HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Pfadplanung und Schrittkoordination für Humanoide Roboter

Bachelorarbeit

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

eingereicht von: Yigit Can Akcay
geboren am: 26.10.1992
geboren in: Bielefeld

Gutachter/innen: Prof. Dr. Verena Hafner
Prof. Dr. Hans-Dieter Burkhard

eingereicht am: .... verteidigt am: ....

Inhaltsverzeichnis

1 Einleitung

Die Fähigkeit zu navigieren ist essenziell für viele Anwendungen humanoider Roboter und stellt eine nicht trivial lösbare Herausforderung dar. Ein Anwendungsgebiet für Roboter dieser Art ist der RoboCup, bei dem verschiedene Teams in einem Fußballspiel gegeneinander antreten.

Im Fußball sind eine schnelle Fortbewegung, mit der Berücksichtigung von anderen Spielern, und eine Interaktion mit dem Ball wichtige Bestandteile. Die Herausforderung drückt sich hierbei auf zwei Ebenen aus: der Pfadplanung und der Schrittplanung.

Die Pfadplanung beschäftigt sich damit einen kollisionsfreien und möglichst optimalen Pfad zwischen Roboter und Ziel zu berechnen. Dabei müssen Hindernisse berücksichtigt und erfolgreich vermieden werden. Die Schrittplanung hingegen ist verantwortlich für die Koordination der Schritte, wodurch eine präzise Interaktion mit dem Ball erst möglich wird.

In dieser Arbeit werden Algorithmen für die Schrittplanung und Pfadplanung im Kontext der *Standard Platform League* (SPL) untersucht. Die SPL ist eine der Ligen des RoboCupSoccer. In dieser Liga treten verschiedene Teams mit dem selben Robotermodell des Herstellers SoftBank Robotics, dem Nao, in einem Fußballspiel gegeneinander an. Ein Fußballspiel stellt eine hochdynamische Umgebung dar. Hinzu kommt, dass diese Roboter verglichen mit herkömmlichen Computern deutlich begrenztere Ressourcen wie Rechenleistung und Arbeitsspeicher besitzen, weshalb die genutzten Algorithmen Rücksicht auf diesen Umstand nehmen müssen. Aus diesem Grund können bereits vorhandene Lösungen in der SPL oft nicht genutzt werden.

In einem RoboCup Spiel gibt es zwei Situationen, in denen sich ein Roboter befinden kann, die für diese Arbeit interessant sind. Der Roboter ist entweder so weit entfernt vom Ball, dass er ihn nicht kontrolliert, oder der Roboter ist am Ball und übt direkte Kontrolle aus. Befindet sich der Roboter in der ersten Situation, ist es sein Ziel an den Ball zu laufen um Kontrolle ausüben zu können. Befindet sich der Roboter in der zweiten Situation, ist es sein Ziel die Kontrolle über den Ball zu behalten, um anschließend den Ball Richtung gegnerisches Tor zu befördern. Die erste Situation wird im Folgenden Ballanlauf genannt, die Zweite Ballkontrolle.

Diese Arbeit wurde auf Grundlage des NaoTH Frameworks entwickelt, auf das in Abschnitt 2.1 und Abschnitt 2.2 näher eingegangen wird. Dieses Framework ermöglicht einen stabilen Gang, so dass sich in dieser Arbeit auf die Pfadplanung und Schrittkoordination konzentriert wird, mit derer das Problem des Ballanlaufs und der Ballkontrolle gelöst werden können.

1.1 Beitrag der Arbeit

Der Beitrag dieser Arbeit kann im Wesentlichen in zwei Bereiche unterteilt werden: Pfadplanung und Schrittkontrolle.

Die Walking Engine des NaoTH Framework wurde um ein Modul zur expliziten Schrittkontrolle erweitert, wie in Abschnitt 2.1, Abschnitt 2.2 und Abschnitt 2.5 beschrieben. Es wurden verschiedene Schritttypen für diese Schrittkontrolle eingeführt. Basierend darauf wurden verschiedene Schüsse realisiert, wie etwa Vorwärts- und Seitwärtsschuß. Durch die explizite Kontrolle können die Schüsse besonders präzise ausgeführt werden, wodurch die Anzahl der Fehlschüsse, insbesondere das Verstolpern des Balls, deutlich verringert wird. Die Algorithmen wurden für den Roboter Nao implementiert und bei der GermanOpen und in der Weltmeisterschaft in Japan erfolgreich eingesetzt.

Um globale Navigation und Hindernisvermeidung zu realisieren wurden zwei verschiedene Pfadplanungsalgorithmen untersucht. Für beide Algorithmen wurde eine abstrakte Simulation implementiert und eine empirische Analyse durchgeführt. Die Algorithmen wurden ebenfalls für den Roboter Nao umgesetzt und unter Laborbedingungen auf einem realen Roboter getestet.

1.2 Verwandte Arbeiten

Die Frage der Pfadplanung ist ein gut untersuchtes Gebiet. Eine gute Einführung bietet zum Beispiel das Standardwerk *Handbook of Robotics* [11]. Grundsätzlich kann man Ansätze zur Lösung der Pfadplanung in vielerlei Hinsicht kategorisieren, beispielsweise abhängig von der Holonomieeigenschaft der Roboter.

Ein Beispiel für einen Algorithmus, der speziell für fahrende, nicht-holonome Roboter optimiert ist, wird in [2] von Jolly, Kumar und Vijayakumar vorgestellt. Die Grundlage des Algorithmus bilden Bézierkurven, wodurch die Beschleunigung des Roboters kontrolliert werden kann. Solche Ansätze sind für den Fall humanoider Roboter weniger relevant. Der Vorteil, der sich für nicht-holonome, fahrende Roboter bildet, fällt nämlich für humanoide Roboter weg, da der Vorteil aus einer Kombination der Kontrolle über die Beschleunigung und der Fortbewegung auf Rädern entsteht. Fahrende, nicht holonome Roboter hingegen können dadurch beispielsweise mit dem Ball interagieren.

Zwei besonders interessante Algorithmen für den Kontext dieser Arbeit sind [9] und [10]. Nieuwenhuisen, Steffens und Behnke stellen in [9] eine Pfadplanung mit Hilfe des A*-Graphsuchalgorithmus und verschiedenen Gittertypen auf Grundlage humanoider Roboter vor. Dabei untersuchen sie die Auswirkungen von uniformen, nicht uniformen und logarithmischen Gittern. Letztere werden in Polarkoordinaten repräsentiert. Rodríguez, Rojas, Pérez, López, Quintero und Calderón stellen in [10] einen Vergleich zwischen dem Rapidly-exploring Random Trees Algorithmus (RRT) und einem von den Autoren neu entwickelten Algorithmus vor. Dabei erzielt der vorgeschlagene Algorithmus bessere Werte als RRT, in dem sie den geraden Pfad zwischen Roboter und Ziel an dem zum Roboter nahsten Kollisionspunkt mit einem Hindernis verschieben. Anschließend wird der Prozess vom Startpunkt zum Kollisionspunkt und vom Kollisionspunkt zum Ziel wiederholt bis ein kollisionsfreier Pfad entsteht.

Diese drei Algorithmen haben drei Gemeinsamkeiten. Die erste Gemeinsamkeit liegt in der globalen Pfadplanung, das heißt, dass Veränderungen der Umgebung nach der Planung des Pfades nicht berücksichtigt werden, beispielsweise durch eine Filterung der berechneten Pfade. Die zweite Gemeinsamkeit liegt in einem expliziten Hindernismodell. Damit ist gemeint, dass ein anderer Algorithmus Hindernisse erkennt um ein Modell zu erstellen, auf dass der Pfadplanungsalgorithmus dann zugreifen kann. Eine weitere, dritte Gemeinsamkeit liegt in der abstrakten Form der Pfadplanung. Damit ist gemeint, dass diese Algorithmen einen Fokus auf einen Pfad im Bezug auf eine Hindernisvermeidung setzen und nicht auf eine Schrittplanung, dessen Fokus auf ausführbaren und stabilen Schritten liegt. Auf diese Pfadplanungsalgorithmen wird in Abschnitt 3 näher eingegangen.

Deits und Tedrake stellen in [1] einen Pfadplanungslgorithmus vor, der seinen Fokus auf die Schrittplanung setzt. Dieser Algorithmus plant einen optimalen und kollisionsfreien Pfad in Abhängigkeit zur sicheren Platzierung der Füße auf unebenem Terrain. Um dies zu bewerkstellingen werden Flächen mit unterschiedlicher Höhe und Neigung berücksichtigt, was zu einer hohen Rechenkomplexität führt. Zusätzlich wird von einem statischen Umfeld ausgegangen. Diese Eigenschaften machen die vorgestellte Lösung für den Gebrauch in der SPL des RoboCup uninteressant.

Relevanter für den RoboCup sind Schrittplanungsalgorithmen, die zweibeiniges Laufen für humanoide Roboter auf möglichst ebenem Terrain, sowie eine Interaktion mit dem Ball ermöglichen.

In [12] wird der aktuell im NaoTH Framework verwendete Algorithmus zur Schrittplanung und Ausführung beschrieben. Dabei wird der Körper des Roboters als Massepunkt modelliert und dessen Dynamik als invertiertes Pendel. Die einzelnen Schritte werden explizit geplant und die Bewegung des Masseschwerpunktes daraus errechnet, um die Stabilität des Roboters zu gewährleisten. Dieser Ansatz erlaubt eine explizite Kontrolle der Füße während des Laufens, was beispielsweise die Realisierung von Schüssen aus dem Lauf heraus ermöglicht.

Missura verfolgt in [7] einen Ansatz, bei dem die Laufbewegung des Roboters aus einer zentral generierten Schwingung, dem Central Pattern Generator, entsteht. Die Fußpositionen werden bei solchen Ansätzen in der Regel nicht direkt gesteuert. Durch die Einführung einer zusätzlichen, expliziten Schrittkontrolle wird hier ein deutlich stabilerer Lauf erreicht, was mit sogenannten Capture Steps auch starke Stöße von Außen abfangen kann. Dieser Algorithmus basiert auf dem Capture Step Framework, welches in [8] vorgestellt wird.

In [6, 13] werden Algorithmen für eine dynamische Schrittkontrolle vorgestellt, mit Hilfe derer Schüsse realisiert werden können. Dabei wird ein Schuß in vier Phasen untergliedert. Die erste Phase ist die Vorbereitungsphase. In dieser Phase verlagert der Roboter sein Gewicht auf den Standfuß und hebt den Schußfuß an. In der zweiten Phase wird der Schußfuß zurückgezogen. In dieser Phase reagiert der Roboter auf Bewegungen des Balls und passt die Position des Schußfußes dementsprechend an. Anschließend wird der Schuß in der dritten Phase ausgeführt. In der vierten Phase geht der Roboter zurück in seine Ausgangsstellung.

1.3 Struktur der Arbeit

In Abschnitt 2 wird auf die Grundlagen eingegangen, die nötig sind um die in dieser Arbeit beschriebenen Algorithmen nachvollziehen zu können. Dazu gehören die technischen Details des NaoTH Frameworks, die ihr zugrunde liegende Walking Engine und die Theorie zur Struktur und Geometrie von Schritten.

In Abschnitt 3 werden verschiedene Algorithmen zur Pfadplanung vorgestellt, die im Zuge dieser Arbeit untersucht wurden. In diesem Abschnitt werden gegebenenfalls Verbesserungen für die vorgestellten Algorithmen vorgeschlagen, die während der Implementierung entstanden sind.

In Abschnitt 4 wird die Ballkontrolle behandelt. Dabei wird auf die Bewegung am Ball, den Übergang vom Ballanlauf zur Ballkontrolle und auf die Interaktion mit dem Ball, in Form von Schüssen, näher eingegangen.

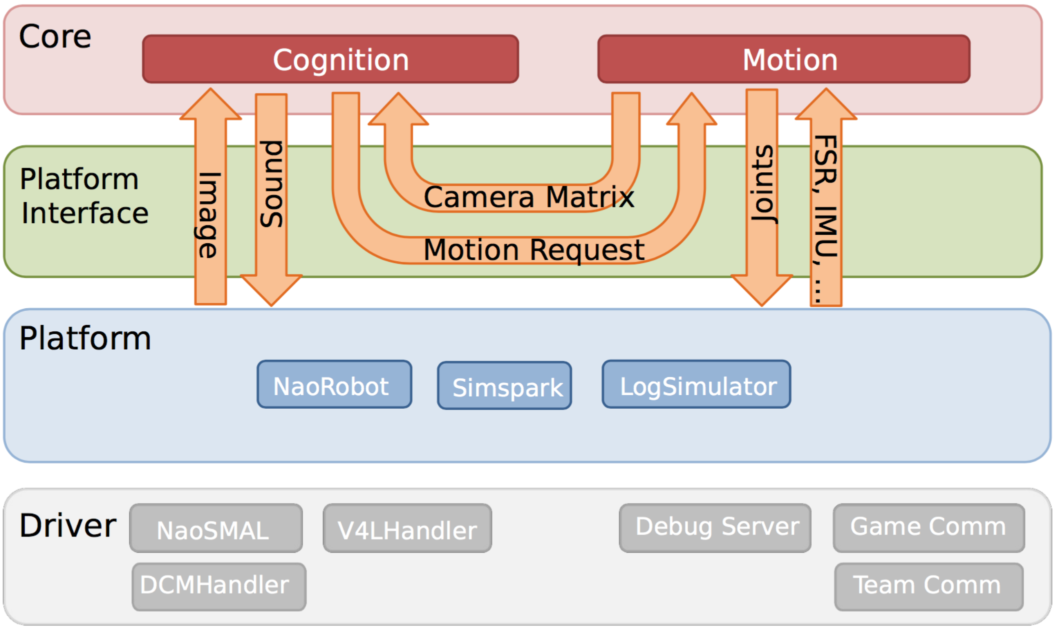


Abbildung 1: *Übersicht über das NaoTH Framework. Die Kernmodule bilden den plattformunabhängigen Teil. Bild entnommen aus [5].*

2 Grundlagen

In diesem Abschnitt werden die technischen und theoretischen Grundlagen erläutert, auf denen diese Arbeit basiert. Auf eine Erklärung des *Zero Moment Point* (ZMP) und des *Linear invertiertem Pendelmodell* (LIPM) wird verzichtet, da sie nicht essenziell zum Verstehen dieser Arbeit sind. Stattdessen wird für eine nähere Erläuterung auf [12] verwiesen.

2.1 NaoTH Framework

Die Grundlage der technischen Umsetzung dieser Arbeit liefert das Framework der Forschungsgruppe *Nao Team Humboldt* (NaoTH) der Humboldt Universität zu Berlin, welcher Teil des Forschungslabors für Adaptive Systeme ist. In [5] wird die Architektur dieses Frameworks erläutert. Ein Überblick des Frameworks ist in Abb. 1 zu sehen.

Das Framework ist aufgeteilt in plattformabhängige und platformunabhängige Teile. Diese Aufteilung ermöglicht es den selben Code für Simulatoren und den Nao zu verwenden. Dadurch wird das Ziel vorangetrieben, die Lücke zwischen Simulation und der realen Ausführung auf Robotern zu schließen [12].

Der platformunabhängige Bereich besteht aus den Kernprozessen Cognition und Motion. Cognition ist für die Wahrnehmung und Interpretation verantwortlich.

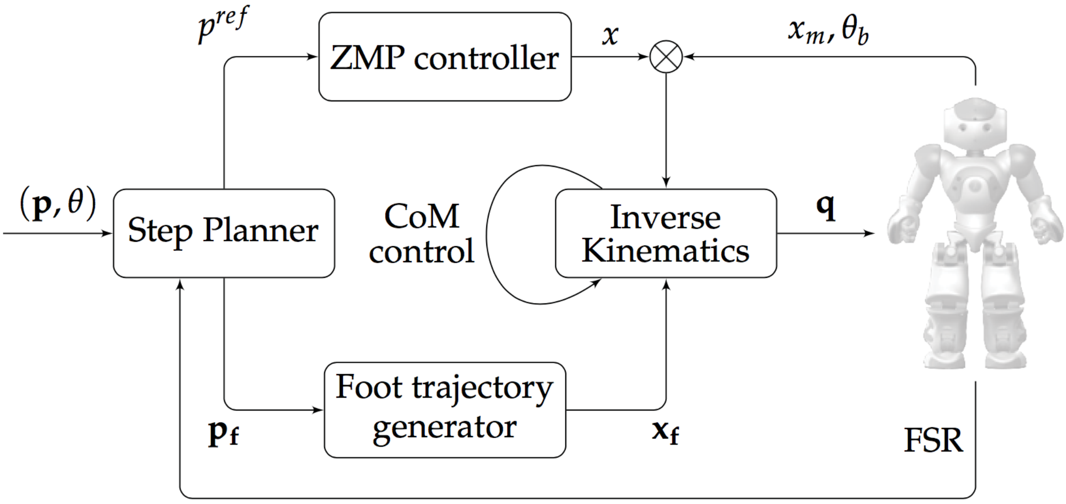


Abbildung 2: *Übersicht über die Architektur der Walking Engine. Bild entnommen aus [12].*

Hierzu werden Sensordaten, wie beispielsweise der Kamera, ausgewertet. Motion steuert die Motoren der Gelenke des Roboters, um von Cognition angefragte Bewegungen ausführen zu können.

