomponenten, die für den Versuchsaufbau erforderlich sind. Es handelt sich dabei um die OpenServo Platine, die die originale Platine des verwendeten Servos ersetzt und um das OpenServo Interface (OSIF), was als Bindeglied zwischen Computer und Servo dient. Da diese entweder modifiziert oder vollständig selbst zusammengebaut wurden, bedürfen sie einiger Erläuterung. Dabei wird zunächst auf die Eigenarten der OpenServo Platine eingegangen, die in ihrer Originalversion dazu neigt, das Servogetriebe zu zerstören. Anschließend werden die Schnittstellen und Modifikationen des OSIF erläutert, das neben der bereitgestellten I^2C Schnittstelle auch eine Möglichkeit zur initialen Programmierung von sich selbst, sowie eines angeschlossenen Servos bietet.

5.2.1 *OpenServo Platine*

Schubumkehr = Spannungseinbruch

brown out reset

Obwohl die käuflich erworbene OpenServo Platine (Abbildung 5.1b) dem originalen OpenServo Schaltplan entspricht, weist sie bei dem von uns verwendeten Motor große Probleme auf. Diese äußern sich bei abrupter Schubumkehr durch heftige Spannungseinbrüche auf der 12 V Versorgungsspannungsleitung des Motors, die auf die 5 V Leitung über-koppeln, welche der Versorgung des Mikrocontrollers dient. Daraufhin tritt häufig eine Unterschreitung der minimalen Versorgungsspannung (brown out) auf, die zu einem Reset des Mikrocontrollers führt und alle seine Ein- und Ausgänge auf *tristate* (d.h.

so, als wären sie nicht angeschlossen) schaltet. Auch die Steuerleitungen zur MOSFET-H-Brücke, die den Motor ansteuert, sind davon betroffen, sodass die geladenen Gatekapazitäten der MOSFETs nicht entladen werden und die Transistoren ihre aktuelle Leitfähigkeit beibehalten. Der Motor beschleunigt weiter, bis sich die Versorgungsspannung stabilisiert und der Mikrocontroller neu startet. Je nach Konfiguration der Firmware macht dieser zunächst nichts oder regelt der aktuellen Bewegung heftig entgegen, was zu einem erneuten Spannungseinbruch führt. Befindet sich an dem Motor das Getriebe des Servos, so wird dieses mit voller Wucht an seine Begrenzung gefahren, was nach einigen Einschlägen die empfindlichen Zahnräder beschädigt.

Motor bleibt eingeschaltet

Getriebe leidet

Da dieses Verhalten den Servo unbenutzbar macht, wurde ein Großteil der Zeit, in der die PG an OpenServo gearbeitet hat, für die Auffindung und Behebung dieses beschriebenen Fehlers aufgewendet. Die Lösung stellt ein 33 nF Tantalkondensator dar (Abbildung 5.6b), der direkt an die Versorgungsspannungspins 3 *(GND)* und 4 *(VCC)* am Mikrocontroller gelötet wird und so plötzliche Spannungseinbrüche abfängt und brown-out-Zustände verhindert.

Abblockkondensator

5.2.2 *OSIF*

Das OSIF (Abbildung 5.7) schlägt im Versuchsaufbau die Brücke zwischen dem steuernden Computer und dem angesteuerten Servo. Es wird über USB am Computer angeschlossen und versorgt auf diesem Wege sowohl sich selbst, als auch die 5 V Versorgungsspannungsleitung angehängter Servos. Zum Versorgen der 12 V Leitung, die von den Servos zum Treiben der Motoren benötigt wird, muss zusätzlich ein (starkes) externes Netzteil angeschlossen werden. Natürlich ist das OSIF nicht zwingend für den Betrieb der Servos erforderlich und kann bei Computern bzw. Robotern mit vorhandener I^2C Schnittstelle und hinreichender Spannungsversorgung auch entfallen.

12 V für Motor von extern

Im Rahmen der Projektgruppe wurde das OSIF aus diskreten Bauteilen aufgebaut und der darauf liegende Mikrocontroller mit einer Firmware programmiert, die Bestandteil des OpenServo Projektes ist. Der Aufbau wurde jedoch im Gegensatz zum OpenServo Original um folgende Aspekte abgeändert:

ZEHNPOLIGER SERVO STECKVERBINDER Servo und OSIF benutzen anstatt einer achtpoligen Stiftleiste einen zehnpoligen Wannenstecker mit Verpolungsschutz (In Abbildung 5.7 als *Servo* gekennzeichnet).

ENTFALLENDE GPIO PINS Die Anschlüsse *GPIO1* und *GPIO2* sind nur teilweise angeschlossen, da sie im Rahmen der Experimente nicht benötigt werden. Sie können jedoch bei Bedarf nachverdrahtet werden.

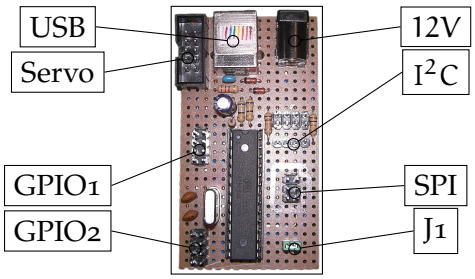


Abbildung 5.7: OpenServo Interface. Modul verbindet den PC über USB sowie den Servo via I^2C und bildet so eine Brücke zwischen PC und Servo.

JUMPER Ein Programmierjumper wurde nachträglich hinzugefügt, der sowohl die Programmierung eines angeschlossenen Servos (*J1* geöffnet), als auch die Programmierung des OSIF selbst (*J1* geschlossen) über den *SPI* Anschluss ermöglicht. Im normalen Betrieb ist *J1* geschlossen.

5.3 SOFTWARE

Im Folgenden wird auf Softwareprogramme eingegangen, die zur Benutzung und Erweiterung des Servos erforderlich sind. Sie befinden sich im Repository `servo.naodevils2011.git` oder sind frei im Internet verfügbar. Die Dokumentation bezieht sich auf Microsoft Windows 7 in der 64 Bit Version. Allerdings befinden sich im Repository jedoch auch Programme und Kernelmodule für einen Einsatz unter Linux, die um schrittweise Anleitungen in Readme-Dateien erweitert wurden.

Software für Windows 7 (64 Bit)

Es werden das initiale Kompilieren und Aufspielen von Firmware, die Installation benötigter Gerätetreiber, das Übersetzen von Bibliotheken, sowie die Struktur des Softwareframeworks erläutert.

5.3.1 *Firmware*

Sowohl die Servoplatine als auch das OSIF besitzen jeweils frei programmierbare Mikrocontroller von Atmel, die mit einer Firmware ausgestattet werden müssen. Dabei handelt es sich um C-Programme, die durch einen speziellen Compiler (AVR-GCC) übersetzt werden und mit einem Programmieradapter (Abbildung 5.8) auf den Chip übertragen werden. Die benötigte Entwicklungsumgebung *Atmel Studio* inklusive Compiler und Gerätetreiber für den Programmieradapter können frei bei Atmel [Atm12] heruntergeladen werden. In dieser Projekgruppe wurde das Atmel Studio in Version 6.0 verwendet.

Atmel Studio 6.0

Abbildung 5.8: Programmieradapter von Atmel (AVRISP mkII) zum Aufspielen von Firmware auf Mikrocontroller.

5.3.1.1 *Servofirmware*

Die Servofirmware befindet sich im Repository unter

```
servo.naodevils2011.git/OpenServo/AVR_OpenServo
```

und kann durch das Projekt `ATmega168_OpenServo.avrgccproj` modifiziert und neu kompiliert werden. Hierzu muss beim geöffneten Projekt zunächst der korrekte Pfad zur *Makefile* sichergestellt werden. Diese befindet sich im Projektverzeichnis und kann in den Projekteinstellungen unter *Build* bei *Use External Makefile* angegeben werden. Das Kompilieren wird durch das Drücken der Taste F7, alternativ über *Build* > *Build Solution* initiiert und erzeugt die Firmwaredatei `ATmega168_OpenServo.hex`.

externes Makefile

Aufbau zum Flashen

Zum Aufspielen der Firmware, muss zunächst der Programmieradapter (Abbildung 5.8) an Computer (USB) und OSIF (SPI, vgl. Abbildung 5.7) angeschlossen werden. An das OSIF wird wiederum der zu programmierende Servo mit dem zehnpoligen Steckverbinder verbunden. Dabei ist zu beachten, dass der Jumper *J1* am OSIF geöffnet ist, da ansonsten der Mikrocontroller des OSIF reprogrammiert wird. Das OSIF muss zusätzlich über USB am Computer angeschlossen werden, um dem Servo eine Versorgungsspannung zu liefern. Auf die 12 V Versorgung kann und soll verzichtet werden, da der Motor sonst unvorhersehbar angesteuert wird.

J1 *geöffnet!*

Aufspielen der Firmware

Sobald der Servo angeschlossen ist, kann im Atmel Studio das Programmiermenü (*Tools* > *AVR Programming*) aufgerufen werden. In der erscheinenden Maske sind als Tool *AVRISP mkII*, als Device *ATmega328P*[1] und als Interface *ISP* auszuwählen. Zur Kontrolle können die *Device ID* und *Target Voltage* ausgelesen werden, die in einer Fehlermeldung resultieren, sofern die Verbindung oder Einstellungen fehlerhaft sind.

1 Obwohl das Projekt den Namen des Mikrocontrollers *ATmega168* trägt, befindet sich auf der OpenServo Platine ein *ATmega328P*.

Fuse Register

Wird der Mikrocontroller zum ersten Mal programmiert, so müssen zunächst einmalig die Fuse Register unter *Fuses* korrekt eingestellt werden. Dies sind Bits, die die grundlegende Funktionalität des Mikrocontrollers bestimmen. Dazu zählen zum Beispiel die gewünschte Betriebsfrequenz, oder ob ein interner oder externer Taktgeber verwendet werden soll. Bei falscher Konfiguration kann der Mikrocontroller unbrauchbar gemacht werden. Die korrekten Werte der Fuse Register lauten:

Fuse Register	Value
EXTENDED	`0xFD`
HIGH	`0xDD`
LOW	`0xE2`

Übertragung der Firmware

Sobald die Fuse Register richtig gesetzt sind, kann die Firmware übertragen werden. Dazu muss in der gleichen Maske unter *Memories* > *Flash* die Firmwaredatei `ATmega168_OpenServo.hex` ausgewählt und durch Betätigung der Schaltfläche *Program* aufgespielt werden. Anschließend kann die Maske geschlossen, der Programmieradapter entfernt und Jumper J_1 geschlossen werden. Damit ist der Servo betriebsbereit.

Servo entspricht virtuellem Speicher

aufgeteilt in Register

Die Kommunikation mit dem Servo erfolgt über I^2C durch Lesen und Setzen virtueller Register, die sich auf dem Servo befinden. Vereinfacht ausgedrückt stellt der Servo einen virtuellen Speicher mit kontinuierlichem Adressraum dar, in dem bestimmten Adressen spezielle Werte oder Funktionalitäten aufweisen. Ein Register entspricht einer Speicheradresse im Servo und behinhaltet ein Byte. Allerdings sind einige Register aus zwei benachbarten Bytes (High-Byte und Low-Byte) zusammengesetzt, um größere Werte speichern zu können. So entsprechen die zwei Bytes ab Adresse `0x08` der aktuell gemessenen Servoposition und das Bytepaar ab `0x10` der Servoposition, die angefahren werden soll. Eine vollständige Auflistung aller Register ist im Repository unter

`servo.naodevils2011.git/OpenServo/TWIProtocol-OpenServoWiki.pdf`

zu finden. In dem Dokument wird zudem die genaue Kommunikation über I^2C erklärt, auf die hier nicht weiter eingegangen werden soll.

5.3.1.2 *OSIF-Firmware*

Das OSIF war zum Abschluss der Projektgruppe bereits korrekt programmiert und bedarf keinem erneuten Aufspielen der Firmware, da es nur ein Werkzeug zur Kommunikation mit dem Servo darstellt, jedoch keinen Einfluss auf die Funktionsweise der Servos selbst hat.

Sollte es dennoch notwendig sein, die Firmware erneut aufzuspielen, z.B. falls der Mikrocontroller durch Verpolung oder Elektrostatik zerstört und ersetzt wurde, so können die nachfolgenden Schritte durchgeführt werden.

Die Firmware für das OSIF befindet sich im Repository unter

```
servo.naodevils2011/OpenServo/Interfaces/OpenServo_
    InterFace/driver/Version_2/firmware/main.hex
```

J1 geschlossen!

und kann direkt mit dem Atmel Studio aufgespielt werden. Dazu muss der Programmieradapter (Abbildung 5.8) bei geschlossenem Jumper J_1 (vgl. Abbildung 5.7) an das OSIF angeschlossen werden. Zusätzlich muss das OSIF über USB mit einer Betriebsspannung versorgt werden. Die 12 V Versorgung wird nicht benötigt. Es ist zu Beachten, dass während der Programmierung kein Servo mit dem OSIF verbunden ist, da dieser sonst ebenfalls mit einer für ihn ungültigen Firmware programmiert wird.

OSIF zusätzlich über USB anschließen

keinen Servo anhängen!

Sobald das OSIF zum Programmieren bereit ist, kann im Atmel Studio bei einem leeren Projekt das Programmiermenü (*Tools* > *AVR Programming*) aufgerufen werden. Analog zum vorherigen Abschnitt sind in der erscheinenden Maske als Tool *AVRISP mkII*, als Device *ATmega8* und als Interface *ISP* auszuwählen. Zur Kontrolle können die *Device ID* und *Target Voltage* ausgelesen werden, die in einer Fehlermeldung resultieren, sofern die Verbindung oder Einstellungen fehlerhaft sind.

Fuse Register

Nun müssen die Fuse Register unter *Fuses* wie folgt eingestellt werden:

Fuse Register	Value
EXTENDED	Wert belassen
HIGH	`0xC8`
LOW	`0x9F`

Hinweis: Falsche Fuse-Werte können den Mikrocontroller unbrauchbar machen.

Aufspielen der OSIF-Firmware

Sobald die Fuse Register richtig gesetzt sind, kann die Firmware übertragen werden. Dazu muss in der gleichen Maske unter *Memories* > *Flash* die o.g. Firmwaredatei ausgewählt und durch Betätigung der Schaltfläche *Program* aufgespielt werden. Damit ist das OSIF einsatzbereit.

5.3.2 *Installation*

OSIF anschließen, dann Treiber generieren

Da die Treiberbibliothek von Windows keinen Treiber für das OSIF bereitstellt, muss dieser manuell erzeugt und installiert werden. Dazu wird zunächst das OSIF über USB am Computer angeschlossen und anschließend mit dem *inf-wizard* aus dem Repository unter

```
servo.naodevils2011.git/OpenServo/Interfaces/OpenServo_
    InterFace/driver/Version_2/dll/WindowsToolchain/
              libusb-win32-bin-1.2.6.0/bin
```

eine inf-Datei erzeugt. Die einzelnen Schritte des Assistenten sind durch Screenshots beschrieben und finden sich im Repository unter

```
servo.naodevils2011.git/OpenServo/Interfaces/OpenServo_
    InterFace/driver/Version_2/dll/WindowsToolchain.
```

Die generierte inf-Datei wird im Anschluss als Treiber für das unbekannte Gerät installiert.

Installation vom MinGW Compiler

Im nächsten Schritt müssen Bibliotheken für den Zugriff auf den Gerätetreiber erzeugt und installiert werden. Dies erfordert die Installation des MinGW Compilers. Ein Installationsprogramm hierzu befindet sich unter

```
servo.naodevils2011.git/OpenServo/Interfaces/OpenServo_
    InterFace/driver/Version_2/dll/WindowsToolchain/
              mingw-get-inst-20120426.exe
```

und lädt die erforderlichen Dateien aus dem Internet nach. Als Installationsverzeichnis sollte ein Pfad ohne Leerzeichen ausgewählt werden, z.B. `C:\MinGW`. Nach der Installation des Compilers muss die Windows Umgebungsvariable `path` um `C:\MinGW\bin;` erweitert werden. Die Einstellung wird unter Windows 7 direkt übernommen und erfordert kein erneutes Anmelden des Benutzers. Anschließend muss LibUSB installiert werden, da die zu kompilierende Bibliothek darauf aufbaut. Dazu muss die Datei aus dem Repository

```
servo.naodevils2011.git/OpenServo/Interfaces/OpenServo_
    InterFace/driver/Version_2/dll/WindowsToolchain/
       libusb-win32-bin-1.2.6.0/lib/gcc/libusb.a
```

in das Verzeichnis `C:\MinGW\lib` kopiert werden. Ebenso ist die Datei

```
servo.naodevils2011.git/OpenServo/Interfaces/OpenServo_
    InterFace/driver/Version_2/dll/WindowsToolchain/
    libusb-win32-bin-1.2.6.0/bin/x86/libusb0_x86.dll
```

dll-Datei umbenennen!

in das Verzeichnis `C:\Windows\SysWOW64` zu kopieren. Diese muss im Zielverzeichnis in `libusb0.dll` umbenannt werden.

Nun kann mit der Eingabeaufforderung in das Verzeichnis

```
servo.naodevils2011.git/OpenServo/Interfaces/OpenServo_
              InterFace/driver/Version_2/dll
```

navigiert werden und

```
> make -f Makefile.win32dll
```

aufgerufen werden. Zur Überprüfung kann

```
> maketestapp.bat
> testapp.exe scan
```

gestartet werden, was bei korrekter Kompilation, und nicht verbundenem OSIF folgendes ausgibt: „Failed to open OSIF device. Check cables“. Bei misslungener Kompilation stürzt die Anwendung ohne Meldung ab. Zum Abschluss der Installation wird die erfolgreich generierte `OSIFdll.dll` Datei nach `C:\Windows\SysWOW64` kopiert. Dadurch ist nun der Zugriff auf das OSIF über Funktionen der Bibliothek möglich.

5.3.3 *Framework*

Für die einfache Ansteuerung des Servos wurde ein Framework in C++ implementiert, welches Betriebssystemunabhängig ist und leicht erweitert werden kann. Es befindet sich im Repository unter

`servo.naodevils2011.git/OpenServoGUI.`

abstrakte Fabrik

Das Framework ist objektorientiert und richtet sich nach dem Entwurfsmuster der abstrakten Fabrik. Dabei wurden möglichst viele einheitliche Funktionalitäten in abstrakten Superklassen gekapselt, deren Spezialisierungen dann die fehlenden Teile für verschiedene Anforderung vervollständigen. Dies ermöglicht gleichzeitig eine hohe Flexibilität für Erweiterungen und eine sehr einfache Bedienbarkeit, da der Benutzer unabhängig von der verwendeten Schnittstelle und dem verwendeten Betriebssystem in jedem Fall mit Servo-Objekten arbeitet, die ihm von einer Fabrik erzeugt werden und sich äußerlich nicht unterscheiden. Da intern *shared pointer* verwendet werden, die jedoch nachträglich zum C++ Standard hinzugefügt wurden, lässt sich das Framework mit Visual Studio erst ab Version 2008 mit installiertem Service Pack 1 kompilieren.

Die Struktur des Frameworks ist in einem Klassendiagramm (Abbildung 5.9) dargestellt. Es lässt sich analog zu Netzwerken in Schichten unterteilen. Die oberste Schicht bildet die Anwendungsschicht, die im Diagramm durch die Klassen *Servo* und *AbstractServoFactory*, bzw. eine ihrer Spezialisierungen repräsentiert werden. Der Benutzer verwendet Objekte vom Typ *Servo* und kann auf diesem Objekt Methoden aufrufen, um den Servo zu konfigurieren oder Informationen auszulesen, z.B. *enablePWM* oder *getPosition*. Diese Servo-Objekte werden jedoch nicht selbstständig angelegt, sondern von einer Servo Fabrik generiert.

Anwendungsschicht

Transportschicht

Darunter liegt die Transportschicht, die durch die Klasse *AbstractServoInterface* repräsentiert wird. Ihre Spezialisierungen kommunizieren über Nachrichten (*AbstractServoMessage*) mit Servo-Objekten und transformieren diese in Bytefolgen, die der darunter liegenden Bibliothek, bzw. dem darunter liegenden Gerätetreiber weitergegeben werden. So initiiert das *OsifDllInterface* eine Verbindung zur `OSIFdll.`

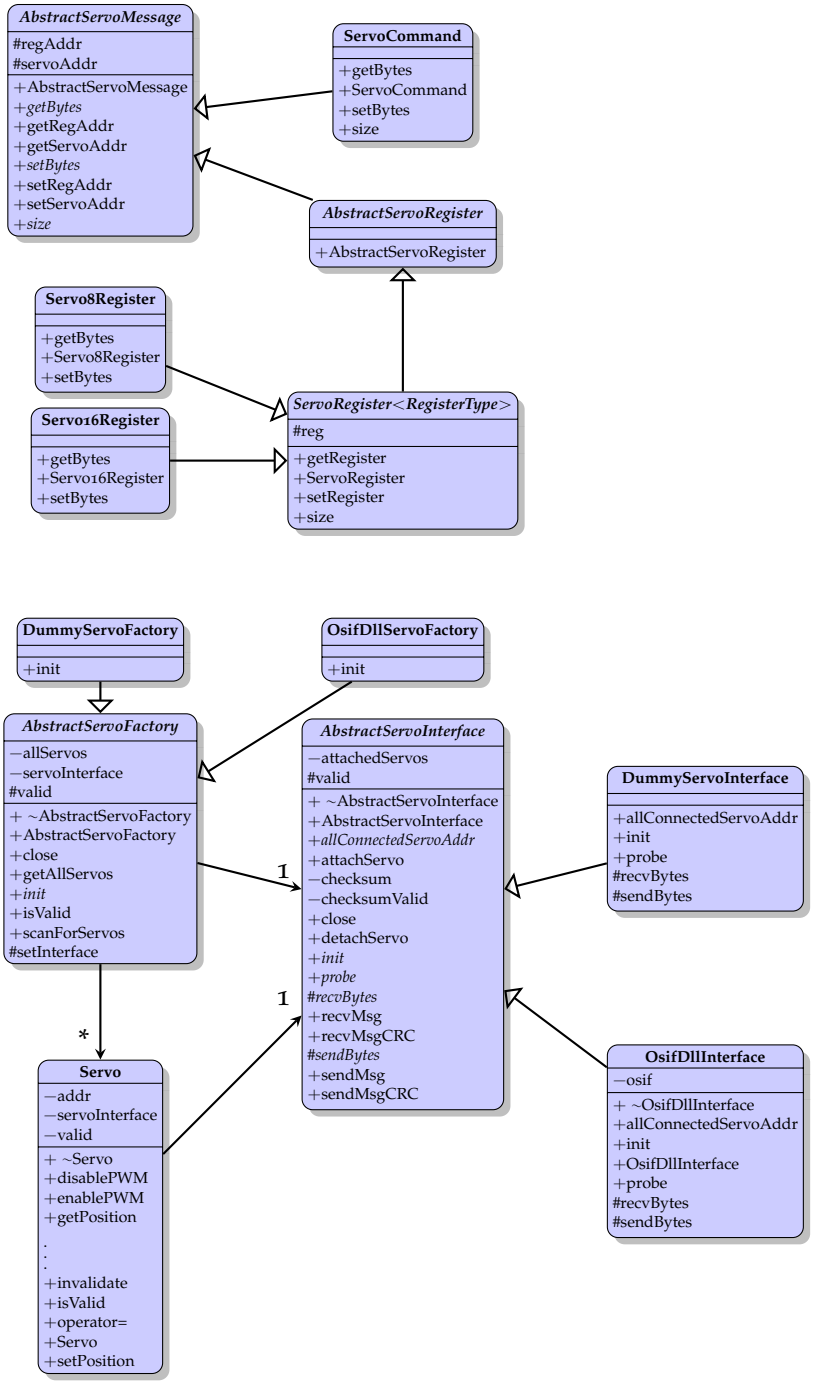


Abbildung 5.9: Vereinfachtes Klassendiagramm des Frameworks zur einfachen Ansteuerung und Erweiterung von OpenServo. Auf Anwendungsebene erzeugt sich der Benutzer eine konkrete Servo Fabrik, die Servo-Objekte generiert. Auf diesen Objekten können durch Methodenaufrufe Interaktion mit dem jeweiligen Servo durchgeführt werden. In der Transportschicht wandelt das Servo Interface Nachrichten vom Servo-Objekt in Bytefolgen um, die an den Gerätetreiber weitergegeben, bzw. empfangen werden.

`dll`, deren Installation im vorherigen Abschnitt erläutert wurde, und reicht dadurch Bytefolgen durch das OSIF zum angeschlossenen Servo weiter. Es stellt außerdem die Integrität der Daten durch Prüfsummen sicher und kümmert sich um die korrekte Adressierung des gewünschten Servos.

Verwendung des Frameworks

Zur Verwendung der Frameworks und der Ansteuerung von Servos muss im Programm zunächst eine konkrete Servo Fabrik erzeugt werden. Diese konstruiert ein zur Fabrik passendes Servo Interface, was die passenden Gerätetreiber initialisiert. Die Fabrik kann anschließend über das Servo Interface nach angeschlossenen Servos suchen und erzeugt dabei für jeden gefundenen Servo ein Servo-Objekt, die der Anwender mit der Methode *getAllServos* erhalten kann, und assoziiert diese mit dem darunter liegenden Servo Interface. Ruft der Anwender auf dem Servo-Objekt Methoden auf, so generiert dieses passende Nachrichten und gibt diese an das Interface weiter, welches sich um die weitere Verarbeitung kümmert. Servo-Objekte können beliebig kopiert und gelöscht werden, verlieren jedoch alle ihre Funktionalität (*valid*), sofern ihre Erzeugerfabrik oder das darunter liegende Interface geschlossen wird. Durch den internen Einsatz von *shared pointern* wird beim Löschen von Fabrik und Servos sichergestellt, dass das Servo Inteface erst beim Löschen des letzten Servos aus dem Speicher entfernt wird und es zuvor zu keinen ungültigen Methodenaufrufen auf nicht existenten Objekten kommt. Um Verwirrung zu Vermeiden: Der Anwender ist beim Halten der Servos und der Fabrik nicht an die *shared pointer* gebunden, da dies nur für die interne Verwaltung notwendig ist. Er kann beliebige Zeiger und Referenzen benutzen.

Erweiterung des Frameworks

Für Erweiterung des Frameworks können zusätzliche Spezialisierungen des *AbstractServoInterface* hinzugefügt werden, sodass auch andere Schnittstellen als das bereits fertiggestellte *OsifDllInterface* über OSIF zum Servo unterstützt werden. Gleichzeitig muss auch eine entsprechende Spezialisierung der Servo Fabrik hinzugefügt werden, die jedoch nur die Implementierung der *init* Methode zur Erzeugung des Interface erfordert. Auf Anwendungsschicht kann auch der Servo um zusätzliche Methoden erweitert werden. Die hierbei erforderliche Kommunikation mit dem Servo beschränkt sich lediglich auf die Weitergabe von Objekten einer Unterklasse von *AbstractServoMessage* an das darunter liegende Interface.

Anwendungsbeispiel

Abbildung 5.10 zeigt einen exemplarischen Quelltext für die Verwendung des Frameworks mit dem OSIF. Zunächst wird eine Servofabrik angelegt, nach Servos gesucht und diese in einem Vektor zwischengespeichert. Anschließend wird der Motor des ersten Servos aktiviert, die mittlere Servostellung (Position 500) angefahren und der Motor wieder deaktiviert. Zum Schluss wird die Fabrik geschlossen und gelöscht. Da der Vektor mit den Servos auf dem Stack liegt, wer-

```
#include "AbstractServoFactory.h"
#include "OsifDllServoFactory.h"
#include "Servo.h"

void example()
{
  AbstractServoFactory *factory;
  std::vector<Servo> servos;

  factory = new OsifDllServoFactory();  // Prepare Factory
  factory->init();
  factory->scanForServos(); // Scan for connected servos
  servos = factory->getAllServos();
  if(servos.size() < 1)
  {
    exit(0xC0CAC01A); // No servos connected
  }
  // ...
  servos[0].enablePWM();  // Enable servo's motor
  // ...
  servos[0].setPosition(500); // Drive to position 500
  sleep(1000);
  // ...
  servos[0].disablePWM(); // Disable servo's motor
  // ...
  factory->close(); // Cleanup
  delete factory;
}
```

Abbildung 5.10: Beispielcode zur Ansteuerung eines Servos über OSIF mit Hilfe des entwickelten Frameworks.

den die angehängten Servos beim Beenden der Methode automatisch freigegeben.