Die in dieser Arbeit vorgestellte Lösung siedelt sich in dem Bereich zwischen Cognition und Motion in der Verhaltenssteuerung an. Dabei wird die in Motion vorhandene Walking Engine genutzt. Anfragen aus der Verhaltenssteuerung, abhängig von Daten aus Cognition, werden so getätigt, dass einzelne Schritte ausgeführt werden. Die Walking Engine ist verantwortlich für die sichere Ausführung der Anfragen.

2.2 Walking Engine

Die Walking Engine verfolgt einen modellbasierten Ansatz [12]. Das Zero Moment Point Modell wird als Kriterum für die Stabilität eines Schrittes genutzt. Mithilfe des Linear Inverse Pendulum Modells wird die Dynamik des Roboters approximiert.

Als Teil der Motion erhält die Walking Engine *Walkrequests*, die das Ziel des Ganges mithilfe eines 2D Punktes (x_r, y_r) und einem Rotationswinkel θ_r beschreiben. Durch einen Parameter c wird im *Walkrequest* angegeben, welches lokale Koordinatensystem genutzt werden soll. Für dieses Koordinatensystem gibt es drei Optionen: der linke Fuß, der rechte Fuß oder der Rumpf. Zusätzlich wird durch einen Parameter c_{char} angegeben, wie schnell ein Schritt ausgeführt werden soll. Dieser Parameter wird erst durch einen weiteren, globalen Parameter aktiviert und hat ebenfalls drei Optionen: *stable*, *normal* und $fast$. Auf die genaue Funktionalität dieses Parameters wird in Abschnitt 2.5 eingegangen. $(x_r, y_r, \theta_r, c, c_{char})$ bildet die

Schnittstelle der *Walkrequests*.

Die Walking Engine besitzt zusätzlich eine weitere Schnittstelle zur Steuerung der Schritte, *Stepcontrol* genannt, bei der zusätzliche Parameter angegeben werden müssen, wie z.B. welcher Fuß den angefragten Schritt ausführen soll. Die zusätzlichen Parameter sind f, t, s und d. $(x_r, y_r, \theta_r, c, c_{char}, f, t, s, d)$ bildet die Schnittstelle der *Stepcontrol*, wobei das Flag f den einzusetzenden Fuß angibt. Die Funktionen der restlichen, zusätzlichen Parameter werden in Abschnitt 2.5 näher erläutert.

Im Zuge dieser Arbeit wurde die Schnittstelle der *Stepcontrol* um einen weiteren Parameter p_{type}, der den Typ des Schrittes angibt, erweitert. Die aktualisierte Schnittstelle ist $(x_r, y_r, \theta_r, c, c_{char}, f, t, s, d, p_{type})$. *Walkrequests* sind immer noch ein Bestandteil des Frameworks. Dies hat Gründe der Rückwärtskompatibilität. Im weiteren Verlauf dieser Arbeit wird ausschließlich die Schnittstelle *Stepcontrol* genutzt. Ein Schritt ist demnach ein *Stepcontrol*-Tupel. Dieses Tupel wird im Folgenden *Steprequest* genannt.

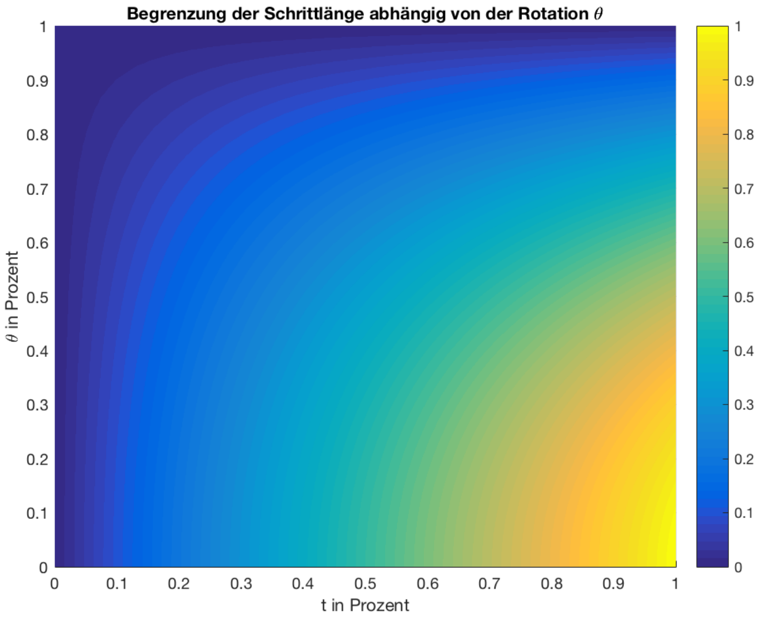


Abbildung 3: *t ist die angefragte, beschränkte Translation als prozentualer Anteil der maximalen Schrittlänge t_{max}. θ_s ist die angefragte, beschränkte Rotation als prozentualer Anteil der maximalen Rotation θ_{max}. Sie schränkt t weiter ein.*

Die Ausführung einer *Walkrequest* oder einer *Steprequest* erfolgt in mehreren Schritten [12]. Zu erst muss der Stepplanner den nächsten Schritt p_f für den bewegbaren Fuß auf dem Weg zum Ziel (x_r, y_r) festlegen. Die Koordinaten dieses Schrittes sind abhängig von der maximalen Schrittlänge, die von der technischen Umsetzung des Nao Roboters festgelegt und durch weitere Stabilitätskriterien eingegrenzt wird. Darauf wird in Abschnitt 2.3 näher eingegangen. Gleichzeitig wird der Zero Moment Point p^{ref} festgelegt. Im nächsten Schritt erstellt der Foot Trajectory Generator die Fußtrajektorie x_f und der ZMP Controller berechnet das Massezentrum x abhängig von p^{ref}. Anschließend werden die Winkel q der Gelenke abhängig von x und x_f durch Lösen der inversen Kinematik bestimmt.

2.3 Ausführung von Walk- und Steprequests

Bevor der Roboter einen Schritt, den der Footstepplanner in Form einer *Walk*- oder *Steprequest* erhält, ausführt, wird dieser Schritt im Footstepplanner abhängig von den folgenden zwei Parametern begrenzt:

$$\theta_{max} \tag{2.3.1}$$

$$t_{max} = (t_x,\, t_y) \tag{2.3.2}$$

θ_{max} ist die maximal mögliche Rotation und t_{max} die maximal mögliche Translation in einem Schritt. Die Rotation θ_s und die Translation t_s für einen Schritt werden folgendermaßen berechnet:

$$\theta_s = \min(|\theta_a|,\, \theta_{max}) \cdot \operatorname{sign}(\theta_a) \tag{2.3.3}$$

$$b_{tx} = \min(|t_{ax}|,\, t_x) \cdot \operatorname{sign}(t_{ax}) \tag{2.3.4}$$

$$b_{ty} = \min(|t_{ay}|,\, t_y) \cdot \operatorname{sign}(t_{ay}) \tag{2.3.5}$$

$$t_s = (b_{tx},\, b_{ty}) \cdot \cos(\frac{\theta_s \cdot \pi}{\theta_{max} \cdot 2}) \tag{2.3.6}$$

$t_a = (t_{ax},\, t_{ay})$ ist die angefragte Translation und θ_a die angefragte Rotation. Die angefragte Rotation und Translation wird auf die maximale Rotation und maximale Schrittlänge begrenzt, sofern sie diese überschreitet. Anschließend wird die begrenzte Translation b_{tx} und b_{ty} durch die Rotation in diesem Schritt θ_s weiter eingeschränkt. Ziel dieser Begrenzung durch die Rotation ist die Stabilisierung des Ganges mit Berücksichtigung der Beschleunigung des Roboters. In Abb. 3 ist die Begrenzung der Schrittlänge t abhängig von der Schrittrotation θ_s zu sehen.

Anschließend werden für *Walkrequests* oder *Steprequests* mit $p_{type} = Walkstep$ θ_s und t_s weiter beschränkt, so dass die Änderung dieses Schrittes zum vorherigen Schritt die Parameter max_{ct} für die Translation und max_{cr} für die Rotation nicht überschreitet. In unseren Experimenten haben die Parameter $max_{ct} = (13.5, 15)$ und $max_{cr} = 10$ gut funktioniert. Auf die p_{type} wird in Abschnitt 2.5 näher eingegangen.

Zu letzt werden noch geometrische Restriktionen auf die Schritte angewandt, so dass beispielsweise der rechte Fuß keinen Schritt ausführen kann, der mit dem linken Fuß kollidieren würde, oder ein Fuß sich nicht mehr rotieren kann, als die Gelenke des Roboters zulassen.

2.4 Lokale Koordinaten des Roboters

Die lokalen Koordinaten des Roboters im linken Fuß, im rechten Fuß und im Rumpf sind in Abb. 4 zu sehen.

Während einer Bewegung bewegt sich das Koordinatensystem des Rumpfes und des bewegten Fußes mit, das Koordinatensystem des Standfußes ist fest. Deshalb wird intern mit dem Koordinatensystem des Standfußes gerechnet. Hierfür wird der Zielpunkt im angegebenen Koordinatensystem wenn nötig in das Koordinatensystem des Standfußes umgerechnet.

Der Grund für die drei verschiedenen Koordinatensysteme liegt in der Einfachheit die dadurch entsteht für Bewegungen der Füße relativ zum Ball. Dadurch wird ein intuitiver Umgang ermöglicht. Beispielsweise kann der Ball mittig vom Roboter gehalten werden, in dem das lokale Koordinatensystem des Rumpfes genutzt wird. Möchte man mit dem rechten Fuß einen Vorwärtsschuß realisieren, kann man die lokalen Koordinaten des rechten Fußes nutzen um den Fuß mit dem Ball auszurichten.

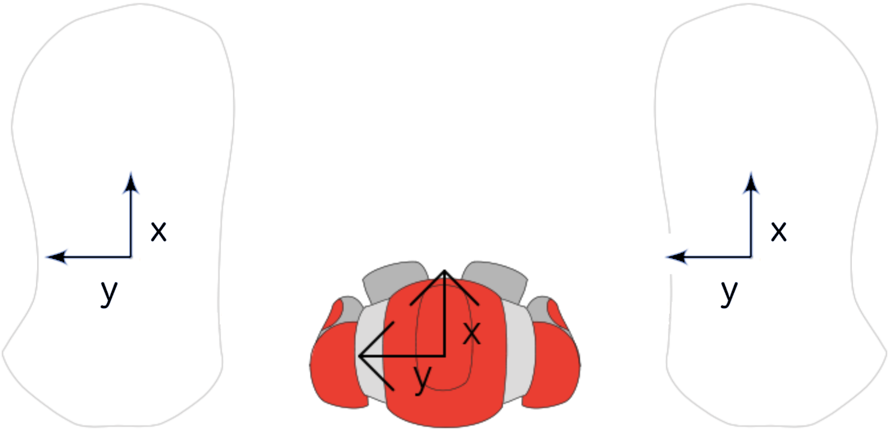


Abbildung 4: *Die lokalen Koordinaten des Roboters im linken Fuß, rechten Fuß und Roboterrumpf.*

2.5 Geometrie eines Schrittes

Ein Schritt wird definiert durch einen 2D Start- und Endpunkt, einer 3D Trajektorie und einem Typ. Der Start- und Endpunkt ist dabei relativ zu dem Punkt, an dem der Roboter gestartet wurde. Der Startpunkt des Schrittes ist der Punkt, an dem sich der Fuß vor der Ausführung des Schrittes befindet. Der Endpunkt des Schrittes ist der Punkt, an dem der Fuß nach der Ausführung des Schrittes sein soll.

In Abb. 5 ist die Geometrie eines Schrittes zu sehen. P_{sup} stellt den bei der Schrittausführung unterstützenden Fuß dar, P_{mov} entsprechend den zu bewegenden Fuß. d_{fs} ist der Abstand der Füße zueinander in der Ausgangsstellung, in der beide nebeneinander sind. Der unterstützende Fuß wird als Koordinatenursprung verwendet. O stellt die Haltung des Roboters vor dem Schritt, O' die Haltung

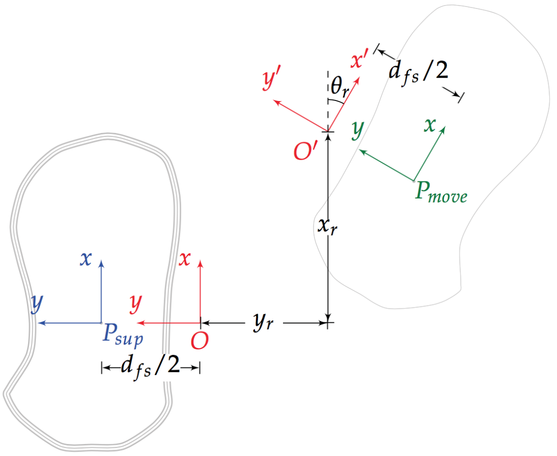


Abbildung 5: *Die Geometrie eines Schrittes. Bild entnommen aus [12].*

des Roboters nach dem Schritt dar. [12] zufolge wird der unterstützende Fuß als Koordinatenursprung verwendet, da der Rumpf des Roboters sich dazu nicht eignet. Das liegt daran, dass die Haltung des Rumpfes sich während der Bewegung ständig ändert.

Um die Bewegung des Schrittes auszuführen wird eine 3D Trajektorie interpoliert. Die Kriterien für die Interpolationsfunktion sind die folgenden [12]:

- Die horizontale Beschleunigung ist gleich 0, wenn der Fuß Kontakt mit dem Boden hat.
- Der gehobene Fuß soll so lange wie möglich vom Boden entfernt bleiben.
- Die Trajektorie soll glatt sein. Dafür muss die Beschleunigung stetig sein.

Es existieren drei verschiedene Schritttypen p_{type}: *Walkstep*, *Kickstep* und *Zerostep*.

Walksteps werden für das Laufen benutzt und funktionieren wie normale *Walkrequests*. Das bedeutet, dass die Trajektorie für den Schritt durch folgende Interpolationsfunktionen berechnet wird [12]:

$$\alpha(t) = \frac{1}{1 + \lambda \cdot e^{-(t-0.5)}} \tag{2.5.1}$$

$$s(t) = (1 - \alpha(t)) \cdot S_0 + \alpha(t) \cdot S_1 \tag{2.5.2}$$

$$\beta(t) = \cos((t - 0.5) \cdot \pi) \tag{2.5.3}$$

$$h(t) = \beta(t) \cdot H \tag{2.5.4}$$

t ist der normalisierte Zeitzyklus zwischen 0 und 1. λ ist der Kurvenfaktor, der durch Experimente auf 7 festgesetzt wurde. S_0 und S_1 sind der Start- und Endpunkt des Schrittes. H ist die maximale Schritthöhe, welche gegeben ist. $\alpha(t)$ berechnet die Trajektorie des Schrittes in der Horizontalen, $\beta(t)$ in der Vertikalen. $s(t)$ und $h(t)$ sind entsprechend die Positionen des bewegten Fußes in der Horizontalen und der Vertikalen.

Bei *Walksteps* wechselt der Fuß, der bewegt wird, nach jedem Schritt. Das heißt, es ist nur der Fuß bewegbar, welcher als letztes nicht bewegt wurde. Beide Füße sind bewegbar nach *Zerosteps*, oder wenn der Roboter sich im Stand befindet, das heißt, wenn seine Beschleunigung 0 ist. Wenn beide Füße bewegbar sind, wird der Fuß, der als nächstes bewegt wird, folgendermaßen ausgewählt:

- Ist die angefragte Rotation oder Translation an der y-Achse größer 0, nutze den linken Fuß.
- In allen anderen Fällen nutze den rechten Fuß.

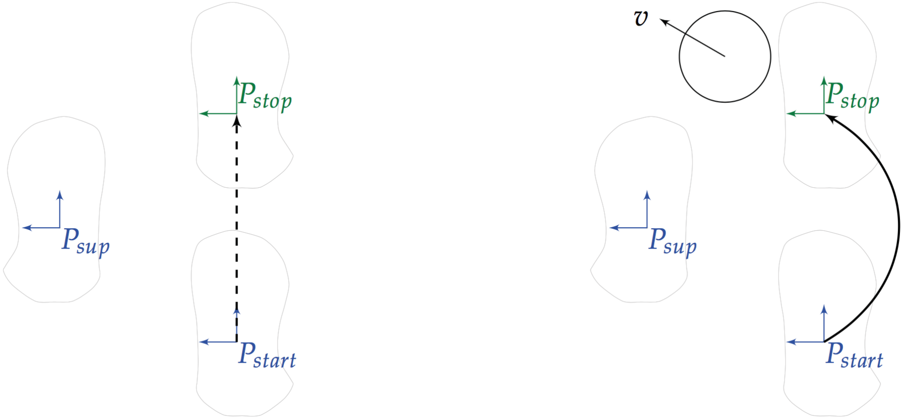


Abbildung 6: *Links die Trajektorie eines Walksteps, rechts die Trajektorie eines seitlichen Kicksteps. Bild entnommen aus [12].*

Zerosteps sind besondere Schritte, die keine Bewegung hervorrufen. Ihr Start- und Endpunkt sind derselbe, deshalb besitzen sie keine 3D Trajektorie.