GUI zur Steuerung eines Servos

Für erste Tests wurde eine GUI erstellt, die die grundlegenden Funktionen des Servos über das OSIF benutzbar macht (Abbildung 5.11). Die Funktionen, die mit diesem Programm angesteuert werden können, sind bisher das Setzten der Position des Servos (über einen Schieberegler oder über eine Scroll-Box), das Setzen der PID-Parameter des Servos und das Messen der aktuellen Position. Die Benutzeroberfläche basiert dabei auf dem QT-Framework von Nokia [QT212] in Version 4.8.2.

5.4 VERSUCH ZUR DREHMOMENTBESTIMMUNG

Bei der Ansteuerung eines Feed-Forward-Controllers wird vorab ein erwartetes Drehmoment angegeben, das an der Zielposition vom Motor aufgewendet werden muss. Das Drehmoment des Motors wird jedoch vom Controller durch Pulsweitenmodulation, engl. Pulse Width

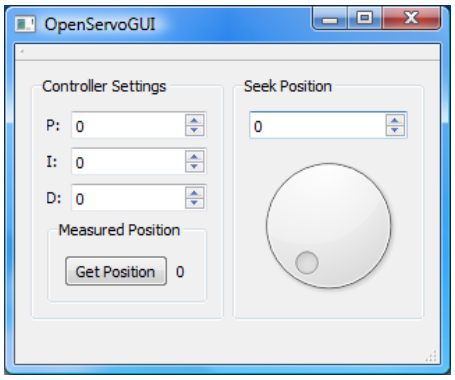
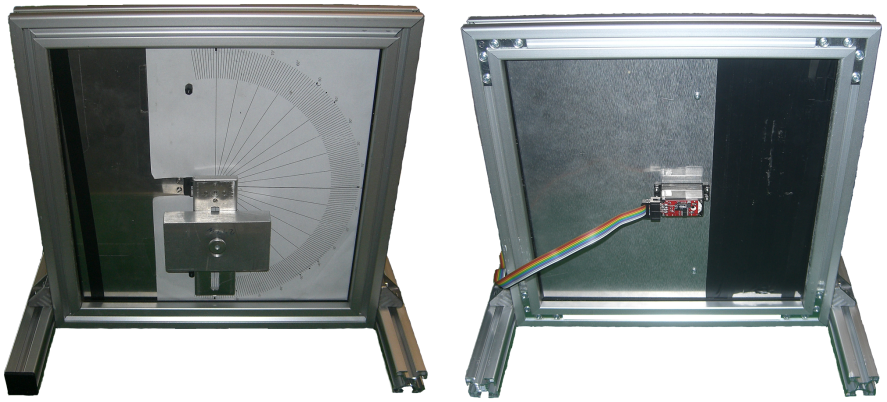


Abbildung 5.11: GUI für das OSIF Framework. Über den großen Schieberegler rechts lässt sich die Position des Servos anfahren. Über die Textfelder links oben können die PID-Reglerparameter gesetzt werden. Die aktuell gemessene Position des Servos kann über den Knopf unterhalb der Textfelder abgefragt werden.



(a) Vorderseite. (b) Rückseite.

Abbildung 5.12: Versuchsaufbau des Servos zur Bestimmung der Drehmomente je nach Position. Der Servo ist an einer senkrechten Metallplatte befestigt und bewegt einen Hebel mit Gewicht.

PWM

Modulation (PWM) reguliert, anstatt direkt den Stromfluss durch den Motor zu regulieren. Dabei wird der Motor stets mit dem Höchststrom betrieben, der periodisch für Intervalle bestimmter Länge vollständig unterbrochen wird. Das Verhältnis d zwischen Periodenlänge T und der Dauer t, in der Strom durch den Motor fließt ergibt dann den mittleren Stromfluss

$$\bar{I} = I_0 \cdot d = I_0 \cdot \frac{t}{T} \tag{5.1}$$

durch den Motor. Dieser ist bei hinreichend hoher Frequenz und konstantem t aufgrund der Induktivität des Motors annähernd konstant.

Drehmoment je nach Stellwinkel

Um den genauen Zusammenhang zwischen dem PWM-duty d und dem resultierenden Drehmoment festzustellen, wurde der Servo wie in Abbildung 5.12 an einer senkrecht aufgestellten Metallplatte befestigt. Der Servo bewegt dabei einen Hebel mit einem Gewicht. Die Entfernung des Gewichtes und die Position der Metallplatte im Rahmen sind variabel. Das Drehmoment M am Servo ergibt sich aus

$$M = r \cdot F \cdot \sin \alpha, \tag{5.2}$$

wobei r für die Entfernung des Gewichts von der Achse steht und F die Kraft durch das Gewicht angibt. Diese entspricht im statischen Fall der senkrecht nach unten wirkenden Schwerkraft. Der Winkel α gibt die Auslenkung des Hebels von der Ruhelage 0° (senkrecht nach unten) an, sodass das Drehmoment in der Ruhelage $0\,\mathrm{N\,m}$ entspricht, und bei $\pm 90°$ im Betrag maximal ist.

Versuchsmessung

Im Versuch wurden zwei Messungen durchgeführt, in der der Servo die Winkel im Intervall von ca. $-80°$ bis $80°$ angefahren hat. Dabei wurde im Servo ein PID Regler mit $400\,\mathrm{Hz}$ und folgenden Parametern verwendet:

Parameter	Wert
P	`0x0600`
I	`0x0000`
D	`0x1800`

Bei jedem angefahrenen Winkel wurde zunächst eine Sekunde lang gewartet, bis sich das Gewicht in Ruhelage befand. Anschließend wurde pro Position in Abständen von $20\,\mathrm{ms}$ die aktuelle Position und PWM-duty aus dem Servo ausgelesen und der Mittelwert gespeichert. Die Ergebnisse des Versuchs sind in Abbildung 5.13 abgebildet.

Sinusförmiger Verlauf

Es zeigt sich, dass der PWM-duty trotz starker Schwankungen einen sinusförmigen Verlauf aufweist und sich somit linear zum Drehmoment aus Gleichung 5.2 verhält. Zur Verdeutlichung ist eine an die Größenordnung angepasste Sinuskurve gestrichelt in das Diagramm eingezeichnet. Der von Versuch zu Versuch stark streuende PWM-duty erschwert die Entwicklung eines zuverlässigen FFPID-Reglers, da ein

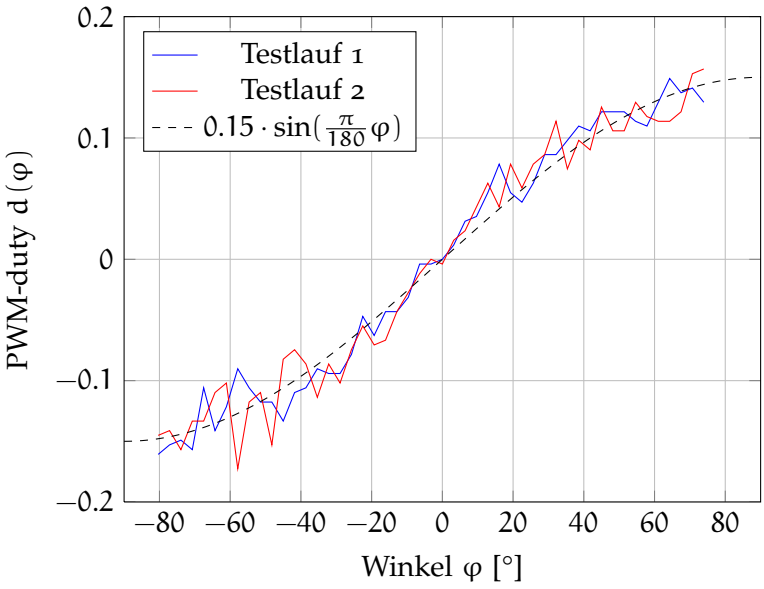


Abbildung 5.13: PWM-duty des Servos in Abhängigkeit vom angefahrenen Winkel φ. Jeder Messpunkt beider Testläufe (rot, blau) entspricht dem Mittelwert aus jeweils 64 Messungen an einer Position. Der Verlauf folgt einer Sinuskurve (gestrichelt) entsprechend zu Gleichung 5.2, weist jedoch eine ausgeprägte Streuung auf.

mögliche Ursachen für Streuung

bestimmter Wert in einem sehr breiten Winkelintervall streut. Mögliche Ursachen für die Streuung sind nicht optimale Parameter für den PID-Controller oder Messfehler beim Bestimmen der Position des Servos durch das verwendete Potenziometer, die zu einem „Zappeln“ um die Zielposition führen. Leider genügte die Zeit im Rahmen der Projektgruppe nicht für eine Untersuchung dieser Aspekte.

5.5 ZUSAMMENFASSUNG UND FAZIT

In diesem Kapitel wurde gezeigt, dass bei der Regelung von Servos durch das Vorabwissen von Drehmomenten an der Zielposition diese schneller angefahren werden kann. Dies kann durch den FFPID-Regler, einer Hybriden Form aus PID-Regler und einer Steuerung realisiert werden. Für Experimente war es deshalb notwendig, einen solchen Regler in einen Servo zu integrieren. Da das Umprogrammieren vorhandener Standardservos nicht möglich ist, wurde die gesamte Elektronik des Servos durch eine Platine aus dem OpenServo Projekt ersetzt und mit einer eigenen Firmware programmiert. Leider weist der so modifizierte Servo aufgrund elektrischer Probleme ein unberechenbares Verhalten auf, was in einer aufwendigen Fehlersuche resultierte. Die Lösung bringt ein zusätzlicher Tantalkondensator

über den Versorgungsspannungsbeinchen des Mikrocontrollers, der Spannungseinbrüche abblockt und den Chip betriebsfähig hält.

Da der Servo über die I^2C Schnittstelle mit der Außenwelt kommuniziert, wurde das OSIF als Verbindungsstück zwischen Computer und Servo aufgebaut. An dieser Stelle ergaben sich mangels Dokumentation Probleme bei der Treiberinstallation und Benutzung des OSIF unter Windows, weswegen in diesem Kapitel ausführlich darauf eingegangen wird.

Schließlich wurde in der verbleibenden Zeit ein übersichtliches und einfaches Framework zur Bedienung der Servos über das OSIF implementiert. Außerdem war es möglich, erste Messungen des PWM-duty in Abhängigkeit von der Position durchzuführen. Dieser weist starke Streuungen auf, verhält sich jedoch linear zum physikalischen Modell des Drehmomentes. Die Ursache für die Streuungen konnte jedoch nicht untersucht werden, und bietet eine Möglichkeit für weitere Forschung auf diesem Gebiet. Ebenso blieb keine Zeit für die tatsächliche Implementierung eines FFPID-Controllers. Aus diesem Grund soll dieses Kapitel einen Einstieg in die Thematik in theoretischer, hard- und softwaretechnischer Sicht bieten und die gewonnenen Erkenntnisse für künftige Arbeiten dokumentieren und reproduzierbar gestalten.

6 BEWEGUNGSPLANUNG

Die Bewegungsplanung behandelt das Umsetzen von Bewegungen auf Grund einer Verhaltensvorgabe. Diese Planung findet unter Berücksichtigung von Informationen aus der Wahrnehmung statt. So müssen Hindernisse auf dem Spielfeld beachtet werden und ein möglichst schneller Weg zum Ziel gefunden werden. Ziel ist hierbei eine schnelle, aber immer noch stabile Bewegung vorzugeben.

6.0.1 *Motivation*

Erfahrungen und Beobachtungen aus dem bisherigen Roboterverhalten zeigten, dass die Planung sich dem Ball zu nähern, sowie das Umgehen von Hindernissen oft umständlich verläuft und in einigen Fällen nicht funktioniert. Defizite dies bezüglich waren auf Wettkämpfen, wie der RoboCup German Open 2012 (in Magdeburg) (German Open 2012), zu beobachten. Die Projektgruppe erhofft sich durch neue Ansätze im Bereich der Bewegungsplanung deutliche Fortschritte im Spielablauf.

Das Thema umfasst zwei Bereiche. Der erste Bereich umfasst die Verbesserung der Pfadplanung. Deren bisheriger Ansatz basiert auf einem Potentialfeld und soll nun durch eine Alternative ersetzt werden. Die Anforderung an diese ist, einen möglichst stabilen Pfad zu erzeugen, ohne sich dabei zu verfangen. Der zweite Bereich ist die Neuerstellung eines *Pattern Generators*. Die bisherige Implementierung wandelt einen Pfad in Geschwindigkeiten um, welche der *Pattern Generator* in Fußstapfen umsetzt. Der Zwischenschritt soll entfallen und statt dessen aus dem Pfad direkt Fußstapfen erzeugt werden, so dass eine Ansteuerung ohne Umwege entsteht. Dadurch wird es denkbar den Roboter genauer zu platzieren, um z.B. einen direkten Schuss auszuführen.

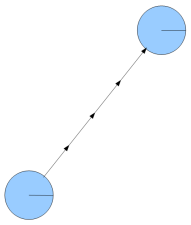
6.1 PFADPLANUNG

Im Bereich der Bewegungsplanung soll durch die Projektgruppe unter anderem die Pfadplanung verbessert werden. Die Pfadplanung trägt einen wichtigen Anteil an der Geschwindigkeit des eigenen Spiels. Nicht nur die Laufgeschwindigkeit des Roboters ist wichtig, um beispielsweise als erster am Ball zu sein, sondern auch die Art und Weise wie zum Ball gelaufen wird („kann er zielgerichtet laufen?“, „muss er sich drehen oder seitlich laufen?“, „muss er einem

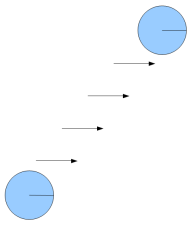
Hindernis ausweichen?“, usw.). Genau diesen Teil übernimmt die Pfadplanung. Der Pfad muss gewissen Anforderungen genügen.

1. Der Pfad soll möglichst kurz sein.
2. Gleichzeitig muss Hindernissen ausgewichen werden.
3. Der Pfad soll auf die Bewegungsgeschwindigkeiten des NAO Rücksicht nehmen.

Der letzte Teil bedeutet, dass der Pfad einrechnet, wie schnell der Roboter sich in verschiedene Richtungen bewegen kann (vgl. Abb. 6.1). Dazu kann angenommen werden, dass der Roboter sich bidirektional langsamer bewegt als bei einem unidirektionalen Lauf geradeaus. Bei der Planung des Pfades muss daraufhin entschieden werden, ob der Roboter unidirektional mit Drehungen (Abb. 6.1a) oder bidirektional (Abb. 6.1a) laufen soll.



(a) Unidirektionaler Lauf - Der Roboter dreht sich zunächst auf der Stelle, läuft auf die Zielposition und dreht sich in Blickrichtung.

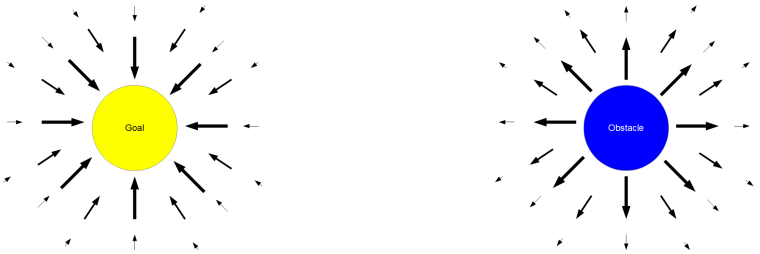


(b) Bidirektionaler Lauf - Der Roboter läuft schräg von seiner aktuellen Position auf die Zielposition.

Abbildung 6.1: Vergleich von uni- und bidirektionalem Lauf

Aktuell wird ein Potentialfeld für die Pfadplanung genutzt. Besonders Punkt 3 (s.o.) ist in einem Potentialfeld schwer zu modellieren. Außerdem bestehen weitere Probleme, die im folgenden Kapitel aufgezeigt werden. Deshalb soll das Potentialfeld durch einen, auf einem Rapidly-Exploring Random Tree (RRT) basierenden Algorithmus ersetzt werden. Die Umgebung auf der dieser Algorithmus arbeitet, besteht aus dem Feld, auf dem gespielt wird und den sich darauf befindlichen Hindernissen. Beim RRT-Pfadplanungsalgorithmus wird der Pfad aus dem RRT, einer Baumstruktur, aufgebaut, der vom Startpunkt bis zum Zielpunkt wächst und dabei Hindernissen ausweicht. Dieser Algorithmus wird bereits in der Small-Size-League (SSL) verwendet [BV02]. Der Startpunkt ist allerdings nicht die aktuelle Roboterposition, sondern der Roboterposition nach Durchlaufen der Previewphase der im Framework eingebauten Walking-Engine. In der Previewphase läuft der NAO schon definierte Fußstapfen aus einem Puffer ab; der aktuelle Pfad kann deren Richtung nicht beeinflussen.

6.1.1 *Status-Quo*



(a) Anziehendes Potential (b) Abstoßendes Potential

Abbildung 6.2: Anziehende und abstoßende Potentiale

Zur Zeit übernimmt ein Potentialfeld die Pfadplanung. In einem Potentialfeld werden Hindernisse als abstoßende Potentiale (vgl. Abb. 6.2b) modelliert, während der Zielpunkt ein anziehendes Potential (vgl. Abb. 6.2a) darstellt. Jedes Potential besitzt eine individuelle Stärke mit der es das Potentialfeld beeinflusst. Durch diese attraktiven (anziehende Potentiale) und unattraktiven (abstoßende Potentiale) Gebiete versucht der Roboter zu planen. Hierzu werden alle Potentiale, die an einer bestimmten Position des Potentialfeldes wirken, aufsummiert [MLLo8]. Der Pfad wird in Richtung des größten Gradientenabstiegs geplant, sofern abstoßende Potentiale als positive und anziehende Potentiale als negative Feldstärken modelliert werden. Jeder Roboter auf dem Feld stellt ein Hindernis dar. Um zu verhindern Hindernisse zu durchlaufen, muss in einem bestimmten Gebiet um das Zentrum des Hindernisses herum das abstoßende Potential so groß sein, dass ein Pfad unmöglich durch dieses Gebiet geplant werden kann. Im weiteren Distanzbereich des Roboters wirkt das Potential schwächer, bis es, ab einer gewissen Distanz zum Zentrum, das Potentialfeld nicht nennenswert beeinflusst. Ein Problem, das bei dieser Form der Pfadplanung auftreten kann, stellen lokale Minima dar, in die der Pfad hinein und nicht wieder hinaus führt. Das bedeutet, dass an einer Position des Potentialfeldes der Gradient null wird. An dieser Position kann also keine Richtung in die weiter geplant werden soll, gefunden werden. Es gibt mehrere Möglichkeiten auf lokale Minima zu reagieren. Hier einige Beispiele:

1. Entlang des Pfades aus dem lokalen Minimum heraus laufen und eine Strategie zum Vermeiden des lokalen Minimum verwenden.

2. Randomisierte Bewegungen ausführen, um das lokale Minimum zu verlassen.

3. Komplexere Potentialfelder verwenden, in denen keine lokalen Minima auftreten. Zum Beispiel können nachträglich an Stellen mit lokalen Minima Wirbelfelder (Tangentialfelder; vgl. Abb. 6.3) eingefügt werden, die den Pfad um das lokale Minimum herum führen.

Diese Ansätze gehen davon aus, dass der Roboter erkennt, dass er in einem lokalen Minimum gefangen ist; ein weiteres Problem. Um diese Schwierigkeiten zu umgehen, soll ein anderes Verfahren zur Pfadplanung, in diesem Fall der RRT-Pfadplanungsalgorithmus, implementiert werden.

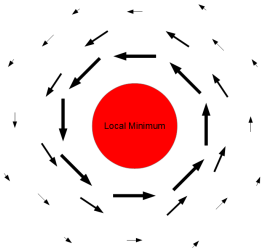


Abbildung 6.3: Tangentialfeld

6.1.2 *Funktionsweise der RRT-Pfadplanung*

Als Datenstruktur benutzt der RRT einen *k-d-tree*. Ein *k-d-tree* ist eine Baumdatenstruktur in der x-dimensionale Punkte, geordnet nach ihren Koordinaten, gespeichert werden können. Die Knoten des *k-d-tree* sind in diesem Fall dreidimensionale Punkte auf dem Spielfeld, X- und Y-Koordinate, sowie deren Rotation und besitzen jeweils zwei Kinder. Die Kanten eines *k-d-tree* unterteilen die Punkte in ihren jeweiligen Dimensionen. Dabei wird zunächst nach der ersten, zweiten,... Dimension unterteilt. Wurde in der letzten Dimension unterteilt, wird wieder mit der ersten Dimension begonnen. In diesem speziellen Fall ist die Wurzel des Baumes der Startpunkt. Die Kante zum ersten Kind führt zu den Punkten auf dem Feld die eine kleinere X-Koordinate besitzen, während die Kante zum zweiten Kind zu den Punkten mit größerer oder gleicher X-Koordinate führt. Die Kanten von diesen Kindern ausgehend teilen die Punkte anhand ihrer Y-Koordinate auf. Anschließend wird mit der Rotation fortgefahren bevor wieder mit einer Unterteilung durch die X-Koordinate begonnen wird.

Der Kern des Algorithmus besteht aus folgenden drei Methoden, die iterativ durchlaufen werden:

GETTARGET wählt die Richtung in der ein neuer Punkt erzeugt wird.

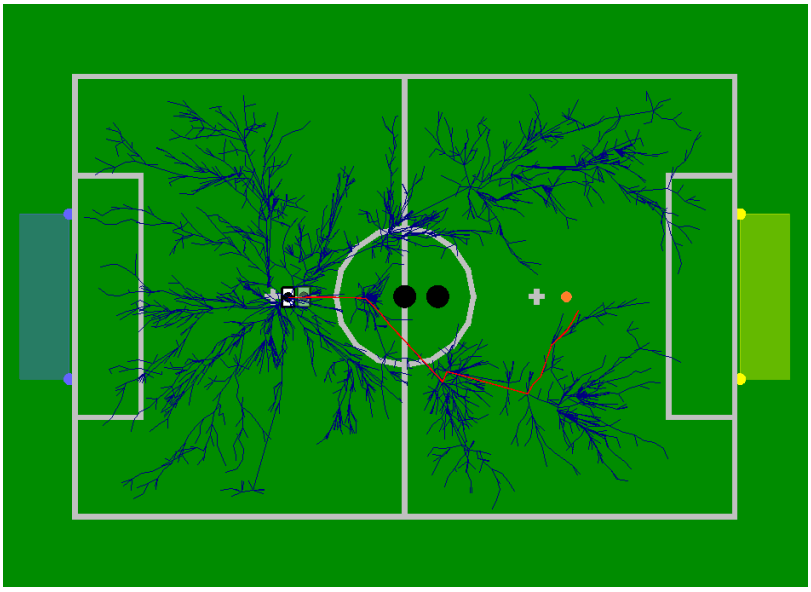
GETNEAREST gibt aus dem Baum den Knoten an, der den geringsten Abstand zum neuen Punkt hat.

EXTENDTO überprüft, ob zu dem gewünschten Punkt expandiert werden kann. Es können weitere sinnvolle Entscheidungen darüber getroffen werden, ob und in welcher Form ein neuer Punkt in den Baum eingefügt wird.

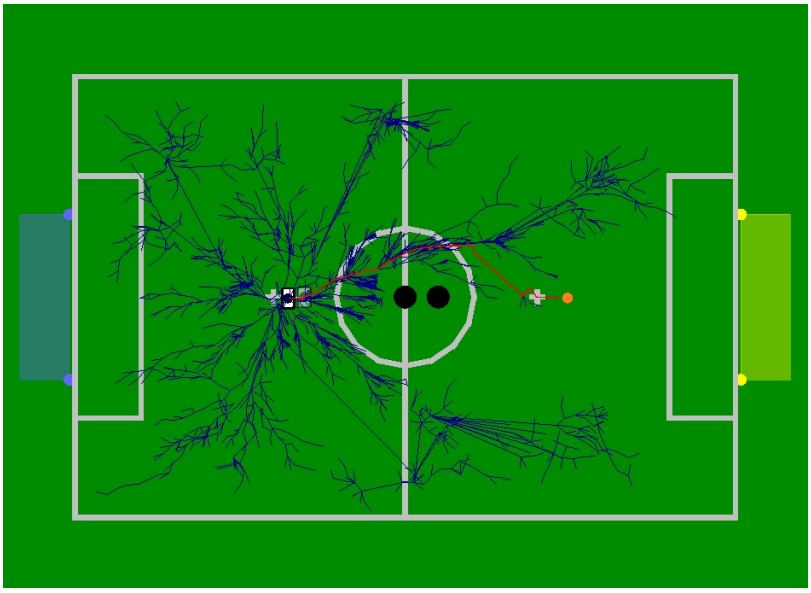
Zu Beginn des Algorithmus besteht der Baum nur aus einem initialen Knoten, dem Startpunkt. In der *getTarget*-Methode wird zunächst entschieden, in welche Richtung weiter expandiert werden soll. Mit Wahrscheinlichkeit p wird in Richtung des Ziels und mit Wahrscheinlichkeit $1 - p$ zu einem Zufallspunkt expandiert. Hierzu wird der aktuelle Baum mit Hilfe der *getNearest*-Methode nach dem Knoten, der dem neuen Punkt am nächsten liegt durchsucht, um von dort mit der *extendTo*-Methode zum neuen Punkt zu expandieren. Normalerweise wird die Distanz, die der neu berechnete Punkt vom aktuellen entfernt sein kann, durch die euklidische Distanz oder einen Zeitfaktor begrenzt [BV02]. Diese Begrenzungen der *extendTo*-Methode werden als „physical constraint" bezeichnet. Je kleiner diese, zuvor beschriebene, Distanz ist, desto länger braucht der Algorithmus um einen Pfad zu berechnen; allerdings wird dieser dadurch genauer und die Kurven weicher. Während dieses Vorgangs wird ebenfalls überprüft, ob der neue Punkt oder die Verbindungsstrecke zu ihm auf ein Hindernis trifft, es also zu einer Kollision kommt. In diesem Fall würde der neue Punkt verworfen und nicht in den Baum eingehängt. In die *extendTo*-Methode können außerdem Begrenzungen des Roboters, zum Beispiel bei der Beschleunigung, als „physical constraint" einfließen [BV02]. Dieser Vorgang wird so lange wiederholt, bis ein Knoten oder Blatt des Baumes im Zielbereich liegt. Ein RRT Codebeispiel ist in Abbildung 6.6 zu finden.

6.1.2.1 *Performance-Verbesserungen*

Eine Verbesserung der Performance wurde bereits angesprochen und liegt in der Verwendung eines *k-d-tree* als Datenstruktur [BV02]. Da bei einer Iteration im RRT der bereits bekannte Punkt mit der minimalen Distanz zum neuen Punkt gefunden werden muss, müssten ohne den *k-d-tree* alle bekannten Punkte durchlaufen werden. Eine weitere Verbesserung entsteht, wenn die Richtung, in die expandiert wird, geschickter gewählt wird. Hierzu kann ein Cache für Wegpunkte eingebaut werden [BV02]. Dieser dient dazu, dass sobald ein neuer Pfad berechnet wird, mit einer Wahrscheinlichkeit q (mit $(0 < q < 1 - p)$) wieder in Richtung der alten Wegpunkte expandiert wird, weil diese unter Umständen noch immer einen guten Pfad liefern. Ein weiterer Vorteil dieses Caches liegt in den signifikant weniger springenden Pfaden (vgl. Abb. 6.4), weil mit erhöhter Wahrscheinlichkeit ein Pfad ähnlich dem Vorherigen abgelaufen wird. Das heißt: Verläuft der Pfad rechts von einem Hindernis, wird der nächste Pfad mit erhöhter Wahrscheinlichkeit wieder rechts vom Hindernis verlaufen (vgl. Abb. 6.5).