Kicksteps sind Schritte, die für Vorwärts- und Seitwärtskicks benutzt werden. Ihre Trajektorie wird nicht mit der Gleichung (2.5.2) berechnet, sondern durch kubische Splines. Für diesen Typ Schritt spielt der Parameter d eine besondere Rolle. d entscheidet darüber, in welche Richtung der Ball bei erfolgreichem Schuß bewegt wird, indem die Punkte der berechneten Splinetrajektorie durch d beeinflusst werden. Darauf wird in Abschnitt 4.5 näher eingegangen. Die Schritttrajektorien sind in Abb. 6 zu sehen.

Die Parameter t und s sind für *Walksteps* und *Kicksteps* interessant. Mit dem Parameter t wird die Zeit, die der Schritt insgesamt benötigen soll, in Millisekunden angegeben. Der Parameter s sorgt für eine Stauchung des normalisierten Zeitzyklus t der Interpolationfunktionen. s ist demnach ein Wert zwischen 0 und 1, und ein Wert näher an 0 sorgt für eine schnellere Ausführung als durch den Parameter t angefragt wurde. Der Parameter c_{char} wird aus Gründen der Rückwärtskompatibilität zu *Walkrequests* beibehalten. Wenn dieser Parameter aktiviert ist überschreibt er den Parameter t für *Walksteps* mit einem Wert abhängig davon, ob $c_{char} = stable$, $c_{char} = normal$ oder $c_{char} = fast$ ist.

2.6 Kontrolle der Füße

Um eine sinnvolle Schrittkoordination bewerkstelligen zu können ist eine genaue Kontrolle über die Füße notwendig. Die Schnittstelle der *Walkrequests* vertraut auf eine implizite Auswahl des Fußes, der den angefragten Schritt ausführen soll. Diese Auswahl wird im Footstepplanner getroffen. Dabei wird zwischen den Füßen alterniert.

Mit den Erweiterungen, die an der Schnittstelle der *Steprequests* gemacht wurden, können die Füße explizit gewählt werden, die den angefragten Schritt ausführen sollen. Dadurch können Füße präzise platziert werden. Steht der Roboter beispielsweise vor dem Ball, so dass der Ball in den lokalen Koordinaten des rechten Fußes am Punkt $p_r = (p_{rx}, 0)$ liegt, kann explizit der linke Fuß einen Schritt zum Punkt $p_l = (0, p_{ly})$ in den lokalen Koordinaten des linken Fußes machen, um mit dem rechten Fuß anschließend einen Vorwärtsschuß auszuführen. Mit einer impliziten Auswahl der Füße könnte stattdessen der rechte Fuß einen schrägen Vorwärtsschritt und anschließend der linke Fuß einen Vorwärtsschuß ausführen.

Ein weiteres Szenario, in dem die explizite Wahl der Füße von Vorteil ist, bildet sich bei Seitwärtsschüssen. Nach Ausführung des Schusses kann der Schußfuß nun explizit zurückgezogen werden. Mit einer impliziten Auswahl der Füße ist nach dem Schuß nur der Standfuß bewegbar. Dadurch kann nur der Standfuß einen Vorwärtsschritt ausführen um beide Füße auf die selbe Höhe in eine Ausgangsposition zu bringen. Dieser Vorwärtsschritt kann den Seitwärtsschuß torpedieren.

3 Pfadplanung

In diesem Abschnitt wird die Pfadplanung behandelt mit Hilfe derer der Ballanlauf gelöst werden kann. Die Problemstellung der Pfadplanung drückt sich folgendermaßen aus: Gegeben seien zwei Punkte p_s und p_z und eine Menge von Hindernissen. Finde einen Pfad zwischen den Punkten, der kollisionsfrei und möglichst optimal ist. Dabei kann beispielsweise die Länge des Pfades ein Faktor der Optimalität sein.

Bei der Pfadplanung kann unterschieden werden zwischen der globalen und der lokalen Planung. Eine globale Pfadplanung geht von einer statischen und keiner dynamischen Umgebung aus. Diese Art der Pfadplanung ist von Vorteil für Umgebungen, in denen die möglichen Wege sich nicht ändern, bis das Ziel erreicht wurde. Ein Beispiel für eine solche Umgebung ist eine Stadt. Eine Navigationssoftware berechnet den kompletten Weg mit dem Auto von A nach B in dieser Stadt. Auf dem Weg bewegen sich zwar andere Autos, Ampeln ändern ihre Farbe und Baustellen entstehen, der geplante Pfad ändert sich aber nicht insofern, dass der gefundene Weg inakzeptabel lang oder gar falsch wird, in einer Zeitspanne, die während dem Folgen des Pfades relevant ist.

Die *Standard Platform League* (SPL) des RoboCup bietet allerdings eine hoch dynamische Umgebung in Form eines Fußballspiels. Die gegnerischen Roboter und auch der Ball können sich in jedem Zeitpunkt bewegen und nur kurzfristig in ihrer Position verharren. Der Pfad der durch eine globale Pfadplanung entsteht, die nach der Planung des Pfades Veränderungen in der Umgebung nicht berücksichtigt, ist deshalb ungültig, bevor der Roboter den Pfad abläuft und am Ziel, zum Beispiel dem Ball, ankommt.

Die lokale Pfadplanung hingegen bietet einen Vorteil in hoch dynamischen Umgebungen, indem der Pfad nur bis zu einer bestimmten Länge geplant wird, die mit hoher Wahrscheinlichkeit ausführbar ist bevor sie ungültig oder ungünstig wird, oder den Pfad modifiziert oder komplett neu berechnet, wenn die wahrgenommene Umgebung sich verändert.

Eine weitere Möglichkeit Pfadplanungsalgorithmen zu kategorisieren ist die Unterscheidung zwischen Algorithmen die vorteilhaft sind für holonome oder nicht holonome Roboter. Holonomie ist gegeben, wenn alle Freiheitsgrade des Roboters kontrollierbar sind. Ein omnidirektionaler, humanoider Roboter ist im Idealfall holonom. Ein Roboter, der auf vier Rädern fährt und dessen vordere Räder bewegbbar sind während die beiden Hinteren starr in ihrer Position verharren, ist nicht holonom. Unter Umständen können die Vorteile des Algorithmus, die sich für nicht-holonome Roboter bilden, für holonome Roboter wegfallen.

Ein Beispiel für einen solchen Algorithmus bietet [2], worin ein Algorithmus beschrieben wird, der einen Pfad mit Hilfe von Bézierkurven plant. Es werden vier Kontrollpunkte für die Bildung der Bézierkurve genutzt, der Start- und Endpunkt des Pfades und zwei Punkte dazwischen. Die zwei Punkte zwischen Start- und Endpunkt werden als Parameter genutzt, die die Form der Bézierkurve verändern.

Eine Verlängerung oder Verkürzung der Strecke zwischen Startpunkt und dem darauf folgenden Kontrollpunkt, sowie der Strecke zwischen dem Endpunkt und dem davor stehenden Kontrollpunkt führt zu einem größeren oder kleineren Bogen in der Bézierkurve. Diese Eigenschaft wird zum Ausweichen von Hindernissen genutzt. Die zwei Parameter ermöglichen zusätzlich eine Kontrolle über die Beschleunigung des Roboters. Für Roboter, die sich auf Rädern fortbewegen lässt sich diese Eigenschaft für Schüsse oder zur Erhaltung der Stabilität, beispielsweise in Kurven, nutzen.

Da die SPL des RoboCups humanoide Roboter nutzt sind Algorithmen, die auf Holonomie setzen, relevanter. In dieser Kategorie gibt es Algorithmen, die beispielsweise Potentialfelder oder Graphen und Graphsuchalgorithmen nutzen. In den folgenden Unterabschnitten werden drei Algorithmen dieser Kategorie vorgestellt.

Alle drei Algorithmen geben einen Richtungsvektor zurück. Dieser Richtungsvektor wird in einem *Steprequest* an die Walking Engine übergeben. Der *Steprequest* wird wie in Abschnitt 2.3 beschrieben beschränkt. Diese Beschränkung kann als rudimentäre Schrittplanung für einen einzelnen Schritt betrachtet werden. Da keiner der folgenden drei Algorithmen die Rotation des Roboters berücksichtigt, wird die Rotation des Balles zum Roboter im *Steprequest* übergeben. Alle Punkte sind in den lokalen Koordinaten des Rumpfs angegeben.

Anschließend werden Experimente, die mit den Algorithmen durchgeführt wurden, beschrieben. Es wurden Experimente in einer abstrakten 2D Simulation und auf der Roboterplattform durchgeführt. Die Ergebnisse der Experimente wurden zusätzlich miteinander verglichen. Die Simulationsergebnisse und deren Vergleich sind in Abschnitt 3.4 zu finden. Die Ergebnisse auf der Roboterplattform und deren Vergleich sind in Abschnitt 3.5 zu finden.

3.1 Der naive Ansatz (Alg. A)

Dieser Algorithmus ist ein einfacher, reaktiver Algorithmus. Mit Hilfe zweier Ultraschallsensoren des Roboters Nao wird eine simple Hindernisvermeidung realisiert. Der Algorithmus ist Teil des NaoTH Frameworks und als solcher in diesem bereits implementiert.

Um den Richtungsvektor für den nächsten Schritt zu ermitteln wird zu erst überprüft, ob die zwei Ultraschallsensoren ein Hindernis messen. Misst nur der rechte Ultraschallsensor ein Hindernis wird $(0, y)$ als Richtungsvektor zurückgegeben. Der Roboter soll demnach nach links ausweichen bis der Weg vor ihm frei wird. Als Richtungsvektor für den Fall, dass nur der linke Ultraschallsensor ein Hindernis misst, wird analog dazu $(0, -y)$ zurückgegeben. Für den Fall, dass beide Ultraschallsensoren ein Hindernis messen tritt der erste oder der zweite Fall in Kraft, abhängig davon ob das linke oder das rechte Hindernis näher ist. Im NaoTH Framework wird der Parameter $y = 40$ genutzt.

Da die Ultraschallsensoren nicht sehr zuverlässig und präzise sind ist dieser Weg ein Hindernis zu vermeiden keine optimale Lösung. Außerdem wird immer nur das unmittelbar bevorstehende Problem gelöst. Das kann dazu führen, dass in eine Richtung ausgewichen wird in der man nur temporär voranschreiten kann, da sich dort ein weiteres Hindernis befindet, welches unter Umständen bereits vorher erkennbar war.

Misst weder der linke noch der rechte Ultraschallsensor ein Hindernis wird als Richtungsvektor der Zielpunkt p, in unserem Fall der Ball, zurückgegeben.

3.1.1 Pseudocode Algorithmus A

Sei p die Position des Ziels, $avoid\_y$ der Ausweichparameter, $isBlocked$ ein boolean und $dist\_left$, $dist\_right$ die gemessene Distanz des rechten und linken Ultraschallsensors zum Hindernis. $isBlocked$ ist $true$, falls einer oder beide Ultraschallsensoren ein Hindernis erfassen.

Algorithm 1 Berechne den Richtungsvektor

```
function COMPUTE_GAIT($p$, $avoid\_y$, $isBlocked$, $dist\_left$, $dist\_right$)
    if $isBlocked$ then
        if $dist\_left < dist\_right$ then
            return $(0, -avoid\_y)$
        else
            return $(0, avoid\_y)$
        end if
    else
        return $p$
    end if
end function
```

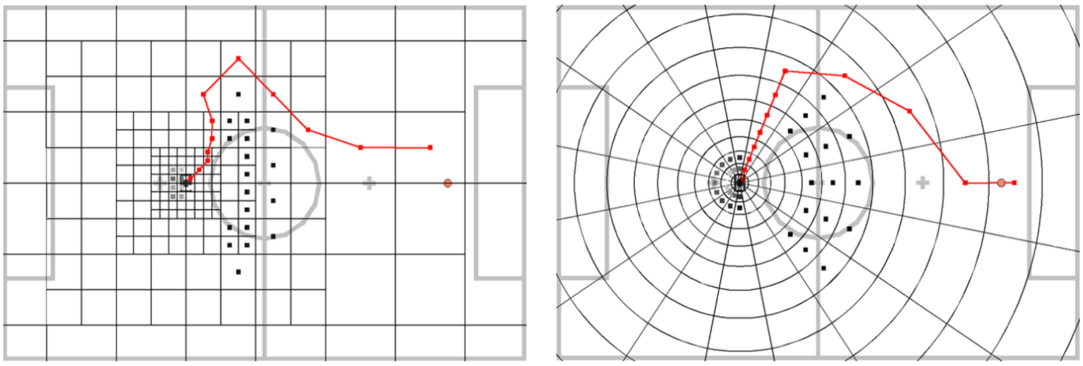


Abbildung 7: *Nicht uniformes Gitter variabler Auflösung links, Gitter in Polarkoordinaten logarithmischer Natur rechts. Bild entnommen aus [9].*

3.2 Globale Pfadplanung in einem Log-Polar Grid (Alg. B)

Nieuwenhuisen, Steffens und Behnke beschreiben in [9] eine globale Pfadplanung mit Hilfe des A*-Graphsuchalgorithmus'. Dabei wurden drei verschiedene Arten von Gittern, die vom Roboter ausgehend über das Spielfeld ausgebreitet werden und sich mit dem Roboter bewegen und rotieren, untersucht. Bei den Gittern handelt es sich um uniforme Gitter, nicht uniforme Gitter mit variabler Auflösung und logarithmische Gitter in Polarkoordinaten.

Der Graph für die Suche des optimalen Pfades wird nicht explizit gebildet sondern ist implizit im Gitter enthalten. In [9] wurden 1000 Planungen durchgeführt um die Ausführungszeiten der verschiedenen Gittertypen zu ermitteln und es wurde zwischen Fällen, bei denen keine Hindernisse, Hindernisse mit den Ultraschallsensoren, Hindernisse mit der Kamera, oder Hindernisse mit den Ultraschallsensoren und der Kamera des Roboters erfasst wurden, unterschieden. Die Werte wurden auf einem Nao V3+ Roboter gemessen.

Die für den A*-Graphsuchalgorithmus genutzte Heuristik ist die euklidische Distanz von einer Zelle zur Zielzelle. Die Kosten von einer Zelle zu seiner Nachbarzelle, also von einem Knoten zu seinem Kindknoten, stellen sich aus der euklidischen Distanz zwischen den Mittelpunkten beider Zellen und den Kosten der Hindernisse, die die Nachbarzelle besetzen, zusammen.

Es wird ein explizites Modell für Hindernisse genutzt. Die Hindernisse bekommen einen Radius, der sich aus einem festen Teil und einem von der Distanz zum Roboter abhängigen, variablen Teil zusammensetzt. Ein Hindernis beeinflusst in diesem Radius die enthaltenen Zellen durch konstante Kosten. Zusätzlich wird eine Sicherheitsmarge, in denen die Kosten linear fallen, zu diesem Radius hinzugefügt. Um die Kosten eines Hindernisses für eine Zelle zu berechnen werden folgende Formeln genutzt:

$$r_f = T \tag{3.2.1}$$

$$r_d = \frac{\sqrt{x^2 + y^2}}{10} \tag{3.2.2}$$

$$a = r_f - r_d \tag{3.2.3}$$

$$r = r_f + r_d \tag{3.2.4}$$

$$s = T_s \cdot r \tag{3.2.5}$$

$$c = \max(\min(1 - ((d - r)/s), 1), 0) \cdot a \tag{3.2.6}$$

r_f ist der feste Radius eines Hindernisses. In unseren Experimenten wurde hier der Parameter $T = 300mm$ gewählt. r_d ist dementsprechend der von der Distanz zum Roboter abhängige Radius. x und y sind die kartesischen Koordinaten des Hindernisses. d ist die Distanz der Zelle, für die die Kosten des Hindernisses berechnet werden. Um die Distanz zur Zelle zu berechnen wird der Mittelpunkt der Zelle gewählt. a sind die konstanten Kosten, r der Radius, in dem die Kosten den konstanten Wert a haben. T_s bestimmt wie schnell die Kosten in der Sicherheitsmarge fallen. Die Kosten eines Hindernisses für eine Zelle bildet c.

Die Unsicherheit, die aus der Wahrnehmung und der Bewegung des Roboters entspringt, wird beim nicht uniformen Gitter durch die variable Auflösung und bei dem Gitter in Polarkoordinaten durch die logarithmische Natur berücksichtigt. Bei allen Gittertypen wird die Unsicherheit der Wahrnehmung der Hindernisse durch die Sicherheitsmarge und den variablen Radius, der abhängig ist von der Distanz zum Roboter, berücksichtigt. Die variable Auflösung des nicht uniformen Gitters und die logarithmische Natur des Gitters in Polarkoordinaten sorgt zusätzlich für die implizite Berücksichtigung der dynamischen Natur eines Fußballspiels, da die Auflösung des Gitters sich mit größerer Distanz zum Roboter verringert. Dadurch werden weiter entfernte Objekte weniger präzise im Gitter repräsentiert.

Durch die Berücksichtigung der Unsicherheit und der dynamischen Natur eines Fußballspiels sind das nicht uniforme Gitter und das Gitter in Polarkoordinaten besonders interessant. Sie sind zu sehen in Abb. 7.

Das Gitter in Polarkoordinaten logarithmischer Natur sorgt zusätzlich für einen geraden Pfad zu Zielen, die vor dem Roboter liegen, unabhängig von der Entfernung des Ziels [9]. Deshalb ist trotz der etwas schlechteren Performanz verglichen zum nicht uniformen Gitter dieser Gittertyp dem nicht uniformen Gitter vorzuziehen.