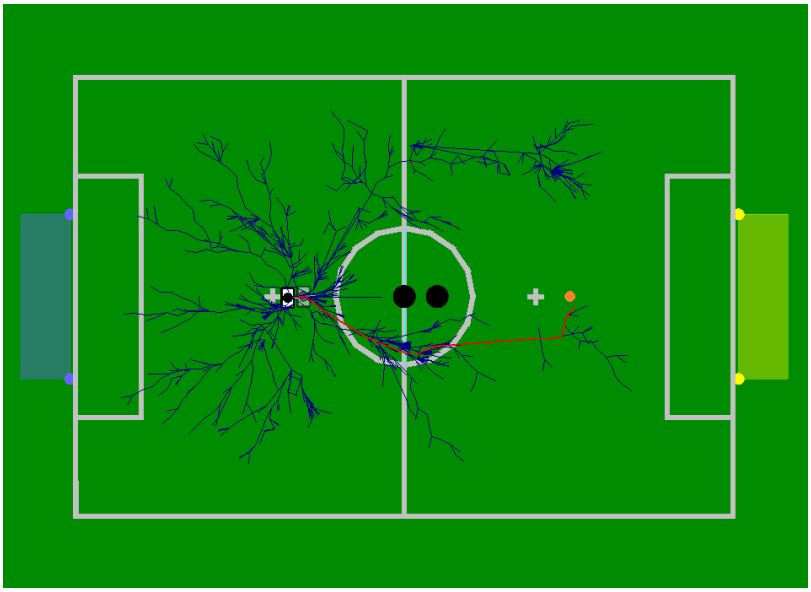


(a) Erster Durchlauf des RRT

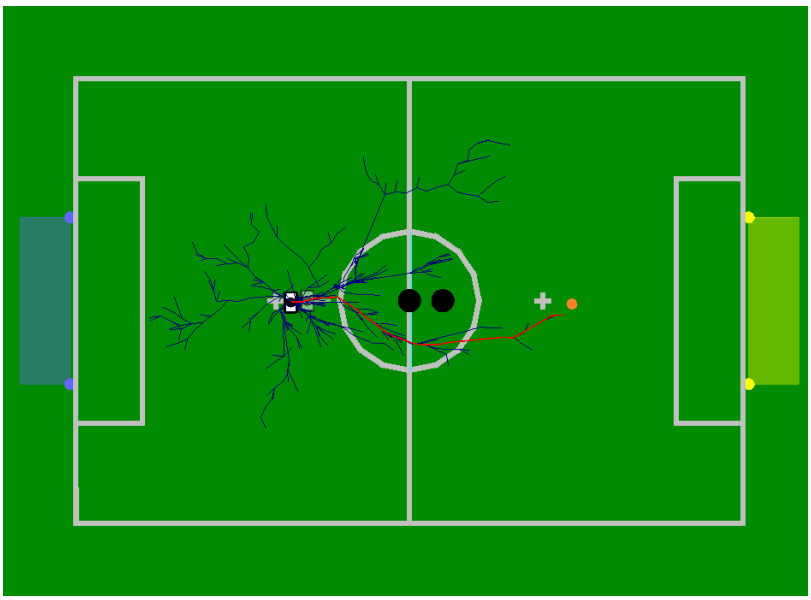


(b) Zweiter Durchlauf des RRT

Abbildung 6.4: RRT-Durchlauf ohne Cache für Wegpunkte;
Das Ziel des Roboters (weiß) ist es den Ball (orange) zu erreichen. Dabei muss er den Hindernissen (schwarz) ausweichen. Es fällt auf, dass der resultierende Pfad (rot) im ersten Durchlauf unterhalb und im zweiten Durchlauf oberhalb der Hindernisse verläuft (springender Pfad). Außerdem besitzt der Baum (blau) in beiden Durchläufen ungefähr gleich viele Knoten. Das bedeutet, dass beide Planungen ungefähr die gleiche Anzahl an Iterationen gebraucht haben (keine Zeitersparnis). Es gibt also *keinen Geschwindigkeitsvorteil* in der Pfadplanung im zweiten Durchlauf.



(a) Erster Durchlauf des RRT



(b) Zweiter Durchlauf des RRT

Abbildung 6.5: RRT-Durchlauf mit Cache für Wegpunkte;
Das Ziel des Roboters (weiß) ist es den Ball (orange) zu erreichen. Dabei muss er den Hindernissen (schwarz) ausweichen. Es fällt auf, dass der resultierende Pfad (rot) in beiden Durchläufen unterhalb der Hindernisse verläuft (kein springender Pfad). Außerdem besitzt der Baum (blau) im zweiten Durchlauf deutlich weniger Knoten als beim ersten Durchlauf. Das bedeutet, dass die Planung im zweiten Durchlauf deutlich weniger Iterationen gebraucht hat (Zeitersparnis). Es gibt also *einen Geschwindigkeitsvorteil* in der Pfadplanung im zweiten Durchlauf.

6.1.3 *Einbau des RRT*

Um einer neuen Implementierung des RRT und damit einem erhöhtem Zeitaufwand entgegen zu wirken, wird auf eine in C implementierte Bibliothek zurückgegriffen. Diese ist als Source-Code erhältlich und implementiert zusätzlich den *k-d-tree*. Implementiert wurde der Algorithmus von *Sertac Karaman* beim *Department of Electrical Engineering and Computer Science* am *Massachusetts Institute of Technology*. Ein weiterer Vorteil der Implementierung ist, dass sie, dadurch, dass es sich um einen Open-Source-Code handelt, sehr leicht in das schon bestehende Framework einbinden lässt. Die Entwicklung der RRT-Pfadplanung lief parallel zur Weiterentwicklung der bereits wettkampferprobten Pfadplanung. Es wurde ein neues Modul angelegt, um den Pfad mit dem RRT zu planen. Dadurch kann durch einfaches Austauschen der Module gewählt werden, wie der Pfad geplant werden soll. Nach Einarbeitung in den Code von *Sertac Karaman* konnten die Parameter, die die RRT-Pfadplanung spezifizieren, so verändert werden, dass der Algorithmus Pfade auf dem Feld ausgab. Diese Parameter sind in der *rrt.cfg* gespeichert und werden beim Start aus ihr geladen, um eine Veränderung der Parameter ohne erneutes Kompilieren zu ermöglichen. Da die Berechnung teilweise zu viel Zeit benötigte, wurden weitere Parameter eingeführt, um die Pfadplanung besser steuern zu können.

MINITERATIONCOUNTER stellt sicher, dass eine Mindestanzahl an Iterationen vom RRT ausgeführt wird, bevor ein Pfad übernommen wird. Dadurch wird eine gewisse Güte der Pfade gewährleistet, da der RRT zufällig arbeitet.

MAXITERATIONCOUNTER startet die Pfadplanung nach einer bestimmten Anzahl von Iterationen neu. Beim Neustart wird der aktuelle Start- und Zielpunkt neu gesetzt. Dadurch wird verhindert, dass zu lange ein alter Pfad abgelaufen wird, obwohl sich das Ziel eventuell schon verändert hat.

MINVIEWDISTANCE gibt die minimale Distanz an, die der NAO von einem Hindernis entfernt sein muss, um dieses erkennen zu können. Unterschreitet ein der Pfadplanung bereits bekanntes Hindernis diese Grenze, wird es von der Pfadplanung automatisch weiter berücksichtigt, selbst wenn der NAO es nicht weiter als solches erkennt.

Ein weiteres Problem stellen springende Pfade dar. Falls sich auf der geraden Linie zwischen Start und Ziel ein Hindernis befindet, springt der Pfad. Das bedeutet, dass der berechnete Pfad links beziehungsweise rechts um das Hindernis herum führt, während der anschließend berechnete Pfad auf der jeweils anderen Seite um das Hindernis herum führen kann (vgl. Abb. 6.4). Das hat zur Folge, dass

der NAO in den gegnerischen Roboter hinein läuft, anstatt an ihm vorbei zu laufen. Abhilfe soll an dieser Stelle der zuvor erwähnte Wegpunkte-Cache schaffen (vgl. Abschnitt 6.1.2.1). Hierzu wird der Code von *Sertac Karaman* um diesen Cache und eine Wahrscheinlichkeit, mit der in Richtung eines alten Wegpunktes expandiert wird, erweitert. Nach Einbau des Caches fällt auf, dass die Pfade deutlich robuster sind (weniger bis gar nicht mehr springen) und außerdem neue Pfade bei gleich bis ähnlich bleibendem Ziel deutlich schneller generiert werden. Umgekehrt muss darauf geachtet werden, den Cache bei sich veränderndem Ziel zu löschen.

In den entstehenden Pfaden wird die Rotation allerdings nicht mit geplant. Es entstehen *2D*-Pfade, die vom aktuellen *Pattern Generator* umgesetzt werden können; dazu mehr im Abschnitt Konvertierung. Die Rotation der Punkte der *2D*-Pfade wird dabei darüber berechnet, wie die Punkte zueinander stehen, sodass ein Punkt stets auf seinen Nachfolger zeigt, was Probleme mit sich bringt. Steht der Roboter mit dem Rücken zum nächsten Wegpunkt, muss ein Zwischenschritt eingefügt werden, in dem der NAO sich auf der Stelle dreht ehe er zum nächsten Wegpunkt läuft, an dem er sich gegebenenfalls wieder drehen muss. Der Pfad muss entsprechend nachbearbeitet werden. Diese Nachbearbeitung kann vermieden werden, indem die Rotation als dritte Dimension mit geplant wird. Bei dieser Veränderung in der Pfadplanung des RRT muss darauf geachtet werden, dass die Hindernisse die dritte Dimension voll ausfüllen, weil der Pfad sonst keine Kollision feststellt, falls das Hindernis und der Punkt auf dem Pfad über verschiedene Rotationen verfügen. Außerdem ist es notwendig die Funktion, die die Kosten berechnet, um zum neuen Punkt zu expandieren, dahingehend abzuändern, dass sie den Pfad dazu veranlasst, den „physical constraints“ des NAO entgegenzukommen.

XPENALTY bestraft rückwärts laufen,

YPENALTY bestraft seitwärts laufen,

ROTPENALTY bestraft starkes rotieren.

Dabei wird davon ausgegangen, dass der Roboter die höchste Geschwindigkeit bei einem Lauf geradeaus erreicht. Es zeigt sich, dass die Strafen den Pfad dazu veranlassen, die Rotation zwischen den Punkten deutlich weniger stark schwanken zu lassen. Die Pfade, die von dieser umgebauten RRT-Pfadplanung generiert werden, sind nicht so weich wie die Pfade, die vom Potentialfeld generiert werden. Sie besitzen stärkere Rotationen und umlaufen Hindernisse teilweise großräumiger als dies notwendig wäre, was zu einem insgesamt längeren Pfad führt. Bei näherer Betrachtung der Pfade fällt auf, dass der Abstand, den ein neuer Punkt von dem zu ihm nächsten Punkt des Baumes haben kann, in dem verwendeten Code nicht nach oben

begrenzt wird. Deshalb wird der Algorithmus um eine solche Maximaldistanz erweitert, die ebenfalls über die Config-Datei gesetzt werden kann. So werden Äste, um die der Baum pro Iteration wächst, falls sie länger als die angegebene Maximaldistanz wären, auf diese gestutzt. Dadurch dauert die Berechnung der Pfade länger, was die RRT-Pfadplanung bei dem gewählten Parametersatz leider auf Grund der zu hohen Rechenzeit unbrauchbar macht.

6.1.4 *Konvertierung*

Aktuell wird zwischen der *Pfadplanung* und dem *Pattern Generator*, der Fußstapfen entlang des Pfades setzt, ein weiteres Modul, der *Request Translator* ausgeführt. Dieser berechnet aus den Punkten des Pfades, die in Feldkoordinaten angegeben sind, Geschwindigkeiten mit denen der *Pattern Generator* arbeitet. Zukünftig soll ein neuer *Pattern Generator* direkt mit dem Pfad als Eingabe arbeiten. Deshalb muss der Pfad in Roboterkoordinaten konvertiert werden. Da der Pfad, der von der RRT-Pfadplanung ausgegeben wird, nicht kontinuierlich ist, sondern nur aus einzelnen Punkten Eckpunkten besteht, sollen diese einem *B-Spline* [PBP02] als Eingabe dienen. Dieser soll den kontinuierlichen Pfad künftig darstellen. Dazu wurde eine neue *B-Spline*-Klasse implementiert. Sie arbeitet auf dreidimensionalen Vektoren. Ein *B-Spline* berechnet, in Abhängigkeit eines Grades, eine Basisfunktion. Mit Hilfe dieser Basisfunktion werden die Kontrollpunkte, in diesem Fall die Eckpunkte des Pfades, projiziert. Der Grad bestimmt, wie viele Stützpunkte die Projektion des aktuellen Punktes beeinflussen. Beim Projizieren gilt, dass die entstehende Projektion der Kontrollpunkte stets in der konvexen Hülle aller Kontrollpunkte liegt. Der *B-Spline* erlaubt dem *Pattern Generator* den Grad als Eingabe für die Basisfunktion frei zu wählen, während die *Pfadplanung* bestimmen kann welche Kontrollpunkte durchlaufen (auf sich selbst projiziert) werden müssen, um beispielsweise Hindernissen auszuweichen. Auf diesen projizierten Punkten arbeitet der neue *Pattern Generator*.