Die Formel um die entsprechende Zelle in Polarkoordinaten $(r,\, t)$ für gegebene kartesische Koordinaten $(x,\, y)$ zu berechnen ist folgende:

$$\rho = \log_b\left(\left(\frac{\sqrt{x^2+y^2}\,(b-1)}{l}\right)+1\right) \tag{3.2.7}$$

$$\theta = \arctan\left(\frac{y}{x}\right) \tag{3.2.8}$$

$$(r,\, t) = \left(\lfloor\rho\rfloor,\, \lfloor\left(\frac{T}{2\pi}\right)\theta + 0.5\rfloor\right)\ . \tag{3.2.9}$$

Die inverse Operation dazu ist folgende:

$$(x,\, y) = \begin{pmatrix}\cos(\theta)\\ \sin(\theta)\end{pmatrix}\left(\frac{(b^{r+0.5}-1)\cdot l}{b-1}\right)\ . \tag{3.2.10}$$

In [9] wird $b = 1.1789$, eine minimale Zellgröße $l = 100\,mm$ und $T = 16$ Winkelpartitionen empfohlen.

3.2.1 Pseudocode Algorithmus B

Sei p die Position des Ziels und *obstacles* ein Array, das die Hindernisse enthält. Da alles in lokalen Koordinaten des Rumpfes berechnet wird ist die Zelle des Roboters $(0, 0)$.

Die Funktion *get_polar* berechnet die Zelle in Polarkoordinaten für gegebene kartesische Koordinaten, die Funktion *get_cartesian* führt die dazu inverse Operation aus.

Dem A*-Graphsuchalgorithmus werden der Start- und Endpunkt vertauscht übergeben. Dadurch müssen die erhaltenen *waypoints*, also die Mittelpunkte der Zellen für den gebildeten Pfad, nicht invertiert werden um den Pfad vom Roboter zum Ziel zu erhalten.

Algorithm 2 Berechne die Wegpunkte

```
function COMPUTE_WAYPOINTS($p$, $obstacles$)
    start ← GET_POLAR($p$)
    waypoints ← A_STAR_SEARCH($start$, (0, 0), $obstacles$)
    return GET_CARTESIAN($waypoints$)
end function
```

Algorithm 3 Berechne den Richtungsvektor

```
function COMPUTE_GAIT($p$, $obstacles$)
    waypoints ← COMPUTE_WAYPOINTS($p$, $obstacles$)
    return farthest waypoint that is reachable with one step
end function
```

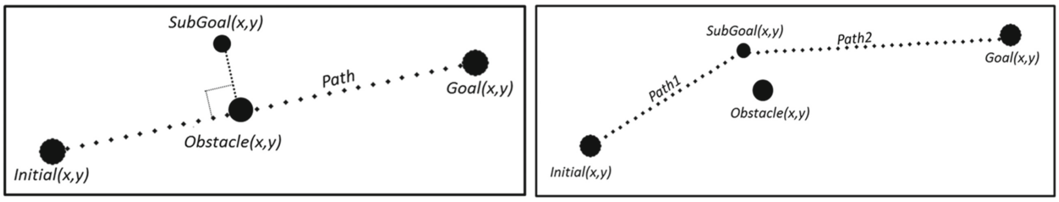


Abbildung 8: *Links die Ausgangstrajektorie, rechts die am Hindernis verschobene Trajektorie. Bild entnommen aus [10].*

3.3 Globale Pfadplanung mit einer rekursiv verschobenen Trajektorie (Alg. C)

Der Algorithmus von Rodríguez, Rojas, Pérez, López, Quintero und Calderón aus [10] berechnet durch rekursiven Aufruf einen hindernisfreien Pfad vom Startpunkt zum Ziel. Dabei wird zu erst eine Trajektorie zwischen Ziel und Start berechnet, die eine gerade Linie zwischen den beiden Punkten bildet. Im nächsten Schritt wird die Trajektorie auf Kollision mit einem Hindernis abgesucht. Existiert eine Kollision zwischen einem Hindernis und der Trajektorie wird ein Zwischenziel berechnet. Der Pfad bildet sich nun aus der Trajektorie vom Start zum Zwischenziel und vom

Zwischenziel zum endgültigen Ziel. Anschließend wird der Algorithmus auf die zwei neuen Trajektorien angewendet, bis der Pfad mit keinem Hindernis kollidiert. Ein Beispiel ist in Abb. 8 zu sehen.

Um im Falle einer Kollision mit einem Hindernis einen möglichst optimalen Pfad zu finden wird ein möglichst optimales Zwischenziel gesucht. Eine Eigenschaft des optimalen Zwischenziels ist, dass der Roboter mit keinem Hindernis kollidiert, wenn er sich an diesem Zwischenziel befindet. Ein Zwischenziel kann sich orthogonal zur Trajektorie in Abständen des Roboterumfangs befinden. Hierfür kann in zwei Richtungen gesucht werden, $+90°$ und $-90°$ zur Trajektorie. Allerdings scheint aus [9] nicht klar hervorzugehen, wie genau zwischen den zwei Kandidaten, die in jedem rekursiven Schritt entstehen können, ausgewählt wird. Folgende Interpretationen sind möglich:

- Es werden insgesamt nur zwei Pfade gebildet. Das heißt, dass in jedem rekursiven Schritt nur ein und nicht zwei Zwischenziele entstehen. Der eine Pfad sucht immer Zwischenziele die $+90$ Grad orthogonal zur Trajektorie liegen, der zweite Pfad sucht Zwischenziele immer -90 Grad orthogonal zur Trajektorie. Am Ende werden beide Pfade miteinander verglichen und der Kürzere wird ausgewählt.

- In jedem rekursiven Schritt wird bei einer Kollision in beide Richtungen ein Zwischenziel gesucht. Es werden zwei temporäre Pfade gebildet, die jeweils eins der beiden entstehenden Zwischenziele enthalten. Das Zwischenziel, welches den kürzeren von beiden Pfaden bildet, wird ausgewählt.

In Abb. 9 ist in blau die zweite Interpretation und in schwarz die erste Interpretation zu sehen. Zahlen markieren die Reihenfolge, in der Zwischenziele zum Pfad dazukommen. Rote Zahlen repräsentieren Punkte, die in beiden möglichen Interpretationen entstehen. Schwarze Zahlen repräsentieren Punkte, die in der ersten Interpretation entstehen und blaue Zahlen repräsentieren Punkte, die in der zweiten Interpretation entstehen. Der Algorithmus versucht mit der zweiten Interpretation zwischen den Hindernissen, die als weiße Kreise dargestellt werden, hindurch zu laufen. Es entsteht jedoch eine Kollision auf der Trajektorie zwischen Punkt 1 und Punkt 2, wodurch Punkt 3 entsteht. Anschließend entsteht eine Kollision auf der Trajektorie zwischen Punkt 3 und Punkt 2, weshalb Punkt 4 entsteht.

Die zweite, mögliche Interpretation sorgt in gewissen Situationen für stark suboptimale Pfade, die sich eventuell bei Näherung zum Endziel auflösen. Das heißt für das Beispiel in Abb. 9, je näher der Roboter zu Punkt 3/5 kommt, desto mehr ähnelt die blaue Trajektorie der Schwarzen. Allerdings kann dieses Verhalten auch zu einem Deadlock führen. Die erste Interpretation hat den Nachteil, dass ein Pfad, der in gewissen Situationen zwischen zwei Hindernissen durchführt und kürzer wäre, ignoriert wird. In dieser Arbeit wurde die zweite Interpretation genutzt.

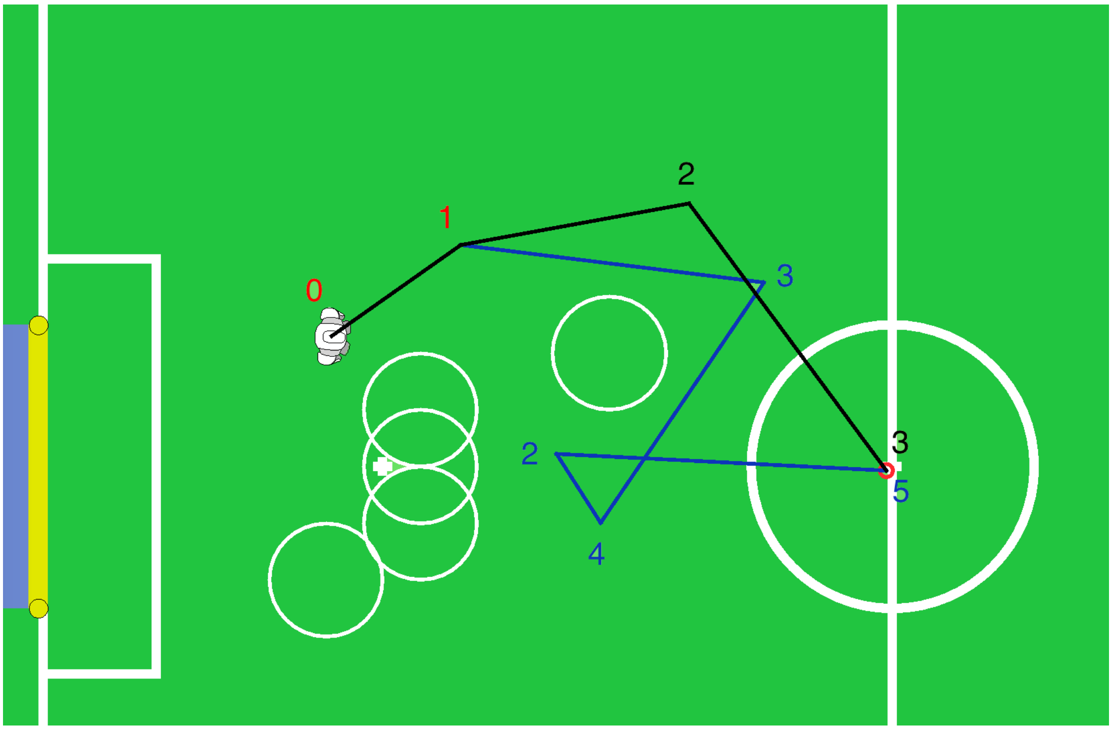


Abbildung 9: *Zwei mögliche Interpretationen des Algorithmus'.*

3.3.1 Pseudocode Algorithmus C

Sei p die Position des Ziels, *obstacles* ein Array, das die Hindernisse enthält, und *robot_radius* der Radius des Roboters.

Algorithm 4 Berechne den Richtungsvektor

```
function COMPUTE_GAIT($p$, $obstacles$)
    trajectory ← COMPUTE_PATH((0, 0), $p$, $obstacles$, 0)
    return $trajectory[0][1] - trajectory[0][0]$
end function
```

Algorithm 5 Berechne Zwischenziel

```
function COMPUTE_SUB_TARGET($start$, $end$, $obstacles$, $sign$, $col_p$)
    direction ← $end - start$
    unit ← $direction$ / $||direction||$
    if sign is +1 then
        orth ← $(unit_y \cdot -1, unit_x \cdot 1)$
    else
        orth ← $(unit_y \cdot 1, unit_x \cdot -1)$
    end if
```

```
        offset ← 0
        sub_target ← $orth * robot\_radius + col_p$
        while sub_target collides with obstacle in $obstacles$ do
            offset ← $offset + robot\_radius$
            sub_target ← $orth * (robot\_radius + offset) + col_p$
        end while
        return sub_target
    end function
```

Algorithm 6 Berechne den Pfad

```
    function COMPUTE_PATH($start$, $end$, $obstacles$, $depth$)
        trajectory ← (start, end)
        if $depth > max\_depth$ then
            return trajectory
        else if collision between trajectory and obstacles then
            $col_p$ ← position of closest obstacle that collides with $trajectory$
            sub1 ← COMPUTE_SUB_TARGET($start$, $end$, $obstacles$, $+1$, $col_p$)
            sub2 ← COMPUTE_SUB_TARGET($start$, $end$, $obstacles$, $-1$, $col_p$)
            traj1 ← COMPUTE_PATH($start$, $sub1$, $obstacles$, $depth + 1$)
            traj2 ← COMPUTE_PATH($start$, $sub2$, $obstacles$, $depth + 1$)
            traj3 ← COMPUTE_PATH($sub1$, $end$, $obstacles$, $depth + 1$)
            traj4 ← COMPUTE_PATH($sub2$, $end$, $obstacles$, $depth + 1$)
            if $traj1 + traj3 < traj2 + traj4$ then
                trajectory ← $traj1 + traj3$
            else
                trajectory ← $traj2 + traj4$
            end if
        end if
        return trajectory
    end function
```

3.4 Simulation und Evaluation

Zur Evaluation der Algorithmen wurden Experimente in einer 2D Simulation durchgeführt. Jedes Experiment wurde separat für jeden Algorithmus simuliert. Ein Experiment besteht aus einem Ziel-, einem Startpunkt und neun zufällig generierten Hindernissen. Jeder Algorithmus berechnet für die aktuelle Roboterposition den Pfad zum Zielpunkt und generiert daraus einen Richtungsvektor. Der Richtungsvektor wird mit der Roboterposition addiert um die neue Roboterposition zu erhalten. Anschließend wird der Pfad mit der neuen Roboterposition neu berechnet, wodurch eine lokale Pfadplanung ensteht.

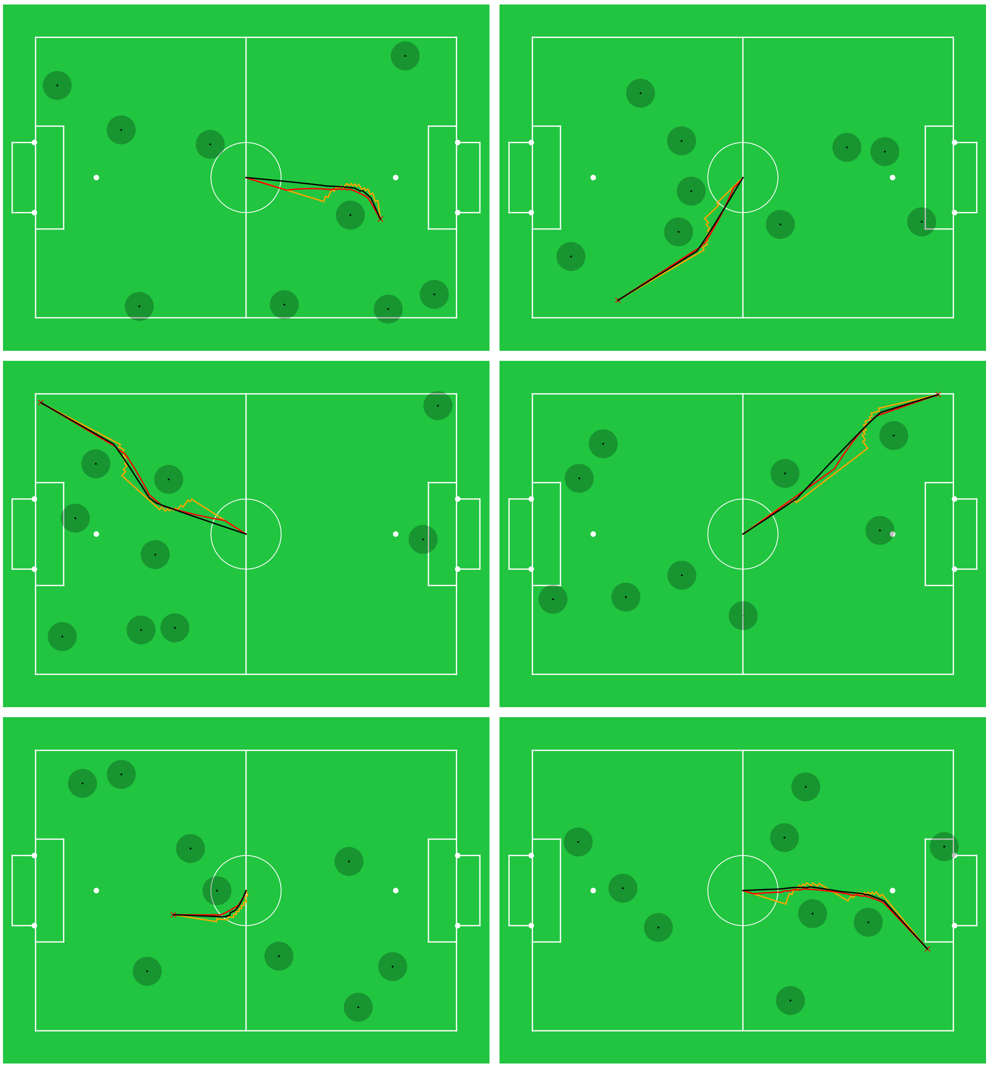


Abbildung 10: *Orange: Algorithmus A. Rot: Algorithmus B. Schwarz: Algorithmus C.*

In Algorithmus A (Abschnitt 3.1) und Algorithmus C (Abschnitt 3.3) besitzen die Hindernisse, ähnlich wie in Algorithmus B, einen Radius. Dieser besteht allerdings nur aus einem statischen Teil. Zusätzlich besitzt auch der Roboter in den beiden Algorithmen A und C einen Radius, damit die Kollisionserkennung realistischer wird. Im Algorithmus B fällt dieser Radius weg, da die Sicherheitsmarge diese Funktion übernimmt.