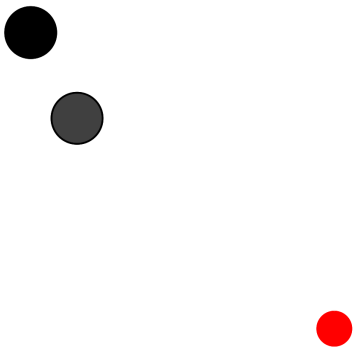
```
Tree rrt(Region start, Region goal, vector<Region> obstacles) {
  Tree rrt_tree = new Tree(start);

  while (dist(getNearest(rrt_tree, goal).ownRegion, goal) <
      threshold)
  {
    Region target = getTarget(goal);
    Node nearest = getNearest(rrt_tree, target);
    extendTo(nearest, target, obstacles);
  }
  return rrt_tree;
}


Node getNearest(Tree current, Region towards) {
  Node nearest = current.root;

  foreach (Node n in current)
    if (dist(n.ownRegion, towards) < dist(nearest.ownRegion,
        towards))
      nearest = n;

  return nearest;
}


Region getTarget(Region goal) {
  double p = rand(0,1);
  if (p < goalProb)
    return goal;
  else
    return randomRegion();
}


void extendTo(Tree current, Node nearestNode, Region to, vector<
    Region> obstacles) {
  // if way between two regions is too far, clip it
  Region destination = clipWay(nearestNode.ownRegion, to);

  if !(isInCollision(nearestNode.ownRegion, destination,
      obstacles))
    current.insert(nearestNode, destination);
}
```

Abbildung 6.6: RRT Codebeispiel

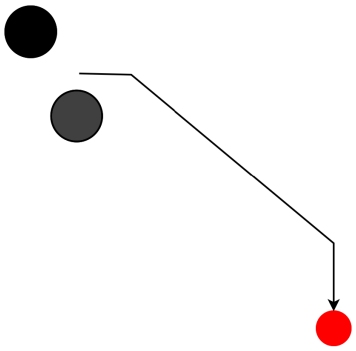
6.2 PATTERN GENERATOR

Der *Pattern Generator* hat die Aufgabe die Positionen der einzelnen Fußstapfen des Roboters zu berechnen. Bislang erfolgt hier eine Umwandlung des Pfades in Geschwindigkeiten, welche dann im nächsten Schritt in konkrete Fußstapfen umgewandelt werden. Fortan werden gegebene Pfade direkt in Fußstapfen umgewandelt. Durch zusammenfassen dieser beiden Schritte wird eine direktere und zukünftig sauberere Ansteuerung erhofft.

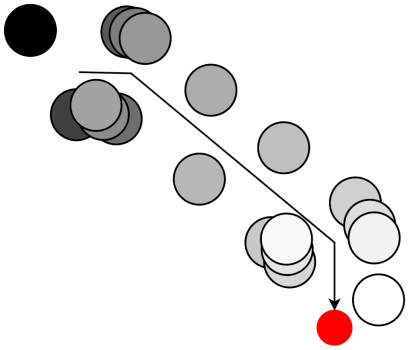
Weiterhin ändert sich auch die Eingabe. In der alten Version sind Wegpunkte gegeben, welche einzeln angesteuert werden. Jetzt wird es durch den zuvor vorgestellten RRT zu einer Vielzahl von Wegpunkten kommen. Diese sollen nicht jeweils einzeln angesteuert werden, sonst wäre ein effektiver Lauf unmöglich. Vielmehr soll ein gewisse Toleranz, was das Erreichen von Wegpunkten angeht, herrschen. Schließlich dienen diese lediglich als Hilfspunkte und müssen nicht exakt abgelaufen werden. Dies erfolgt durch das Interpolieren mittels des *B-Splines*. Hierdurch wird eine effizientere, durchgehende Laufbewegung angestrebt, die sogar direkt die abschließende Ausrichtung des Roboters beinhaltet. Zudem hat es per Konstruktion weniger Effizienzeinbußen, da ein gleichmäßiger durchgehender Lauf ohne unnötige Start- und Stoppbewegungen angestrebt wird (Abb.: 6.7).



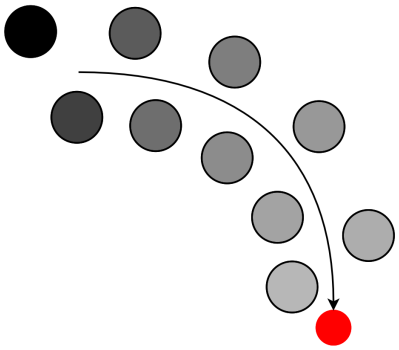
(a) Spielsituation: Darstellung der beiden Fußpositionen, sowie des roten Balls



(b) Normales Vorgehen: plane Wegpunkte



(c) Normal Vorgehen: laufe diese Wegpuntke ab



(d) Spline: laufe Spline ab

Abbildung 6.7: Darstellung der Pfadplanung: Aus der bisherigen Pfadplanung kamen Wegpunkte, welche abgelaufen wurden. Dies führt hier an den beiden Wegpunkten zu einem auf der Stelle drehen. Erst dann wird weitergelaufen. Abschließend das neue Vorgehen, durch Ablaufen eines Splines.

Zu erst wird ein Überblick gegeben, mit welcher Ansteuerung ein Roboter an sich läuft, bevor später die dahinter steckende Terminologie der Ansteuerungsvariablen der hier verwendeten *Walking-Engine* erläutert wird, um anschließend auf das von der PG entwickelte Konzept einzugehen.

6.2.1 *Laufvorgang*

Beim Laufvorgang wird eine Folge von Schritten ausgeführt. Jeder dieser Schritte wird hier nochmals in zwei Phasen unterteilt. Es wird zwischen einer *SingleSupport*- und einer *DoubleSupport*-Phase unterschieden. Die Bezeichnung der Phase bezieht sich jeweils auf die Anzahl der Füße durch welche der Körper gestützt (engl. *supported*) wird.

Initial befindet sich der Roboter in einer *unlimitedDoubleSupport*-Phase, steht also mit beiden Füßen auf dem Boden und bewegt sich

nicht. Soll der Roboter nun loslaufen, so wird in die *SingleSupport*-Phase gewechselt. In dieser bleibt ein Fuß am Boden, während der andere beginnt die Bewegung auszuführen. Im Anschluss daran folgt ein Übergang in die *DoubleSupport*-Phase, welche den sich bewegenden Fuß auf seine Endpostion setzt, so dass der Roboter wieder mit beiden Füßen auf dem Boden steht. Es wird zusätzlich noch unterschieden welches Bein in der Phase aktiv ist, so dass man auf zwei mal zwei Phasen kommt. Als Übergang in einen neuen Schritt wird jeweils eine *DoubleSupport*-Phase verwendet.

Damit aus den einzelnen Schritten auch wirklich eine Reihe an Schritten entsteht, die zusammen einen Lauf ergeben, müssen die Pfadstücke sauber aneinander gehangen werden. Schließlich soll hier eine gleichmäßige Bewegungsabfolge entstehen ohne unangepassten Übergänge, welche die Stabilität des Roboters gefährdet könnten. Die bereits geplante Bewegungsabfolge nennt sich *Preview* und die Endposition dieser Bewegungsabfolge wird in *poseAfterStepPreview* festgehalten. Bis zu dieser Position ist gerade eine Bewegung im Gange und von dieser Position aus wird der *B-Spline* neu berechnet.

Weiterhin kommt eine ständige Aktualisierung der *poseAfterStepPreview* hinzu, welche als Startpunkt für die nächsten Wegpunkte fungiert.

6.2.2 *Konzept*

Die Projektgruppe entwickelt ein Konzept eines neuen *PattenGenerators* der im folgenden beschrieben wird. Das Modul *Pattern Generator* wird von außen durch seine *update*-Methode angestoßen und es wird dieser eine Referenz auf *FootSteps* übergeben. Wenn das Modul aktiv ist und die Variable *framesTillNextStep* den Wert Null angenommen hat, werden durch eine Routine neue Schritte generiert. Ein Schritt besteht aus zwei Fußpositionen, einen für den rechten und einen für den linken Fuß.

Zuvor werden die von der Pfadplanung bereitgestellten Wegpunkte genutzt. Diese werden durch einen *B-Spline* interpoliert. Die daraus entstehende *B-Spline*-Kurve dient dieser Klasse als Basis der Fußstapfen-Erstellung. Um den aktuellen Schritt nicht mittendrin zu unterbrechen, werden von der *poseAfterStepPreview* an Fußstapfen entlang dieser Kurve gesetzt. Dazu sucht die Methode *findPointOnPath* nach einer erreichbaren Position in der Nähe des Roboters. Dies geschieht mittels einer binären Suche auf dem *B-Spline*. Dabei wird versucht einen Punkt maximaler Entfernung zu finden, welcher jedoch eine stabile Bewegung ermöglicht. Geprüft wird die vorgeschlagene Position per *Constraints* der Methode *poseCanBeReached*. Werden diese verletzt, so wird die Größe des Teilstück auf dem gesucht wird verringert. Hält die Position diesen Test stand, wird das Teilstück erweitert. Dies wird solange durchgeführt bis sich die Größe es Teilintervalles,

im Vergleich zur vorherigen Größe, nur noch um einen sehr kleinen Wert verändert. Die dazugehörigen *Constraints* sind frei wählbar und werden durch ein Parameter-Tuning bestimmt. Die ersten Grundannahmen sind so getroffen, dass die nächste Roboterposition z.B. nur 2 cm in y-Achsenrichtung von der zuletzt berechneten Roboterposition entfernt liegen darf.

Nun existiert eine zukünftige Roboter-Position auf dem *B-Spline*, an die später sein Fuß platziert wird. Um die Platzierung des Fußes kümmert sich wiederum die Methode *splineToFootPosition*. Diese setzt den Fuß entsprechend neben der Position auf der Kurve. Dabei wird eine Orthogonale an dem Kurvenpunkt berechnet, auf der der jeweilige Fuß um den Fußabstand in y-Achsenrichtung (*theWalkingEngineParams.footYDistance*) verschoben wird. Die oben genannten Methoden *findPointOnPath* und *splineToFootPosition* werden aus der Methode *findNextStep* aufgerufen, welche auch die Koordinierung der Übergänge zwischen den einzelnen Bewegungsphasen regelt. Dazu wird in der Variable *currentWalkingPhase* die aktuell befindliche Laufphase gespeichert.

Hinzu kommen alle Parameter die die Fußansteuerung betreffen. Diese werden in einer Datenstruktur namens *Footposition* abgelegt . Somit beinhaltet die Datenstruktur alle notwendigen Informationen um diese in *Footsteps* zu einer Bewegungsfolge zusammenzufassen. Die Klasse *Footposition* speichert in *footPos*, jeweils für den linken und rechten Fuß, einen dreidimensionalen Vektor. Dieser beschreibt, aus Sicht des Oberkörpers, die anzusteuernde Position der Füße. Das Koordinatensystem entspricht in der x-Richtung einem Vorwärts/Rückwärtsbewegung, während die y-Richtung Links/Rechtsbewegung angibt und z die Höhe des Fußes. Weiterhin wird durch *onFloor* angegeben, ob der betreffende Fuß Bodenkontakt hat. Es erfolgt noch ein Übergabe der Bewegungsrichtung in *direction*. Damit die Ansteuerung weiß zu welcher Phase die *Footposition* gehört, wird die *currentWalkingPhase* in *phase* übertragen.

6.2.3 *Implementierung-Ausblick*

Das zuvor vorgestellte Konzept wird nun in das Framework des *Pattern Generators* eingebunden. Die *update*-Methode der Klasse wird in jedem Frame einmal aufgerufen und dient dem Zweck die vorher berechneten Roboterschritte zusammenzufassen. Hierbei müssen drei Fälle betrachtet werden. Der Beginn einer Laufbewegung, das Berechnen neuer Schritte und das eine Schrittphase noch nicht abgeschlossen ist. Im ersten Fall müssen *theControllerParams.N*+1 viele Schritte erstellt und dem *FootSteps*-Array hinzugefügt werden. Im zweiten Fall wird die Methode *findNextStep* aufgerufen und es wird ein neuer Schritt berechnet. Im letzten Fall wird der zuvor durch *findNextStep* berechnete Schritt solange der *FootSteps*-Referenz hinzugefügt,

bis die Schrittbewegung abgeschlossen ist. Zu Testzwecken werden mehrere Splinekurven erzeugt und die berechneten Fußpositionen, die dem *FootSteps*-Array hinzugefügt werden, in Textform ausgegeben. Die Ausgabe bestätigt die Erzeugung von korrekt platzierten Fußstapfen, in der richtigen Reihenfolge, in der zugehörigen *Walkingphase*. Bei dem Aufruf eines Laufbefehls bleibt der Roboter auf der Stelle stehen und beginnt sich seitlich aufzuschwingen. Dieses ungewollte Verhalten endet im Umfallen des Roboters. Dies geschieht, obwohl die übergebenen Fußpositionen überwacht und korrekt sind. Aufgrund des Zeitmangels konnte die Schnittstelle zwischen dem Konzept und dem vorhandenen Framework nicht richtig verbunden werden, sodass dieses Problem weiterhin besteht. Die Vorstellungen der PG sind, dass die bisherige Arbeit als Basis genutzt wird, um einen neuen *Pattern Generator* zu implementieren.

7 VERHALTEN

Um Veränderungen in anderen Modulen bei Verhaltensentscheidungen der NAOs zu berücksichtigen, ist auch die Anpassung und Verbesserung des Verhaltens in der Projektgruppe thematisiert und schrittweise umgesetzt worden. Die Implementierung wurde dabei unter anderem in der Extensible Agent Behavior Specification Language (XABSL) vorgenommen, welche schon seit längerem für die Spezifizierung des Roboterverhaltens bei den *NaoDevils* eingesetzt wird.

So wurde zum einen ein Verfahren entwickelt, mit welchem ein NAO eine starke Laufinstabilität, die voraussichtlich zu einem Sturz führen würde, selbstständig erkennen und abmildern kann. Dies ist insbesondere sinnvoll, um ein Aufschaukeln des Roboters beim Laufen zu verhindern und somit Stürze zu vermeiden, welche gerade in Wettkämpfen wertvolle Zeit kosten würden. Außerdem unterstützt dies Laufoptimierungsansätze wie die in Kapitel 4 beschriebenen, indem ungünstige Laufverhalten, die durch ein generiertes Parameterset erzeugt werden, beendet werden, bevor es zu Beschädigungen am NAO kommt.

Des Weiteren wurde die bestehende Entscheidungsfindung, welcher NAO sich am sinnvollsten, das heißt am schnellsten, in eine günstige Ballschussposition begeben kann, überarbeitet und um wichtige Faktoren ergänzt, um das bisherige Verfahren zu verbessern.

Außerdem wurde im Rahmen der Projektgruppe eine Auswahlmöglichkeit geschaffen, um verschiedene Verhaltensmuster direkt über den *Chestbutton* des NAOs zu starten, insbesondere für das Elfmeterschießen nach einem regulären Spiel, aber auch für allgemeine Demonstrationszwecke.

7.1 AUTOMATISCHES ABSTOPPEN BEI LAUFINSTABILITÄT

Ziel ist eine eigenständige Erkennung von Laufinstabilitäten eines NAOs, welche so stark sind, dass sie ihn zu Fall bringen würden.