Für eine Trajektorie zwischen zwei Punkten fand in Algorithmus C eine weiche Kollisionserkennung statt, das heißt, dass ein Punkt mit dem Hindernis kollidiert, wenn die Distanz zwischen dem Punkt und dem Hindernis kleiner oder gleich dem Hindernisradius plus dem Roboterradius abzüglich $5\,cm$ ist. Für die Berechnung der Zwischenziele fand eine harte Kollisionserkennung statt, das heißt, dass keine $5\,cm$ abgezogen wurden. Durch diese Unterscheidung wurde die Anzahl der berechneten Zwischenziele für einen Pfad drastisch reduziert. Außerdem wurde dieser Algorithmus für mehr Geschwindigkeit iterativ implementiert.

In Abb. 10 sind die simulierten Pfade für 6 Experimente abgebildet. Die schwarzen Kreise sind Hindernisse, das rote Kreuz das Ziel und die orange, rote und schwarze Linie der simulierte Pfad des Algorithmus A, B und C in dieser Reihenfolge. Alle drei Algorithmen sind von der Feldmitte aus gestartet.

Es wurden 100.000 Experimente durchgeführt. Anschließend wurde der Median für die Berechnung des Richtungsvektors ermittelt und die Glattheit der Pfade

$$smooth(p) = \frac{\sum \theta_i}{||p||} , \tag{3.4.1}$$

wobei θ_i der Winkel zwischen zwei Punkten auf dem Pfad p ist, berechnet. Die Formel zur Berechnung der Glattheit stammt aus [10].

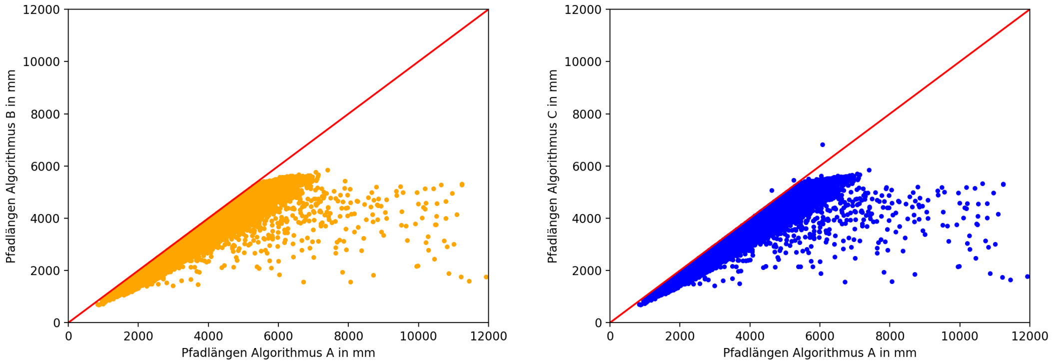


Abbildung 11: *Links die Pfadlängen des Algorithmus B im Verhältnis, rechts die Pfadlängen des Algorithmus C. Jeweils im Verhältnis zum Algorithmus A.*

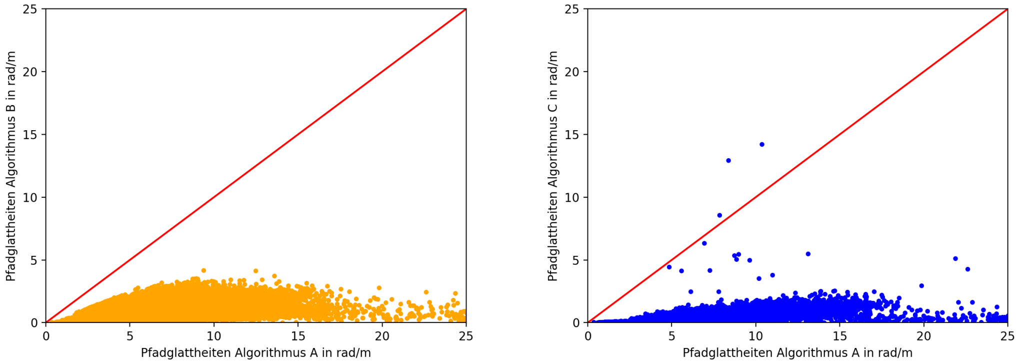


Abbildung 12: *Links die Pfadglattheiten des Algorithmus B im Verhältnis, rechts die Pfadglattheiten des Algorithmus C. Jeweils im Verhältnis zum Algorithmus A.*

Die zwei anderen Algorithmen wurden im Verhältnis zum Algorithmus A betrachtet. Die Ergebnisse sind in Abb. 11 und Abb. 12 zu sehen. Die rote Linie ist eine Orientierungshilfe, die den Algorithmus A visualisiert. Die blauen Punkte sind die Ergebnisse des Algorithmus C, die orangen Punkte des Algorithmus B. Auf der x-Achse wurden die Ergebnisse der Experimente des Algorithmus A, auf der y-Achse der anderen beiden Algorithmen abgebildet. Pfade unterhalb der roten Linie waren kürzer oder glatter als der Algorithmus A, Pfade dadrüber dementsprechend länger oder kantiger.

Es wurden Deadlocks ermittelt und aufgezeichnet. Ein Deadlock tritt auf, wenn der Roboter das Ziel nach 200 Schritten nicht erreicht hat. Die Ergebnisse der Simulation sind in Tabelle 1 zu sehen.

Algorithmus	A	B	C
Median Länge im Verhältnis zu A	1	$0,892$	$0,891$
Median Glattheit im Verhältnis zu A	1	$0,11$	$0,02$
Standardabw. Länge in mm	$981,7$	$905,4$	$897,9$
Standardabw. Glattheit in (rad/m)	$2,8$	$0,5$	$0,27$
Median Berechnungsdauer in s	$0,0001$	$0,028$	$0,001$
Deadlocks	12165 ($12,2\%$)	1 ($0,001\%$)	3986 (4%)

Tabelle 1: *Ergebnisse der Simulation.*

Aus Abb. 11 wird ersichtlich, dass Algorithmus B und C kürzere Pfade als Algorithmus A produzieren. Der Median der Pfadlänge im Verhältnis beträgt $0,892$ für den Algorithmus B und $0,891$ für den Algorithmus C. Daraus wird erkennbar, dass beide Algorithmen sich ähnlich in Bezug auf die Pfadlänge im Verhältnis zum Algorithmus A verhalten und in der Regel Pfade produzieren, die ungefähr 10% kürzer sind.

Mit der Glattheit der Pfade verhält es sich ähnlich, wie in Abb. 12 zu sehen. Während die Pfadlängen sehr nah an der roten Linie liegen bildet sich eine deutliche Lücke bei den Pfadglattheiten. Der Median der Pfadglattheit spiegelt diese Tatsache wider mit dem Wert $0,11$ für Algorithmus B und $0,02$ für Algorithmus C. Der Algorithmus B ist im Schnitt ungefähr 89% glatter, der Algorithmus C hingegen erstaunliche 98%. Im Algorithmus C gibt es allerdings einige Ausreißer, die sich in der Standardabweichung bemerkbar machen. Algorithmus B produziert eine Standardabweichung von $0,5$ und Algorithmus C eine Standardabweichung von $0,27$.

Da die Simulation auf einem deutlich leistungsstärkeren Computer verglichen zur Nao Roboterplattform durchgeführt wurde kann keine Aussage zu der Relevanz der erfassten Berechnungszeiten getroffen werden. Diese Werte sind außerdem implementationsabhängig. Eine zuverlässige Aussage kann erst durch Experimente auf der Roboterplattform selbst getätigt werden, da die Berechnungszeiten dort eine authentische Aussage für die Ausführung der Algorithmen auf dieser Plattform darstellen. Es ist trotzdem interessant anzumerken, dass der Algorithmus C, der für die Simulation iterativ implementiert wurde, während der Simulation 28 mal schneller war als der Algorithmus B. Dass der Algorithmus A eine deutlich bessere Performanz aufgrund seiner simpleren Natur besitzt war zu erwarten und wurde durch die gemessene Zeit bestätigt.

Die Anzahl der Deadlocks beträgt für den Algorithmus B $0,001\%$, für den Algorithmus C 4% und für den Algorithmus A $12,2\%$. Der Grund für die hohe Anzahl an Deadlocks im Algorithmus C ist die Anfälligkeit der rekursiven Verschiebung der Trajektorie gegenüber Symmetrien und die Probleme der gewählten Interpretation, die in Abschnitt 3.3 beschrieben wurden. Der Algorithmus A hingegen bleibt hängen, wenn sich in der Richtung in die ausgewichen wird ein weiteres Hindernis befindet. Der Algorithmus B gerät viel seltener in einen Deadlock durch die Natur des logarithmischen Gitters in Polarkoordinaten, allerdings ist auch dieser Algorithmus anfällig für Symmetrien. Ein Beispiel ist in Abb. 13 zu sehen. Das Verhalten des Algorithmus B und des Algorithmus C in dieser Hinsicht war zu erwarten, da keine Filterung zwischen den berechneten Pfaden geschieht.

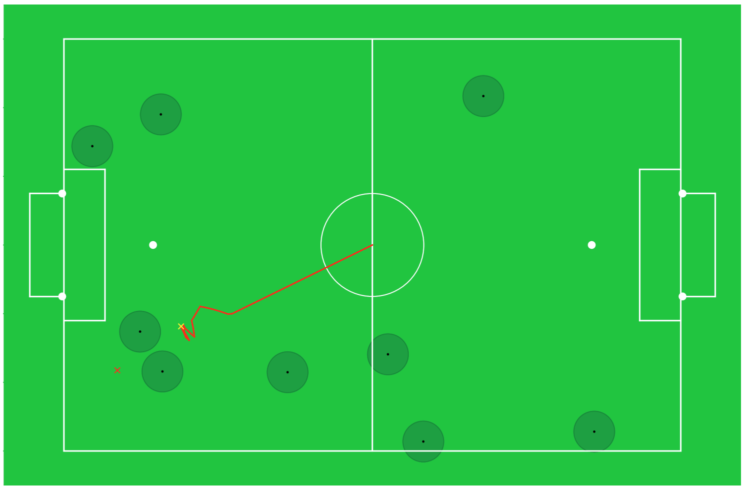


Abbildung 13: *Beispiel für einen Deadlock im Algorithmus B.*

Es wurden weitere 6.000 Experimente durchgeführt, wobei die neun zufällig generierten Hindernisse ein eigenes, zufällig generiertes Ziel besitzen, auf dass sie geradewegs zu laufen. Durch eine fehlende, authentische Kollisionssimulation zwischen Hindernissen oder mit dem Ziel und des willkürlich gewählten Ziels der Hindernisse sind die Ergebnisse dieser Experimente mit Vorsicht zu betrachten.

In Abb. 14 sind analog zu Abb. 11 die Pfadlängen im Verhältnis, in Abb. 15 analog zu Abb. 12 die Pfadglattheiten im Verhältnis zum Algorithmus A zu sehen. Die Ergebnisse der statistischen Auswertung sind in Tabelle 2 zu sehen. Der Algorithmus B hat sich verglichen zum Algorithmus C deutlich weniger verschlechtert. Ein Grund dafür ist der Folgende. Im Algorithmus B existiert kein Mechanismus, der Pfade durch Hindernisse verhindert. Es wird lediglich mit Hilfe einer Kostenfunktion bestimmt, welche Zelle des logarithmischen Gitters in Polarkoordinaten in den Pfad aufgenommen wird. Im Algorithmus C wird dies implizit verhindert. Durch die Kontrolle der Trajekotrien zwischen Startpunkt, Endpunkt und allen Zwischenzielen auf Kollision mit einem Hindernis und der dadurch resultierenden Verschiebung der Trajektorie mit Hilfe eines neuen Zwischenziels entsteht ein Pfad, der durch keine Hindernisse führt - vorausgesetzt das maximale Rekursions- oder Iterationslimit wird nicht überschritten. Dieses Verhalten ist in Abb. 16 zu sehen. Die blaue Linie ist die Trajektorie des Hindernisses, welche im Algorithmus C den Roboter immer weiter weg vom Ziel drängt.

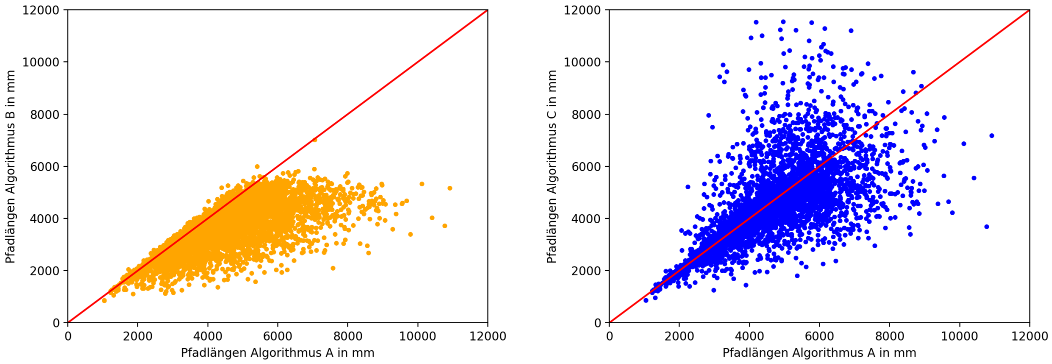


Abbildung 14: *Links die Pfadlängen des Algorithmus B im Verhältnis, rechts die Pfadlängen des Algorithmus C im Verhältnis zum Algorithmus A, jeweils mit simulierten Hindernisse.*

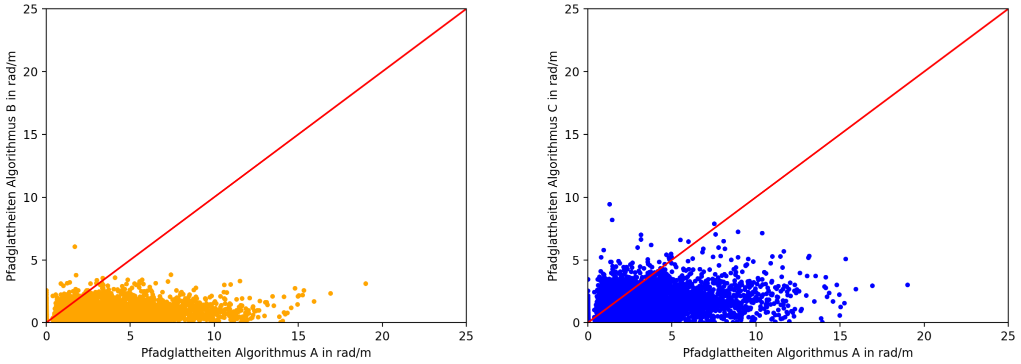


Abbildung 15: *Links die Pfadglattheiten des Algorithmus B im Verhältnis, rechts die Pfadglattheiten des Algorithmus C im Verhältnis zum Algorithmus A, jeweils mit simulierten Hindernisse.*

Algorithmus	A	B	C
Median Länge im Verhältnis zu A	1	$0,9$	$0,98$
Median Glattheit im Verhältnis zu A	1	$0,389$	$0,751$
Standardabw. Länge in mm	$1351,3$	$910,9$	$1448,7$
Standardabw. Glattheit in (rad/m)	$2,66$	$1,05$	$0,56$
Median Berechnungsdauer in s	$6,4 \cdot 10^{-5}$	$0,017$	$0,0002$
Deadlocks	123 (2%)	0 (0%)	11 (0,2%)

Tabelle 2: *Ergebnisse mit simulierten Hindernissen.*

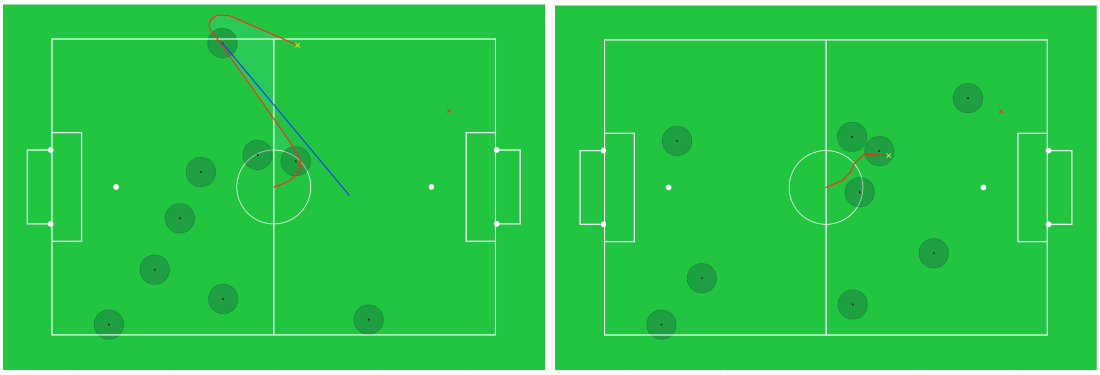


Abbildung 16: *Links wird Algorithmus C vom Hindernis gejagt. Die blaue Linie ist die Trajektorie des jagenden Hindernisses. Rechts wird Algorithmus B nicht gejagt, läuft aber durch Hindernisse.*

Wie zu erwarten haben sich die Deadlocks für Algorithmus A und C verringert. Der Grund wieso Deadlocks trotzdem auftreten ist der, dass die Hindernisse nicht endlos umherwandern, sondern nur bis zu ihrem vorgegebenen Ziel. Die Simulation der Hindernisse hat lediglich einen Teil der potenziellen Deadlocks aufgelöst. Probleme mit Symmetrien bestehen weiterhin.