Solche Instabilitäten treten beispielsweise dann auf, wenn ein NAO mit dem Fuß an einem Hindernis oder einfach nur dem Boden hängen bleibt. Dabei kommt es immer wieder vor, dass sich ein Roboter aus dieser Instabilität soweit aufschaukelt, dass er stürzt. Anschließend muss er sich erst mühevoll wieder aufrichten und lokalisieren, bevor er weiterspielen kann. Dies kostet wertvolle Zeit, welche es gerade in Wettkämpfen zu minimieren bzw. bestenfalls zu vermeiden gilt. Daher soll der Roboter nach dem Erkennen einer Instabilität automatisch abstoppen, bis er sich wieder stabilisiert hat, um dann aus

dieser stabilisierten Lage wieder seinen normalen Betrieb aufzunehmen.

Um dies umzusetzen, sind zunächst die Möglichkeiten zu evaluieren, an welchen Stellen des Frameworks eine solche Funktion sinnvoll eingebaut werden könnte. Für die Implementierung dieses Features kommen dabei grundsätzlich all diejenigen Module in Frage, die innerhalb des Frameworks einen MotionRequest (vgl. *Representations/MotionControl/MotionRequest.h*) verarbeiten können. Im verwendeten Framework sind dies zum Beispiel der RequestTranslator, der MotionSelector oder der MotionCombinator.

Darüber hinaus wäre eine Einbindung prinzipiell direkt in der Laufsteuerung denkbar, welche bei den NAOs derzeit von der *DortmundWalkingEngine* übernommen wird. Aufgrund des Designkonzeptes des Frameworks sowie der sich daraus ergebenden Codestruktur ist von letzterem abzusehen. Dies gilt insbesondere im Hinblick auf die gewünschte Trennung der Komponenten, um Korrekturen oder Wechsel zu anderen Verfahren einfach zu halten (Modularität).

Folglich ist die erstgenannte Variante zu wählen. Basierend auf bereits vorhandenen Funktionen bzw. deren Aufteilung erfolgt die Implementierung im MotionSelector (*Modules/MotionControl/MotionSelector.cpp*).

Um die Stabilität des NAOs einschätzen zu können, werden zunächst die Sensorwerte der in den NAOs verbauten Sensoren benötigt. Bereitgestellt werden diese über die Basisschnittstelle NaoQi, welche vom Hersteller der NAO-Roboter, Aldebaran Robotics, mitgeliefert wird. Auf Grund der engen Verzahnung von NaoQi und dem eingesetzten Framework sind diese Werte anschließend über die Repräsentation der Sensordaten, welche in der Datei *Representations/Infrastructure/SensorData.h* definiert ist, direkt im Framework verfügbar.

Nach Prüfung der verschiedenen Sensoren und Werte, wie etwa des eingebauten Gyroskops (Werte: *gyroX*, *gyroY*), der Beschleunigungssensoren (Werte: *accX*, *accY*) oder auch der Neigungswinkel (Werte: *angleX*, *angleY*), lässt sich feststellen, dass die Verwendung der Neigungswinkel die insgesamt stabilste Entscheidungsfindung ermöglicht. Dies lässt sich insbesondere dadurch erklären, dass für deren Ermittlung die Werte verschiedener physikalischer Sensoren, welche natürlichen Schwankungen und Messfehlern unterworfen sind, zunächst gefiltert und dann in einer mathematischen Berechnung aus diesen verschiedenen Sensorwerten zusammengefasst werden.

Für den Verlauf der Werte während des Laufens ist nun zu überprüfen, ob folgende Bedingungen eingehalten werden:

- angleX *(Neigung in X-Richtung)*: Der Betrag dieses Wertes sollte für einen stabilen Lauf möglichst klein bleiben. Gleichmäßige, geringe Schwankungen sind dabei tolerierbar.

- angleY *(Neigung in Y-Richtung)*: Je höher dieser Wert, desto eher schaukelt sich der NAO beim Lauf auf. Um dieses Aufschaukeln zu verhindern, bevor es zu einem Sturz kommt, sollte dieser Wert ebenfalls eine möglichst geringe Abweichung vom Nullpunkt haben.

Aus einem einzelnen Wert außerhalb des Grenzbereichs lässt sich allerdings nicht sicher ableiten, dass eine Instabilität vorliegt, da es sich dabei möglicherweise auch um einen Messfehler handeln kann. Daher wird mit den letzten Werten von *angleX* und *angleY* jeweils ein Ringbuffer befüllt und der dazugehörige Mittelwert ermittelt. Sobald nun einer dieser Mittelwerte die eingestellten zulässigen Grenzwerte überschreitet, wird von einer Instabilität ausgegangen.

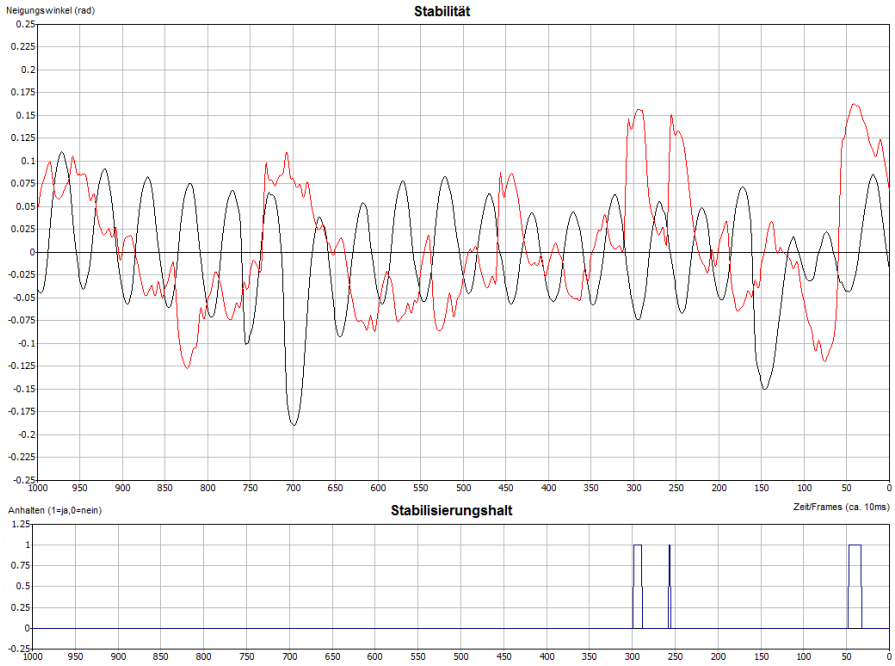


Abbildung 7.1: Plot von einem NAO, der eine kurzfristige Instabilität (bei ca. 300) selbstständig ausgleicht.
(Legende: Neigung des Roboters in x-Richtung (schwarz) und y-Richtung (rot) sowie binäre Entscheidung, ob der Roboter einen Stabilisierungshalt einlegt (unten, blau))

Sobald eine Instabilität festgestellt wurde, stoppt der NAO seine Laufbewegung ab, bis die Schwankungen wieder unterhalb der Grenzwerte sind und fährt anschließend mit seinem aktuellen Verhalten fort. Dabei können die Grenzwerte für die Schwankungen in x- und y-Richtung einerseits jeweils auf feste Werte eingestellt werden, um in verschiedenen Anwendungsfällen einen Sturz zu vermeiden. Alternativ können die Werte auch so konfiguriert werden, dass sie die besonderen Umstände eines Laufoptimierungsverfahrens optimal berücksichtigen. In jedem Fall müssen die Werte je nach verwendetem

Robotermodell oder Simulator entsprechend angepasst werden, wie sich beim Wechsel auf die neue NAO Hardware-Generation zeigte.

Nach mehreren Testläufen, sowohl im Simulator als auch auf den realen Robotern, lässt sich feststellen, dass das implementierte Verfahren die gewünschten Ziele erreicht. Dabei sind folgende Ergebnisse zu beobachten:

Zum einen werden kurze, spontan auftretende Instabilitäten umgehend abgemildert. Dies lässt sich beispielsweise in Abb. 7.1 gut erkennen: Beim Überschreiten des Grenzwertes für die y-Abweichung stoppt der NAO für wenige Augenblicke, bis er sich wieder stabilisiert hat, der Neigungswinkel in y-Richtung also im Mittel wieder unter dem eingestellten Grenzwert liegt. Die Bewegung des Roboters in x-Richtung behält dabei ihr leichtes, jedoch regelmäßiges Schwingungsverhalten nahezu unverändert bei, weshalb der Roboter sich nach dem Stabilisieren ohne großen Zeitverlust wieder in Bewegung setzt.

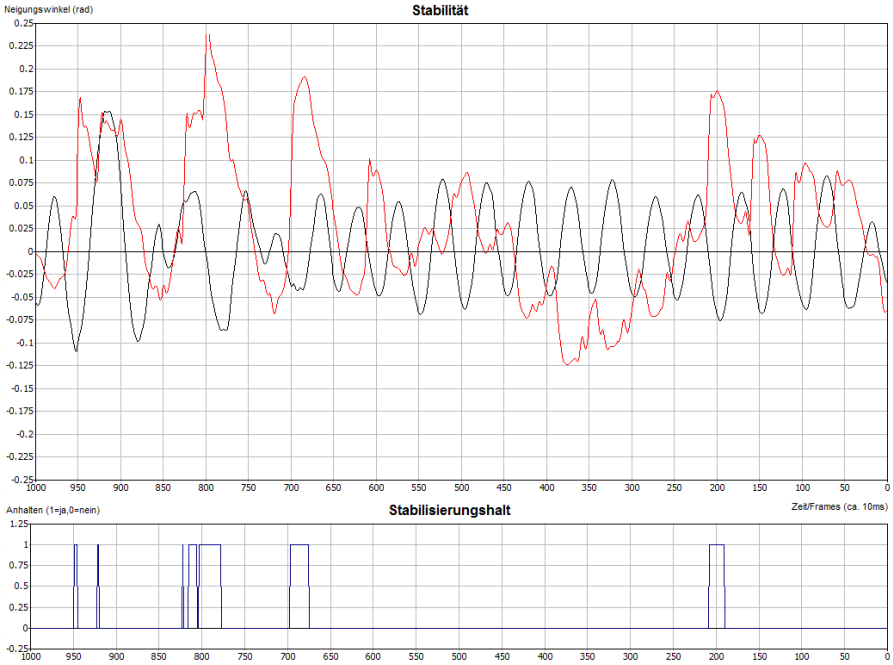


Abbildung 7.2: Plot von einem NAO, der sich aus einer längeren Instabilität selbstständig stabilisiert.
(Legende: Neigung des Roboters in x-Richtung (schwarz) und y-Richtung (rot) sowie binäre Entscheidung, ob der Roboter einen Stabilisierungshalt einlegt (unten, blau))

Zum anderen ist es mit diesem Verfahren möglich, auch größere bzw. länger anhaltende Instabilitäten auszugleichen. Dabei stoppt der NAO beim längerfristigen Überschreiten eines Grenzwerts so lange, bis dieser wieder unterschritten ist (vgl. Abb. 7.2). Dies wiederholt sich, bis ein dauerhaft stabiler Lauf vorliegt.

7.2 OPTIMIERTE BALLANNÄHERUNGSENTSCHEIDUNG

Während der GERMAN OPEN 2012 traten Schwachstellen im Verhalten der Roboter zu Tage. Eine der offensichtlichsten Schwachstellen der zu jenem Zeitpunkt eingesetzten Verhaltensimplementierung ist die Entscheidung, welcher NAO am günstigsten zum Ball positioniert ist und daher zum Ball gehen sollte.
Diese Entscheidung kommt bis dahin ausschließlich durch den Vergleich der Distanzen der einzelnen NAOs zum Ball (*estimated ballposition*) und deren jeweiliger Rotation zum Ziel zustande (Abb.: 7.3).

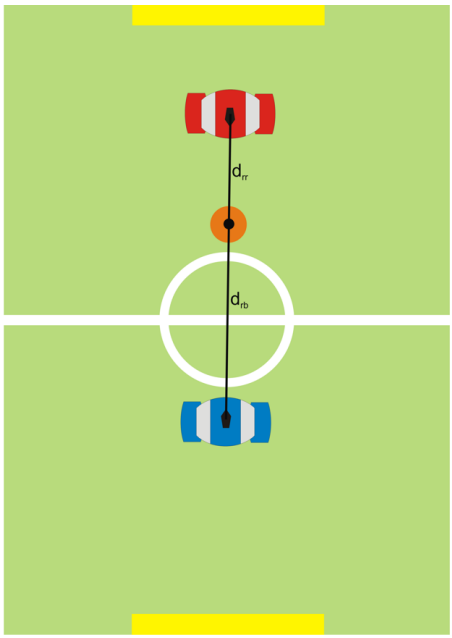


Abbildung 7.3: Bisherige Entscheidung zur Ballannäherung: Beide NAOs haben den gleichen Winkel zum Ziel (oberes Tor). Der rote NAO hat die kürzere Distanz und wird daher zum Ball gehen. Allerdings kostet das Umlaufen des Balles durch den roten NAO mehr Zeit, als der blaue NAO benötigen würde, um sich in Schussposition zu bringen.

Dies kann zur Folge haben, dass ein NAO, der in Torschussrichtung hinter dem Ball steht, als näher zum Ball angenommen wird, als ein NAO, der vor dem Ball steht, allerdings weiter entfernt ist. Da umfangreichere Rotationsbewegungen nötig sind, um den ersten NAO in eine geeignete Schussposition zu bringen, benötigt dieser allerdings deutlich mehr Zeit für die Annäherung an den Ball als der zweite NAO, welcher sich in direkter Schussrichtung zum Tor befindet.

Ein weiteres Problem ist die fehlende Berücksichtigung von Hindernissen bei der Ballannäherungsentscheidung. Auch diese können den Zeitbedarf der Annäherung eines NAOs an das Ziel stark beeinflussen. Derartige Fehlentscheidungen und suboptimalen Entscheidungen im Spiel gilt es zu vermeiden oder zumindest deutlich zu reduzieren.