3.5 Experimente auf der Roboterplattform

Das Verhalten der vorgestellten Pfadplanungsalgorithmen wurde durch die abstrakte 2D Simulation bereits untersucht. Um das Verhalten der Pfadplanungsalgorithmen auf dem Roboter zu testen wurden die Algorithmen B und C für den Roboter implementiert. Algorithmus A war im NaoTH Framework bereits enthalten.

Ein wesentlicher Unterschied dieser Experimente verglichen zur Simulation ist der Einfluss der Unsicherheiten in der Wahrnehmung, insbesondere in der Ballerkennung, und der Ausführung der Bewegungen. In diesen Experimenten untersuchen wir den Einfluss dieser Unsicherheiten. Ein weiteres Ziel ist es zu untersuchen, welchen Einfluss die Algorithmen auf die Fortbewegung des Roboters haben, da

Abbildung 17: *Drei der zehn durch das OptiTrack Trackingsystem genutzten Kameras.*

Algorithmus B und C nicht als lokale Pfadplanung konzipiert sind und es somit beispielsweise keine Filterung der Pfade gibt.

Für die Durchführung der Experimente ist eine verlässliche Hinderniserkennung notwendig. Im aktuellen NaoTH Framework existiert noch kein Algorithmus, der dies bewerkstelligt. Aus diesem Grund wurde für die Durchführung der Experimente auf dem Roboter Nao das OptiTrack Trackingsystem eingesetzt. Mit diesem Trackingsystem werden Objekte mit Markern bestückt, die dann von der OptiTrack Software detektiert werden und dort als zusammenhängend markiert werden können. Eine solche zusammenhängende Gruppe von Markern wird im Folgenden Trackable genannt und repräsentiert ein Objekt. So kann die Position und Rotation der Objekte in globalen Koordinaten sehr genau bestimmt werden. In Abb. 17 sind drei der zehn vom Trackingsystem genutzten Kameras zu sehen.

Ein Trackable besitzt verschiedene Einstellungen, wie beispielsweise einen Namen. Während der Experimente erhält der Roboter per Broadcast eine Liste von Trackables mit ihren entsprechenden Einstellungen. Mit Hilfe der Namen kann der Roboter verschiedene Objekte identifizieren.

Auf diese Weise werden Hindernisse und der Roboter selbst markiert. Dadurch erhält der Roboter Kenntnis von der eigenen Position und Rotation als auch der Position und Rotation der Hindernisse. Die Marker, die die Position des Roboters erfassen, sitzen auf dem Kopf des Roboters. Mit Hilfe der kinematischen Kette wird daraus die globale Position des Roboters auf dem Feld berechnet. Mit der Nutzung des Trackingsystems werden Störungen in der Selbstlokalisierung und der Wahrnehmung der Hindernisse stark reduziert. In Abb. 18 ist die OptiTrack Software an einem Beispiel veranschaulicht.

Ein Experiment besitzt einen Startpunkt und zwei Hindernisse. Als Zielpunkt eines Experiments wird die vom Roboter wahrgenommene Position des offiziellen Balls der SPL genutzt. Der Roboter wird am Startpunkt gestartet und läuft in Richtung des Zielpunktes. Dieser Durchlauf wird für jeden der getesteten Pfadplanungsalgorithmen separat wiederholt. Dabei werden auf dem Roboter folgende

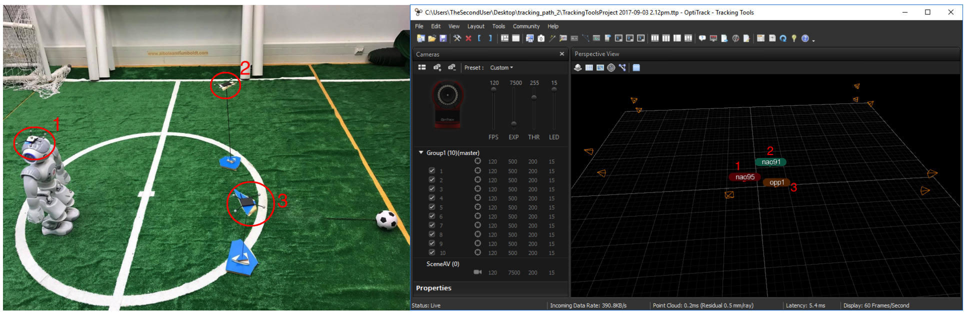


Abbildung 18: *Die OptiTrack Software an einem Beispiel veranschaulicht. Links die angebrachten Markierungen am Roboter und den Hindernissen. Rechts detektierte Trackables.*

Daten aufgenommen:

- Die Position des Roboters, die vom OptiTrack System an den Roboter gesendet wird,
- die Positionen der Hindernisse, die vom OptiTrack System an den Roboter gesendet werden,
- die relative Position des Balls zum Roboter aus seiner eigenen Wahrnehmung,
- die Berechnungszeiten der Algorithmen,
- und die angesteuerten Schritte, die, wie im Abschnitt 2.3 beschrieben, beschränkt und anschließend ausgeführt werden.

Es wurden zwei Experimentalreihen durchgeführt. Bei der ersten Reihe bestehen die Hindernisse, wie in Abb. 18 zu sehen, aus einem dünnen Stock, der in einem Schaumstoffboden steckt. Das erlaubt dem Roboter den Ball auf eine größere Entfernung stabil zu detektieren, was eine größere Freiheit bei der Positionierung der Hindernisse ermöglicht. Die Hindernisvermeidung von Algorithmus A basiert auf den Ultraschallsensoren, die mit diesen Hindernissen nicht funktionieren. Aus diesem Grund wurden jeweils nur die Algorithmen B und C getestet. Es wurden insgesamt zehn Experimente in dieser Reihe durchgeführt.

Bei der zweiten Experimentalreihe wurden reale Nao Roboter als Hindernisse genutzt. Die Durchführung dieser Experimente ist aufwendiger, da die Sichtbarkeit des Balls durch die Hindernisse stark eingeschränkt wird, was die Freiheit der Positionierung der Hindernisse deutlich einschränkt. Es wurden insgesamt drei Experimente für jeden der drei Algorithmen A, B und C durchgeführt.

In Abb. 19 sind ein Beispielexperiment und die Berechnungszeiten für Algorithmus A, B und C in dieser Reihenfolge zu sehen. Dieses Beispielexperiment stammt aus der zweiten Experimentalreihe. Blau ist der angesteuerte Pfad, Rot der resultierende, globale Pfad, der tatsächlich abgelaufen wurde. Der orange Kreis ist die Position des Balls und das blaue Kreuz ist die Startposition des Roboters.

Der Pfad, der sich durch die angesteuerten Schritte bildet, ist an dieser Stelle nur zur Veranschaulichung auf die Roboterposition verschoben. Dieser Pfad entsteht in den lokalen Koordinaten des Roboters.

Die globale Position des Roboters wird in die lokalen Koordinaten des Roboterrumpfes, die in Abschnitt 2.4 beschrieben wurden, umgerechnet. Da sich der Oberkörper des Roboters bewegt während er läuft hat diese Bewegung auch Einfluss auf den globalen, resultierenden Pfad, wodurch eine Schlangenlinie entsteht. Um diese lokale Bewegung herauszufiltern wird der globale Pfad abgetastet, wobei jeweils nur die Positionen am Anfang beziehungsweise Ende eines Schrittes genommen werden.

Insgesamt konnte beobachtet werden, dass der Algorithmus B sehr zuverlässig und stabil funktioniert hat. Bei Algorithmus C traten oft Oszillationen auf. Es traten außerdem Deadlocks auf, so dass einige Durchläufe abgebrochen werden mussten, ohne dass der Roboter das Ziel erreicht hat. Ein Beispiel dafür ist in Abb. 20 zu sehen. In der zweiten Experimentalreihe mussten die Durchläufe für Algorithmus A mehrmals wiederholt werden, da die Ultraschallsensoren nicht zuverlässig funktioniert haben, so dass eine Kollision auftrat und der Durchlauf abgebrochen werden musste. Bei den erfolgreichen Durchläufen wich der Roboter mit dem Algorithmus A den Hindernissen deutlich knapper aus als Algorithmus B und C. Dieser Umstand ist in den Beispielexperimenten aus Abb. 19 erkennbar.

In Tabelle 3 sind die Ergebnisse der ersten Experimentalreihe zu sehen. In Tabelle 4 sind die Ergebnisse der zweiten Experimentalreihe zu sehen.

Algorithmus B produziert kürzere und glattere Pfade als Algorithmus C. Diese Werte widersprechen den Ergebnissen der Simulation aus Abschnitt 3.4. Ein Grund dafür kann die Oszillation der Pfade im Algorithmus C sein.

Zusätzlich liefert Algorithmus A, im Gegensatz zu den Ergebnissen der Simulation aus Abschnitt 3.4, bemerkenswerterweise insgesamt bessere Werte bei der Pfadglattheit und der Pfadlänge als die Algorithmen B und C. Ein Grund dafür ist die bereits erwähnte Eigenschaft des Algorithmus A den Hindernissen deutlich knapper als die Algorithmen B und C auszuweichen. Außerdem scheint der Algorithmus A deutlich robuster bezüglich der Unsicherheiten in der Wahrnehmung und der Ausführung zu sein, was vermutlich an der lokalen Natur des Algorithmus liegt, da kein globaler Pfad im Vorraus geplant wird. Ein weiterer Grund für die Unterschiede in den Ergebnissen könnte in der Diskrepanz zwischen der simulierten Ultraschallwahrnehmung und den realen Ultraschallsensoren liegen.

Bei Algorithmus B scheint die Berechnungszeit von der Entfernung zum Ziel und der Anzahl der Hindernisse zwischen Roboter und Ziel abhängig zu sein. Das bestätigt die Beobachtungen aus [9]. Die Algorithmen A und C haben erwartungsgemäß nahezu konstante Berechnungszeiten. Obwohl die Berechnungszeit bei dem Algorithmus B deutlich höher ist als im Fall der Algorithmen A und C, um etwa den Faktor 6, liegt es dennoch im Rahmen des Akzeptablen. Es besteht jedoch Optimierungspotenzial.

Alle drei Algorithmen sind auf dem Roboter durchaus ausführbar. Der Algorithmus C scheint besonders von den Unsicherheiten in der Wahrnehmung beeinflusst zu werden. Für einen effektiven Einsatz könnte an dieser Stelle eine zusätzliche Filterung helfen. Im Gegensatz dazu liefert der Algorithmus B auch in der aktuellen Form stets einen robusten, abgelaufenen Pfad. Beide Algorithmen B und C weichen den Hindernissen zuverlässiger aus als Algorithmus A. Dabei muss man berücksichtigen, dass die Hindernisvermeidung in Algorithmus A lediglich auf den Ultraschallsensoren basiert.

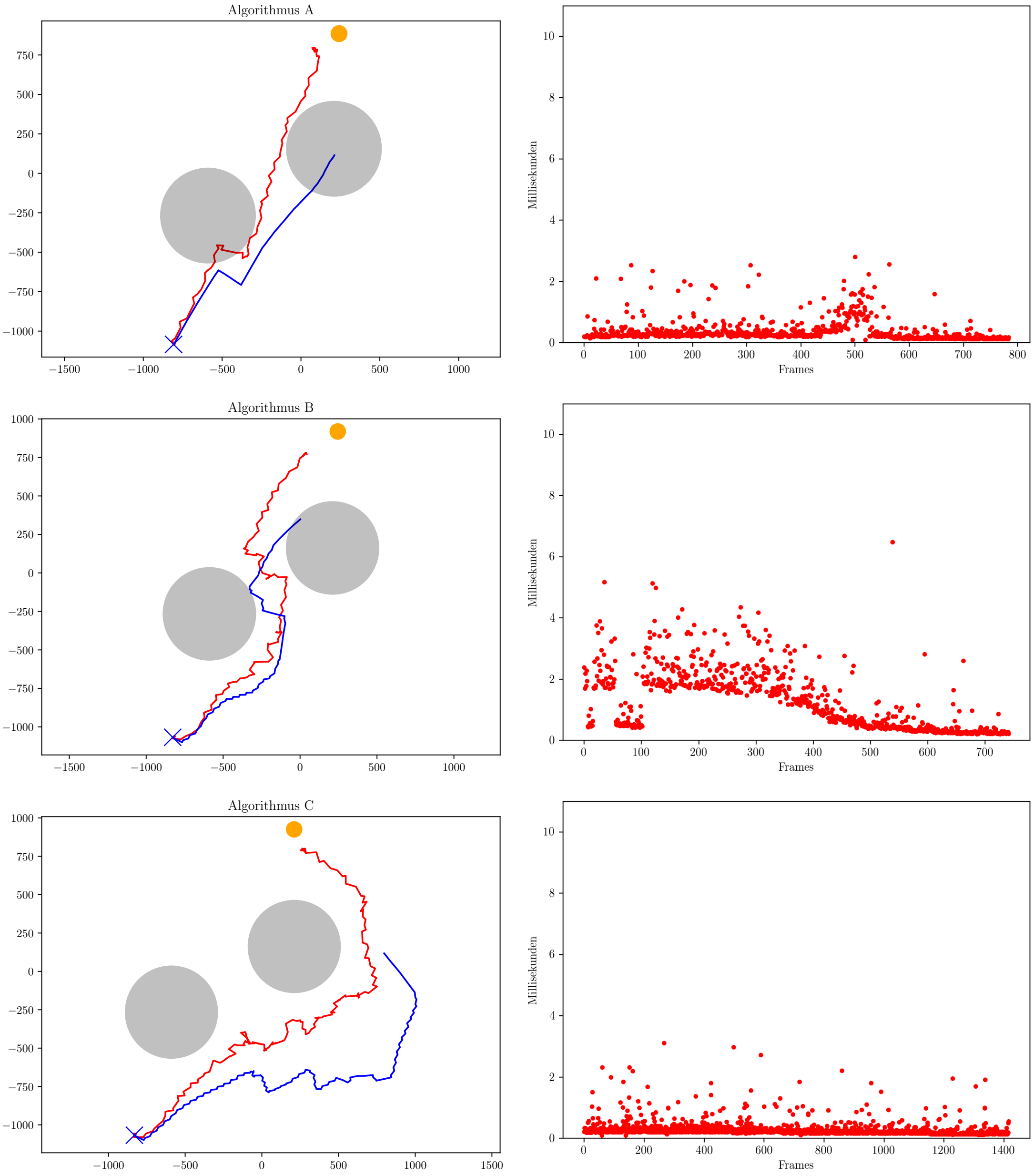


Abbildung 19: *Links das Experiment, rot: abgelaufener Pfad, blau: angesteuerter Pfad, oranger Kreis: der Ball, schwarze Kreise: Hindernisse, blaues Kreuz: Startposition. Rechts die Berechnungszeiten während des Experiments in ms.*

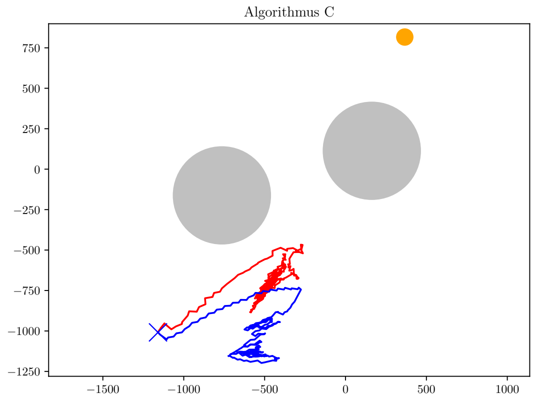


Abbildung 20: *Ein Deadlock im Algorithmus C. Rot: abgelaufener Pfad, blau: angesteuerter Pfad, oranger Kreis: der Ball, schwarze Kreise: Hindernisse, blaues Kreuz: Startposition.*

Algorithmus	A	B	C
Median Länge abgelaufener Pfad in mm	–	$3685,9$	$5202,36$
Median Länge angesteuerter Pfad in mm	–	$2493,38$	$3240,83$
Median Glattheit abgelaufener Pfad n (rad/m)	–	$50,43$	$55,8$
Median Glattheit angesteuerter Pfad in (rad/m)	–	$113,51$	$106,39$
Standardabweichung Länge abgelaufener Pfad in mm	–	$767,73$	1044.89
Standardabweichung Länge angesteuerter Pfad in mm	–	$765,70$	$1089,94$
Standardabweichung Glattheit abgelaufener Pfad in (rad/m)	–	$8,24$	$8,14$
Standardabweichung Glattheit angesteuerter Pfad in (rad/m)	–	$13,9$	$10,37$
Max. Berechnungszeiten in ms	–	$6,05$	$5,229$
Median Berechnungszeiten in ms	–	$1,74$	$0,24$
Standardabweichung Berechnungszeiten in ms	–	$1,04$	$0,39$

Tabelle 3: *Ergebnisse der ersten Experimentalreihe.*

Algorithmus	A	B	C
Median Länge abgelaufener Pfad in mm	$2870,38$	$3296,37$	$4751,47$
Median Länge angesteuerter Pfad in mm	$760,46$	$1824,62$	$3780,05$
Median Glattheit abgelaufener Pfad in (rad/m)	$42,75$	$44,72$	$50,86$
Median Glattheit angesteuerter Pfad in (rad/m)	$181,55$	$107,79$	$96,87$
Standardabweichung Länge abgelaufener Pfad in mm	$328,97$	$471,44$	$143,82$
Standardabweichung Länge angesteuerter Pfad in mm	$478,93$	$917,03$	$265,08$
Standardabweichung Glattheit abgelaufener Pfad in (rad/m)	$11,97$	$1,44$	$5,25$
Standardabweichung Glattheit angesteuerter Pfad in (rad/m)	$48,43$	$32,43$	$1,09$
Max. Berechnungszeiten in ms	$5,244$	$11,375$	$9,251$
Median Berechnungszeiten in ms	$0,261$	$1,392$	$0,224$
Standardabweichung Berechnungszeiten in ms	$0,52$	$1,02$	$0,4$

Tabelle 4: *Ergebnisse der zweiten Experimentalreihe.*

4 Ballkontrolle

In diesem Abschnitt wird die Ballkontrolle behandelt, die die Entscheidung wann geschossen werden soll, den Übergang vom Ballanlauf, die Bewegung am Ball, die Abfolge eines Schusses und die Berechnung der Schußtrajektorie beinhaltet.

Es wurden keine systematischen, empirischen Experimente für die Ballkontrolle durchgeführt. Allerdings kam sie in der Form, wie sie in diesem Abschnitt beschrieben wird, während der GermanOpen und der Weltmeisterschaft 2017 in Japan zum Einsatz. Dabei wurden gute Ergebnisse erzielt. Vor allem die Abfolge eines Schusses profitierte in besonderem Maße von der expliziten Schrittkontrolle, die im Zuge dieser Arbeit eingeführt wurde.

4.1 Übergang vom Ballanlauf zur Ballkontrolle

Im NaoTH Framework wird das Verhalten des Roboters mit Hilfe der Extensible Agent Behavior Specification Language (XABSL) implementiert. XABSL ist eine Sprache zur Implementation des Verhaltens für autonome Roboter mit Hilfe einer hiararchischen, endlichen Zustandsmaschine [3]. Durch die Nutzung von XABSL wird eine einfache Art und Weise das Verhalten des Roboters zu beschreiben ermöglicht, die sich allerdings durch ihre Einfachheit beschränkend auswirkt, wenn komplexes Verhalten implementiert werden soll.

Der Übergang zwischen Ballanlauf und Ballkontrolle findet in der gegebenen Implementation des NaoTH Frameworks lediglich abhängig von der geometrischen Entfernung des Roboters zum Ball statt. Es sei $p = (p_x, p_y)$ die Position des Balls zum Roboter im Koordinatensystem des Rumpfes. Die Bedingung für den Übergang, das heißt für den Abschluß des Ballanlaufs, ist folgende:

$$p_x < T_x \,, \tag{4.1.1}$$

$$|p_y| < T_y \,. \tag{4.1.2}$$

In unseren Beobachtungen haben die Schwellen $T_x = 300$ und $T_y = 50$ gut funktioniert. Die angestrebte Distanz zum Ball während des Ballanlaufs ist deutlich kleiner als die Bedingung für den Übergang voraussetzt. Das soll verhindern, dass der Roboter den Ball überläuft, was durch die Unsicherheit in der Wahrnehmung resultieren kann.

4.2 Bewegung am Ball

Befindet sich der Roboter nicht mehr im Ballanlauf sondern in der Ballkontrolle ist das Ziel des Roboters sich in die angestrebte Schußposition zu bewegen. Der Richtungsvektor für den nächsten Schritt wird im lokalen Koordinatensystem des linken oder des rechten Fußes berechnet, abhängig davon mit welchem Fuß geschossen werden soll.

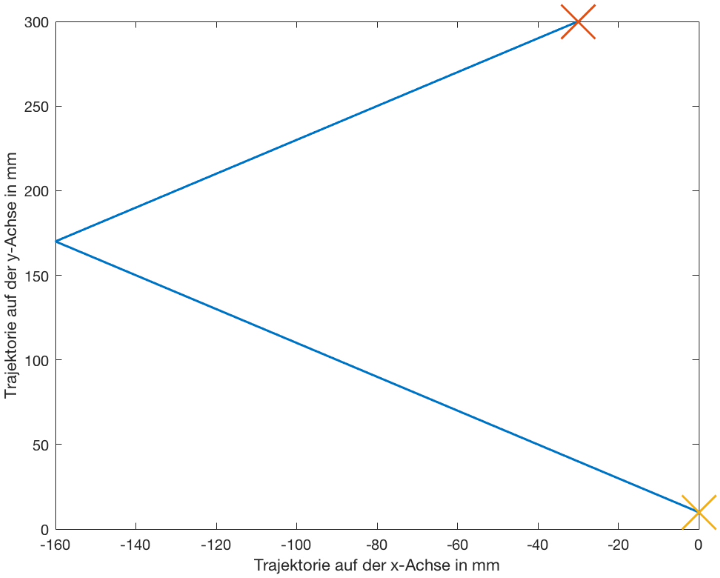


Abbildung 21: *Beispieltrajektorie mit der Ballposition* $p = (250, 300)$*, den Parametern* $distance = 150$*,* $rad_b = 50$ *die aktuelle Ballgröße der SPL und* $offset_y = 0$ *zum Ausführen eines Vorwärtskicks. Das rote Kreuz ist die Startposition des Roboters, das gelbe Kreuz die Endposition des Roboters.*

Die Position des Balls ist mit p gegeben. $offset_y$ und $distance$ sind Parameter, die übergeben werden. rad_b ist der Radius des Balls. Wird das lokale Koordinatensystem des linken Fußes genutzt, wird der Richtungsvektor wie folgt berechnet:

$$t_x = p_x - |\, p_y - offset_y \,| - distance - rad_b \tag{4.2.1}$$
$$t_y = p_y - offset_y \ . \tag{4.2.2}$$

Wird das lokale Koordinatensystem des rechten Fußes genutzt, wird der Richtungsvektor wie folgt berechnet:

$$t_x = p_x - |\, p_y + offset_y \,| - distance - rad_b \tag{4.2.3}$$
$$t_y = p_y + offset_y \ . \tag{4.2.4}$$

Durch $offset_y$ kann eine in der y-Achse des Roboters versetzte Position angestrebt werden, die einen Schuß zur Seite ermöglicht. $distance$ kontrolliert die Entfernung des Fußes zum Ball in der x-Achse des Roboters. In unseren Beobachtungen haben die Werte $distance = 150$ und $offset_y = 0$ für Vorwärtsschüsse und $distance = 140$ und $offset_y = -40$ für Seitwärtsschüsse gut funktioniert.

$|\, p_y \pm offset_y \,|$ sorgt dafür, dass sich der Roboter seitlich vom Ball wegbewegt, wenn dieser weit genug entfernt in der y-Achse des Roboters liegt. Das heißt, wenn der Ball weit genug links oder rechts vom Roboter liegt, bewegt der Roboter sich rückwärts und gleichzeitig seitwärts. Dadurch soll gewährleistet werden, dass der

Abbildung 22: *Standardabfolge eines Schusses.*

Roboter weiterhin gute Chancen hat in Ballbesitz zu kommen, wenn während der Positionierung zum Schuß der Ball bewegt wird. In Abb. 21 ist die Trajektorie an einem Beispiel veranschaulicht.

Da die Unsicherheit in der Wahrnehmung des Roboters für den Übergang vom Ballanlauf zur Ballkontrolle nicht berücksichtigt wird, kann es passieren, dass der Roboter näher am Ball ist als für einen Vorwärts- oder Seitwärtsschuß angestrebt wird. Ist dies der Fall bewegt der Roboter sich rückwärts zur angestrebten Position.

4.3 Entscheidung zu Schießen

Mit der Entscheidung zu Schießen ist gemeint, ob der Roboter sich in einer guten Position zu Schießen befindet und nicht ob generell geschossen oder welcher Typ Schuß eingesetzt werden soll. Diese Entscheidungen werden im NaoTH Framework wie in [4] beschrieben getroffen.

Die Entscheidung fällt ähnlich zum Übergang vom Ballanlauf zur Ballkontrolle und findet nur auf geometrischer Grundlage statt. Die Position des Balls sei mit $p = (p_x, p_y)$ gegeben. $distance$ und rad_b sind die selben Parameter wie in Abschnitt 4.2. Die Voraussetzung ist wie folgt für das Koordinatensystem im linken Fuß

$$p_x - distance - rad_b < T_x \tag{4.3.1}$$

$$Schusses|p_y| - offset_y < T_y\,, \tag{4.3.2}$$

und wie folgt für das Koordinatensystem im rechten Fuß

$$p_x - distance - rad_b < T_x \tag{4.3.3}$$

$$|p_y| + offset_y < T_y\,. \tag{4.3.4}$$

In unseren Beobachtungen haben die Schwellen $T_x = 20$ und $T_y = 10$ gut funktioniert.

4.4 Abfolge eines Schusses

In der bestehenden Implementation des NaoTH Frameworks war es nicht möglich den Schußfuß nach Ausführung der Bewegung zurückzuziehen. Diese Fähigkeit ist essenziell für Seitwärtskicks, um den Ball nach erfolgreichem Ausführen des

Schusses durch eine Vorwärtsbewegung des Standfußes nicht zu torpedieren. Aus diesem Grund wurde eine Schrittabfolge für Schüße eingeführt, die es ermöglicht den Schußfuß nach dem Schuß erneut zu bewegen. In Abb. 22 ist die Abfolge zu sehen.

Der *Zerostep* sorgt dafür, dass der bewegbare Fuß in der Walking Engine zurückgesetzt wird. Dies ermöglicht eine erneute Bewegung des Schußfußes. Da intern, wie in Abschnitt 2.4 beschrieben, im Koordinatensystem des Standfußes gerechnet wird kann der Fuß mit folgender Translation und Rotation vollständig zurückgezogen werden:

$$t_x = 0\,, \tag{4.4.1}$$

$$t_y = 0\,, \tag{4.4.2}$$

$$\theta = 0\,. \tag{4.4.3}$$

Zusätzlich wirkt sich der *Zerostep* positiv auf die Stabilität des Roboters während des Schusses aus. Indem dieser Schritt das Massezentrum zwischen beide Füße schiebt verlagert der Roboter während des Schusses sein Gewicht nicht komplett nach vorne.

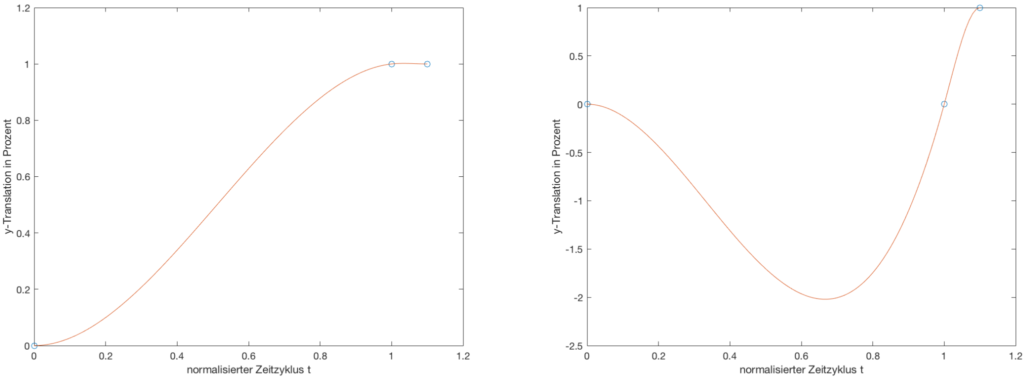


Abbildung 23: *Links die Trajektorie für die Translation an der x-Achse, rechts die Trajektorie für die Translation an der y- Achse.*

4.5 Schußtrajektorie

Die Trajektorie eines Schusses wird nicht wie die eines normalen Schrittes berechnet, sondern mit Hilfe von kubischen Splines. Der erste und zweite Punkt des Splines sind der Start- und Endpunkt des Schusses. Das heißt, der erste Punkt ist der Ausgangspunkt des Fußes vor dem Schuß, der zweite Punkt ist der Punkt an dem der Fuß nach Ausführung des Schusses enden soll. Ein dritter Punkt wird hinter den Zweiten gesetzt um die Beschleunigung kurz vor dem Ball zu erhöhen.

Der im *Steprequest* $(x_r, y_r, \theta_r, c, c_{char}, f, t, s, d, p_{type})$ angegebene Parameter d bestimmt die Koordinaten des dritten Punktes und hat somit einen Einfluss auf die Translation der Trajektorie. Der dritte Punkt befindet sich $3cm$ vom Endpunkt entfernt in der Richtung, in die der Fuß zeigt, rotiert um d Grad.

Bei Vorwärtsschüßen wird $d = 0$, bei Seitwärtsschüßen $d = 90$ oder $d = -90$ verwendet, abhängig davon, ob der Ball nach links oder rechts rollen soll. In Abb. 23 ist die Trajektorie für einen Seitwärtsschuß zu sehen. Die Translation wird an der y-Achse in Prozent angegeben für ein beliebiges (x_r, y_r). Der normalisierte Zeitzyklus t wird ähnlich wie in Abschnitt 2.5 für eine Interpolationsfunktion genutzt, um die Bewegung der Trajektorie auszuführen.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurden zwei Szenarien beschrieben, in denen die Pfadplanung und Schrittkoordination essenziell sind: der Ballanlauf und die Ballkontrolle. Im ersten Szenario, dem Ballanlauf, ist der Roboter so weit entfernt vom Ball, dass er keine Kontrolle ausüben kann. Das heißt er kann den Ball nicht manipulieren ohne vorher näher heranzutreten. Demnach ist das Ziel in diesem Szenario an den Ball anzulaufen um Kontrolle ausüben zu können. Mit Hilfe der Pfadplanung kann die Problemstellung dieses Szenarios gelöst werden. Es wurden hierfür zwei Algorithmen vorgestellt und mit einem naiven Algorithmus, der bereits im NaoTH Framework implementiert war, verglichen. Der Vergleich wurde mit den Ergebnissen einer abstrakten Simulation und der Experimente auf einem realen Roboter unter Laborbedingungen durchgeführt.

Die Ergebnisse der Simulation zeigen, dass die beiden vorgestellten Algorithmen kürzere und glattere Pfade produzieren als der naive Algorithmus. Die vorgestellten Algorithmen sind allerdings als globale Pfadplanungsalgorithmen konzipiert und deshalb anfällig gegenüber Symmetrien, wenn sie in Form einer reaktiven, lokalen Pfadplanung eingesetzt werden. Eine Möglichkeit dieses Problem zu lösen ist die entstehenden Pfade zu filtern. Algorithmus B weist außerdem Probleme in der Berechnung der Kosten auf. Diese sorgen dafür, dass der Roboter unter Umständen durch Hindernisse hindurch laufen will. Es muss untersucht werden ob alternative Ansätze zur Berechnung der Kosten oder Veränderungen der in Abschnitt 3.2 beschriebenen Kostenfunktion dieses Verhalten verhindern können.

Eine weitere Schwierigkeit bildet sich durch die Frage, wie sich der Roboter während der Fortbewegung rotieren soll. Die Autoren der beiden Algorithmen haben diese Fragestellung nicht untersucht. Deshalb wurde während der Experimente auf dem realen Roboter für die angefragte Rotation in jedem Schritt der Winkel zwischen Roboter und Ball gewählt. Dadurch verlangsamt sich die Fortbewegung des Roboters, da, wie in Abschnitt 2.3 beschrieben, die Translation eines angefragten Schrittes durch die angefragte Rotation beschränkt wird.

Die Experimente auf einem realen Roboter widersprechen den Ergebnissen der Simulation. Algorithmus A, welcher im NaoTH Framework bereits enthalten war, produzierte bessere Ergebnisse. Algorithmus C scheint besonders von der Unsicherheit des Roboters beeinflusst zu werden. Algorithmus B hingegen lieferte robuste, abgelaufene Pfade, wies dafür aber eine höhere Berechnungszeit auf. In der Implementation von Algorithmus B steckt Optimierungspotenzial. Es sollte außerdem untersucht werden, wie der Einfluss der Unsicherheit des Roboters auf Algorithmus C gedämpft werden kann. Eine Möglichkeit hierfür liegt in der Filterung der berechneten Pfade.

Damit diese Algorithmen sinnvoll auf einem Roboter im Kontext der SPL genutzt werden können müssen diese Probleme zu erst untersucht und gelöst werden. Zusätzlich muss eine Hinderniserkennung implementiert werden, da diese Funktionalität essenziell ist für die Hindernisvermeidung während der Pfadplanung, im

NaoTH Framework aber nicht enthalten ist.

Das zweite Szenario ist die Ballkontrolle. Das Ziel während dieses Szenarios ist es die Kontrolle über den Ball zu behalten und den Ball in Richtung gegnerisches Tor zu bewegen. Das bedeutet, dass die Ballkontrolle aus folgenden Teilen besteht: die Schußentscheidung, die Abfolge eines Schusses, der Schußtrajektorie, dem Übergang vom Ballanlauf zur Ballkontrolle und der Bewegung am Ball.