Im Zuge der Korrekturen und Verbesserungen wird zudem eine stärkere und eindeutige Trennung zwischen der Verhaltensentscheidung und der Berechnung ihrer Eingaben vorgenommen, um die Modularität des Konzeptes zu erhöhen.
Dazu sollen die Berechnungen eines Teils dieser Verhaltensentscheidungen aus XABSL in neue Symbole ausgelagert werden, welche anschließend wiederum in XABSL weiterverarbeitet werden können. Diese Trennung erlaubt zudem die Option auf eine vereinfachte Ablösung von XABSL als Verhaltenssprache, falls dies in Zukunft gewünscht sein sollte. Die Berechnung der für diesen konkreten Fall nötigen einzelnen Teilrechnungen sowie der Entscheidung, ob ein Roboter der 'nächste' zum Ball ist, ist daher in den Kontext der taktischen Symbole (*TacticSymbols*) einzubetten.

Die Entscheidung, welcher NAO am günstigsten zum Ball steht, entscheidet sich aus dem Vergleich der neu implementierten Gewichtungsfunktion, die basierend auf vier Teilkomponenten die Komplexität des zurückzulegenden Weges in einem Gewichtswert ausgibt. Der NAO mit dem niedrigsten Gewichtswert ist folglich der am günstigsten positionierte, um zu einer optimalen Torschussposition zu gelangen.
Dabei gilt es zu beachten, dass, bedingt durch die Position des Balles und des NAOs auf dem Feld, ein normaler Schuss mit dem linken oder rechten Fuß oder auch ein Seitwärtsschuss auf das Tor am sinnvollsten sein kann. Um dies zu realisieren, wird die relative Position des Balles zum Tor sowie die relative Position des NAOs zum Ball berücksichtigt.

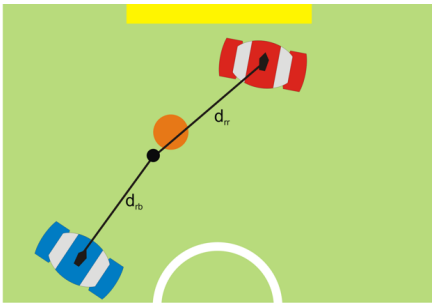
Die ideale Ballschussposition kann also für jeden NAO unterschiedlich sein und hat direkten Einfluss auf die erste Komponente der Gewichtsfunktion.

Die vier Komponenten im Überblick:

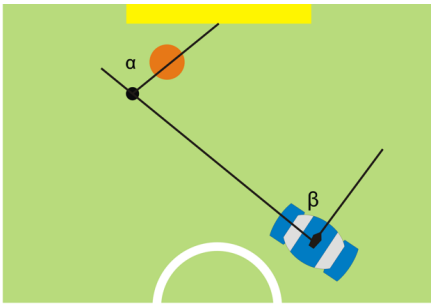
Komponente 1:
Die euklidische Distanz zwischen dem NAO und der optimalen Schussposition (Abb.: 7.4a).

Komponente 2:
Die notwendige Rotation des NAOs von der aktuellen Ausrichtung zur Ausrichtung auf den Zielpunkt (Abb.: 7.4b).

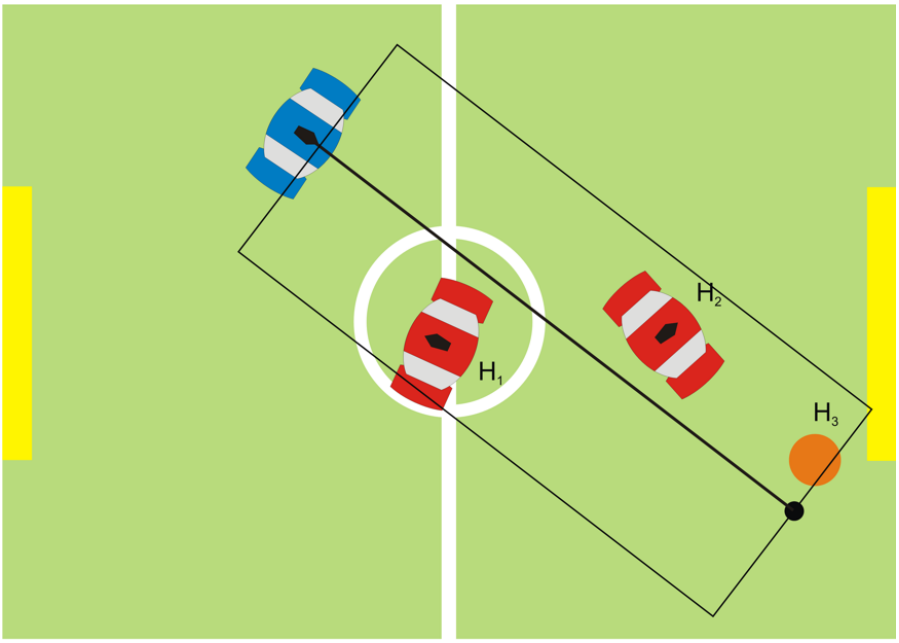
Komponente 3:
Die notwendige Rotation des NAOs am Zielpunkt, um aus der Laufrichtung in die Schussausrichtung zu gelangen (Abb.: 7.4b).



(a) Komponente 1



(b) Komponente 2 und 3



(c) Komponente 4

Abbildung 7.4: Aktuelle Entscheidung zur Ballannäherung

Komponente 4:
Berücksichtigung von Hindernissen innerhalb eines Korridors der gedachten direkten Linie zwischen der aktuellen Position eines NAOs und seiner Zielposition. Jedes bekannte Hindernis in diesem Korridor addiert zusätzliches Gewicht auf, abhängig von der Distanz zum Laufweg. Da keine Informationen über die Bewegungsrichtung der Hindernisse vorhanden sind, kann die Laufrichtung der Hindernisse selbst nicht berücksichtigt werden (Abb.: 7.4c).

Die vier Komponenten werden normiert und sind über Parameter an sich verändernde Situationen anpassbar, welche in einer externen Konfigurationsdatei (*Config/Locations/Default/behavior.cfg*) abgelegt sind.

Um der zuvor angesprochenen, stärkeren Modularisierung und Trennung von Berechnung und Entscheidung Rechnung zu tragen, wird die Gewichtsfunktion, anders als die vorherige Balldistanz, nicht

im Ballmodel, sondern in *BehaviorData* hinterlegt. Diese steht über *TeamDataProvider* jedem einzelnen NAO im WLAN zur Verfügung und erlaubt auf diesem Wege jedem NAO selbst zu entscheiden, ob er zum Ball gehen soll.

Darüber hinaus sind umfassende Debugmöglichkeiten in Form von Plots und Drawings implementiert worden, um auch zukünftig Anpassungen und Funktionskontrollen für nachfolgende Gruppen zu erleichtern.

Während der abschließenden Anpassungen und Korrekturen hat sich gezeigt, dass bei einer fehlerhaften Lokalisierung der NAOs eine eindeutige, differenzierte und korrekte Entscheidungsfindung nicht wirklich möglich ist. Insgesamt bleibt festzustellen, dass das neue Verfahren deutlich stärker von Lokalisationsproblemen betroffen ist als das alte. Dies ist vor allem in den vielen positionsabhängigen Teilentscheidungen der neuen Variante begründet.

7.3 AUSWAHLMÖGLICHKEIT FÜR VERSCHIEDENE VERHALTENSMUSTER

Für verschiedene Anwendungsfälle werden verschiedene Verhaltensmodi benötigt. Diese mussten bisher jeweils einzeln neu übertragen und aktiviert werden. Um dies in Zukunft einfacher handhaben zu können, ist eine Auswahlmöglichkeit des aktuell genutzten Verhaltens per Knopfdruck beim Start hinzugefügt worden.

Dabei sind drei Verhalten in Form von XABSL-Agenten auswählbar:

- Der Agent 'Soccer' (Standard Agent) startet im *pre_initial_state*. Per Knopfdruck wird das Standard Spielverhalten gestartet.
- Der Agent 'Penalty' startet im *pre_initial_state_penalty*. Per Knopfdruck wird das Penalty-Shootout Verhalten gestartet.
- Der Agent 'demoSoccer' startet mit dem *pre_initial_state_selector*, in dem per Knopfdruck das Verhalten gewechselt werden kann.

Dieser Verhaltenswechsel im Agenten 'demoSoccer' erlaubt folgende Auswahl:

- Einmal Chestbutton drücken und kurze Zeit warten (Augen leuchten grün) startet das Standard Spielverhalten.
- Zweimal Chestbutton drücken und kurze Zeit warten (Augen leuchten rot) startet ein Demo Spielverhalten.
- Dreimal Chestbutton drücken und kurze Zeit warten startet ein *goto_ball_and_kick*, ohne die Lokalisierung zu beachten.

In der Auswahl 'Demo Spielverhalten' (Augen blinken links blau, rechts rot) kann man zusätzlich den gewünschten Spielertyp wählen:

- Einmal den Chestbutton drücken und kurze Zeit warten startet das Striker Verhalten, der NAO fungiert demnach als Feldspieler (Augen leuchten jetzt rot - zusätzlich wird angezeigt, ob der Ball gesehen wurde).
- Zweimal Chestbutton drücken startet das Keeper Verhalten, der NAO fungiert demnach als Torwart (Augen leuchten jetzt blau und die Anzeige, ob Ball gesehen wurde, ist aktiv).
- Beide Footbumper drücken ohne den Chestbutton gedrückt zu haben: Wenn beide Augen blinken hat man drei Sekunden um den Chestbutton zu drücken und damit das 'Penalty-Shootout' Verhalten zu starten, ansonsten wird in den *initial_state* zurückgekehrt.

Das genannte Standard Spielverhalten besteht aus mehreren, parallel laufenden Prozessen. Zu diesen gehören:

- die Rollenentscheidung,
- 'Head Control' zur Ansteuerung der Kopfbewegungen,
- 'Body Control' als Auslöser für Aufstehen und das Zurückgehen, wenn die Bumper zu oft ausgelöst werden. Hierüber werden zudem die 'Game States' gewechselt,
- 'Display Control', welche die Leuchtfarbe der Augen kontrolliert, über die Debug-Informationen übermittelt werden,
- sowie das offizielle 'Button Interface' entsprechend den aktuellen Regeln der Standard Plattform Liga.

7.4 FAZIT

Bei den Verhaltensentscheidungen müssen alle möglichen Folgen aus einer Situation berücksichtigt werden. Dies hat zur Folge, dass selbst triviale Problemstellungen eine Herausforderung darstellen können. Weiterhin sind alle Möglichkeiten und verfügbaren Optionen der Wahrnehmung und Bewegung in die Verhaltensentscheidungen mit einzubeziehen. Eine Änderung in einer dieser Komponenten kann deutliche Auswirkungen auf andere Bereiche haben. Wie in Kapitel 7.2 beschrieben, hat z.B. die Güte der Lokalisation einen enormen Einfluss auf die Möglichkeiten in der taktischen Planung. Diese Abhängigkeiten von den Möglichkeiten der anderen Komponenten macht es erforderlich, das Verhalten stets an die Fortschritte in den anderen Bereichen anzupassen. Weiterentwicklungen im Rahmen der Bewegung oder Wahrnehmung können derzeitige Ansätze im Verhalten ad-absurdum führen und neue Strategien erfordern. Daraus folgt, dass das Verhalten einem ständigen Wandel unterzogen ist und bleibt.

LITERATURVERZEICHNIS

Atm12 Atmel Corporation: *Atmel Studio*. http://www.atmel.com/tools/atmelstudio.aspx. Version: 2012

BV02 Bruce, James ; Veloso, Manuela: Real-time randomized path planning for robot navigation. In: *Proceedings of IROS-2002*. Switzerland, October 2002

DAVV David A. Van Veldhuizen, Gary B. L.: *Evolutionary Computation and Convegence to a Pareto Front*. Wright-Patterson AFB,

FR94 Ford, A. ; Roberts, A.: *Colour Space Conversions*. http://www5.informatik.tu-muenchen.de/lehre/vorlesungen/graphik/info/csc/. Version: 1994

Fur08 Furlan, Peter: *Das gelbe Rechenbuch 3*. Verlag Martina Furlan, 2008. – ISBN 3–931645–02–9

GA10 Grosse-Allermann, Jens: *Verbesserte Kamerakalibrierung durch iterative Annäherung der Kalibrierungspunkte*. Bottrop, 2010

GB08 Gary Bradski, Adrian K.: *Learning OpenCV*. Sebastopol : O'Reilly Media Inc., 2008

Han98 Hanne, Thomas: *Multikriterielle Optimierung: Eine Übersicht*. 1998

Hau08 Hauschildt, Daniel: *Implementierung von Hard- und Software für Servomotoren in der Robotik*. 2008

Lun08 Lunze, Jan: *Regelungstechnik 1*. Springer, 2008. – ISBN 978–3–540–68907–2

MLL08 Minguez, Javier ; Lamiraux, Florent ; Laumond, Jean-Paul: Motion Planning and Obstacle Avoidance. In: Siciliano, Bruno (Hrsg.) ; Khatib, Oussama (Hrsg.): *Springer Handbook of Robotics*. Springer, 2008. – ISBN 978–3–540–23957–4, S. 827–852

NM65 Nelder, J. A. ; Mead, R.: A Simplex Method for Function Minimization. In: *The Computer Journal* 7 (1965), Januar, Nr. 4, 308–313. http://dx.doi.org/10.1093/comjnl/7.4.308

Ope12 *OpenServo*. http://www.openservo.com/. Version: 2012

PBP02 Prautzsch, Hartmut ; Böhm, Wolfgang ; Paluszny, Marco: *Bezier and B-spline techniques*. Berlin [u.a.] : Springer, 2002 (Mathematics and visualization)

QCM03 Quinlan, M.J. ; Chalup, S.K. ; Middleton, R.H: Application of SVMs for Colour Classification and Collision Detection with AIBO Robots. In: *Proceedings for NIPS*. Whistler, BC, Canada, 2003

QT212 *Qt - Cross-platform application and UI framework*. http://qt.nokia.com/. Version: 2012

Rud92 Rudolph, Günter: On Correlated Mutations in Evolution Strategies. In: *Parallel Problem Solving from Nature, 2*. Amsterdam, 1992, S. 105–114

Unb07 Unbehauen, Heinz: *Regelungstechnik 1*. Friedr. Vieweg und Sohn Verlag, 2007. – ISBN 978–3–8348–0230–9

Wei07 Weicker, Karsten: *Evolutionäre Algorithmen 2. Auflage*. Stuttgart : Teubner, 2007