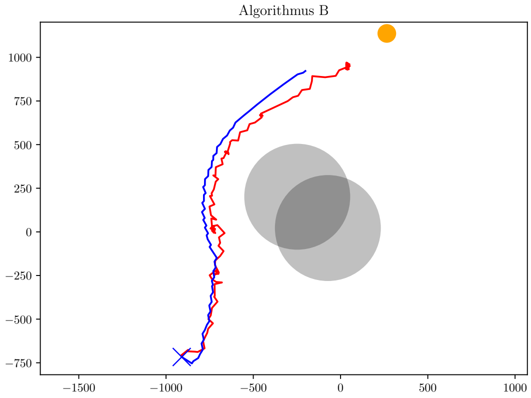
Die Walking Engine des NaoTH Frameworks wurde um eine explizite Schrittkontrolle erweitert. Zu diesem Zweck wurde eine bestehende Schnittstelle zur Anfrage von Schritten überarbeitet. Es wurden drei verschiedene Schritttypen eingeführt, mit deren Hilfe die explizite Schrittkontrolle realisiert wurde. Diese Schrittkontrolle kam während der GermanOpen 2017 und der Weltmeisterschaft 2017 in Japan erfolgreich zum Einsatz. Dort hat sich die explizite Schrittkontrolle besonders positiv auf die Schüsse ausgewirkt. Die Abfolge eines Schusses wurde mit der neuen Schrittkontrolle umstrukturiert. Dadurch wird der Schußfuß nach Ausführung des Schusses zurückgezogen. Eine systematische, experimentelle Untersuchung steht noch aus und sollte in zukünftiger Arbeit nachgeholt werden.

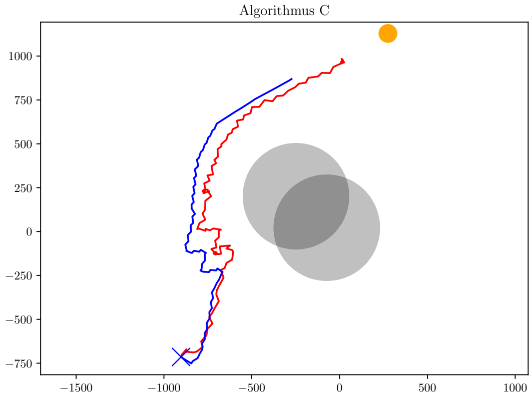
Mit Hilfe der expliziten Schrittkontrolle können präzise Manöver realisiert werden. Ein Beispiel für ein solches Manöver ist ein Rotationsdribbling. Statt einem Seitwärtsschuss rotiert der Roboter in eine Richtung und läuft gleichzeitig vorwärts um den Ball zur Seite zu bewegen. Die Entwicklung verschiedener Manöver ist vielversprechend, da dadurch eine dynamische Spielweise realisiert werden kann.

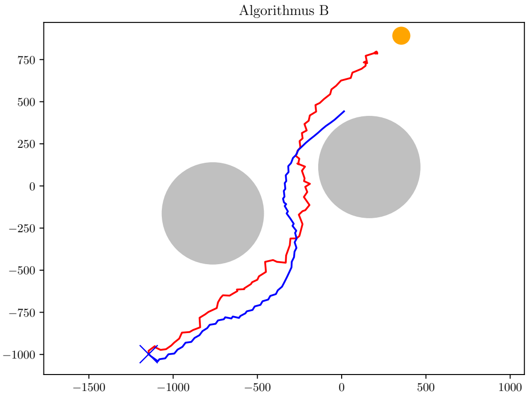
Die Entscheidung, ob der Roboter in einer schußbereiten Position ist oder ob vom Ballanlauf in die Ballkontrolle übergegangen werden soll, wird lediglich auf geometrischer Basis getroffen. Hierfür sollten Möglichkeiten untersucht werden, die auf der Unsicherheit in der Wahrnehmung des Roboters basieren. Dadurch kann beispielsweise verhindert werden, dass sich der Roboter beim Übergang vom Ballanlauf in die Ballkontrolle zu nah am Ball befinden kann. Ist dies nämlich der Fall läuft der Roboter während der Ballkontrolle erst rückwärts bevor er schießt.

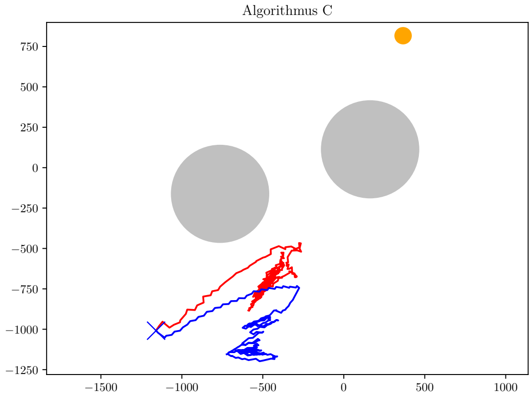
Appendix

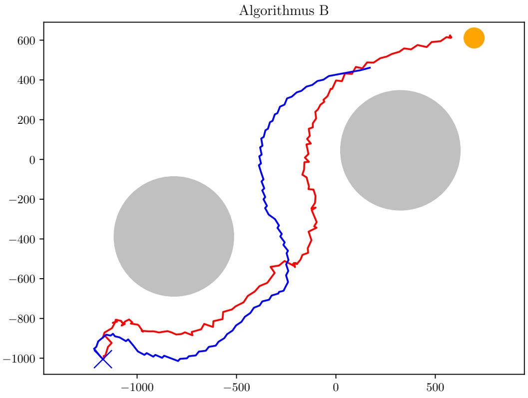
1.1 Erste Experimentalreihe

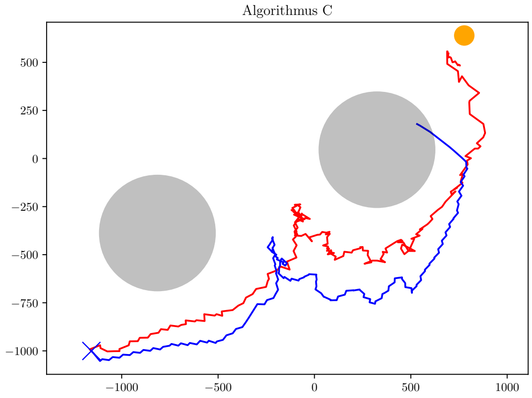


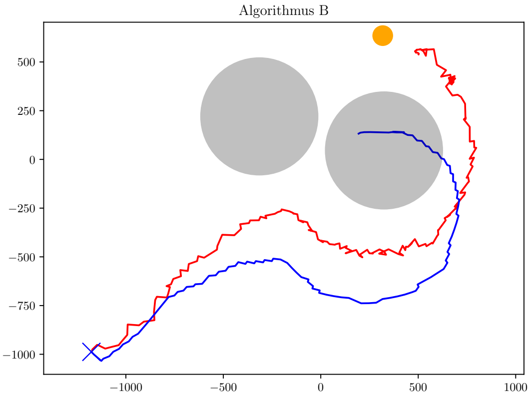


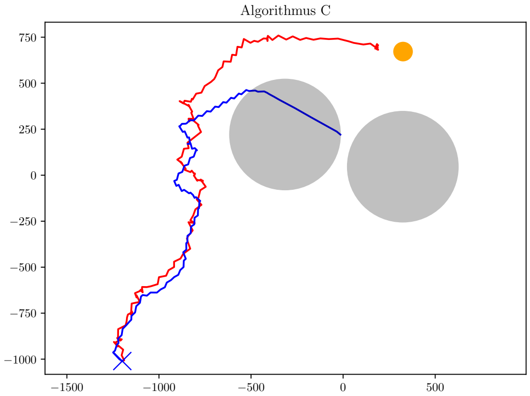


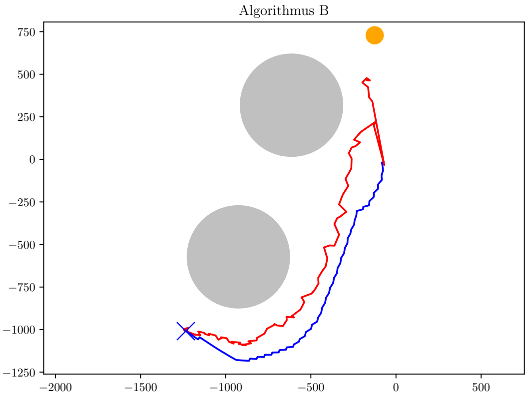


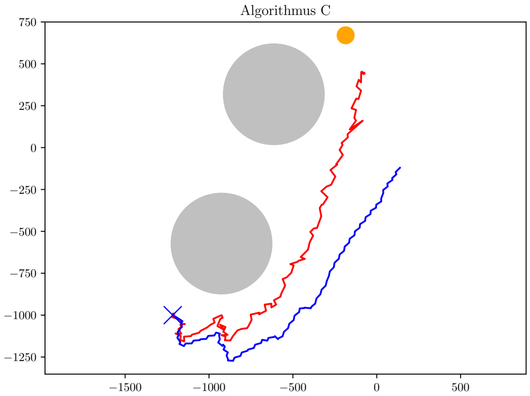


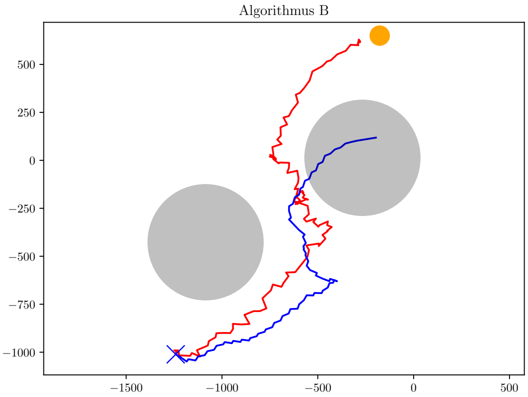


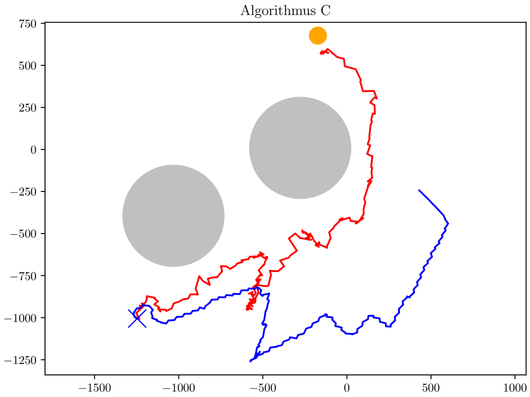

Algorithmus B
500
250
0
−250
−500
−750
−1000
−1000
−500
0
500
1000

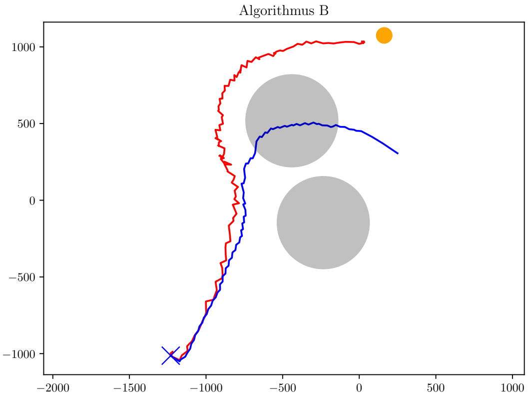

Algorithmus C
750
500
250
0
−250
−500
−750
−1000
−1500
−1000
−500
0
500

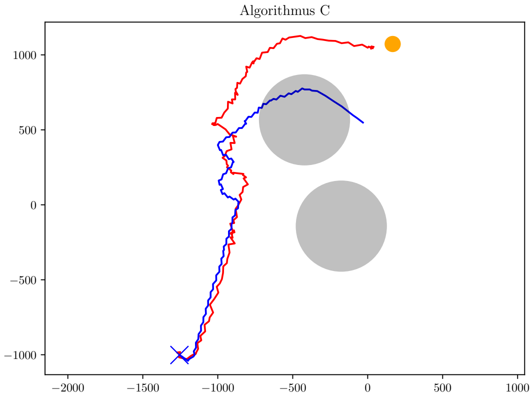

Algorithmus B
750
500
250
0
−250
−500
−750
−1000
−1250
−2000
−1500
−1000
−500
0
500

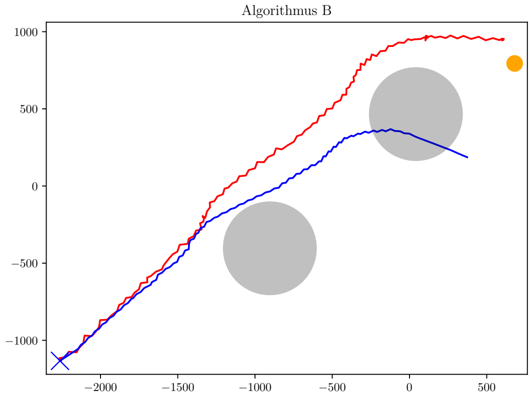

Algorithmus C
750
500
250
0
−250
−500
−750
−1000
−1250
−1500
−1000
−500
0
500

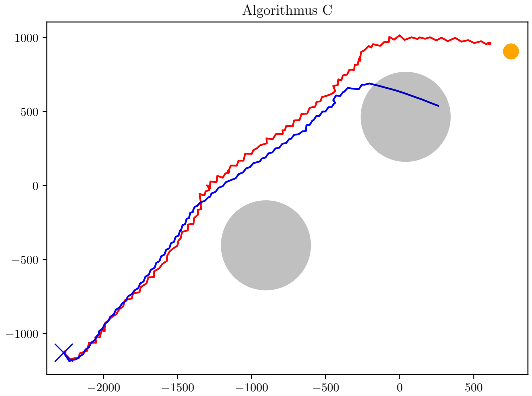

Algorithmus B
500
250
0
−250
−500
−750
−1000
−1500
−1000
−500
0
500

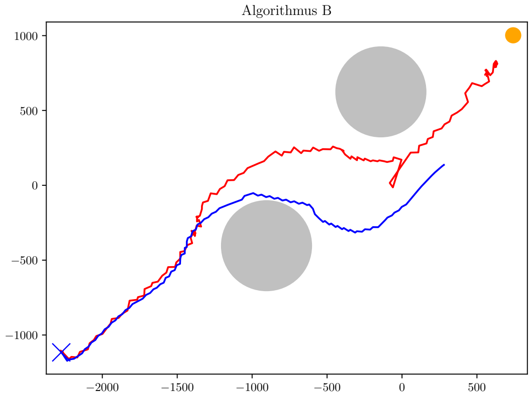

Algorithmus C
750
500
250
0
−250
−500
−750
−1000
−1250
−1500
−1000
−500
0
500
1000

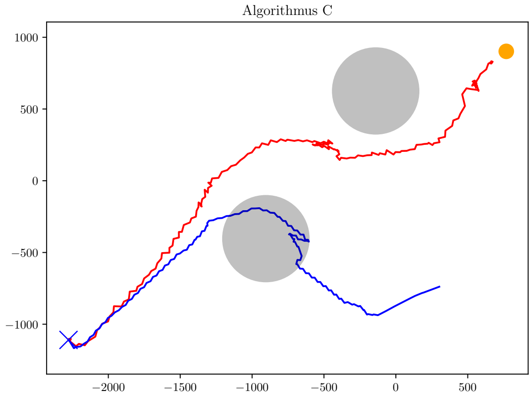

Algorithmus B
1000
500
0
−500
−1000
−2000
−1500
−1000
−500
0
500
1000


Algorithmus C
1000
500
0
−500
−1000
−2000
−1500
−1000
−500
0
500
1000

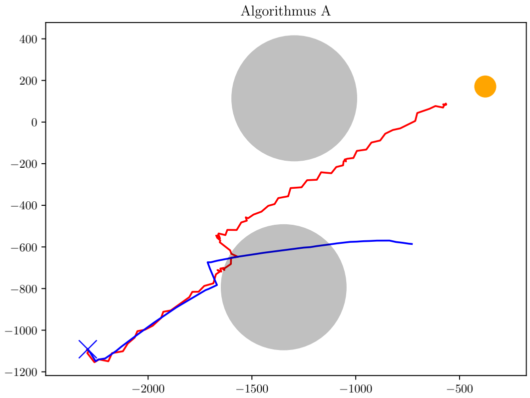

Algorithmus B
1000
500
0
−500
−1000
−2000
−1500
−1000
−500
0
500

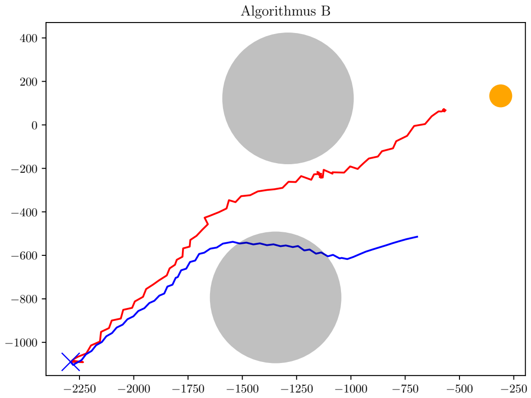

Algorithmus C
1000
500
0
−500
−1000
−2000
−1500
−1000
−500
0
500

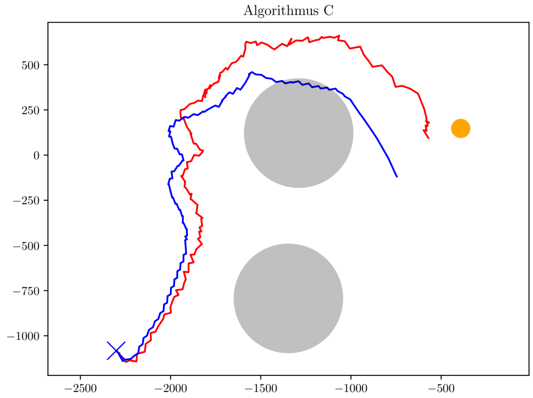

Algorithmus B
1000
500
0
−500
−1000
−2000
−1500
−1000
−500
0
500

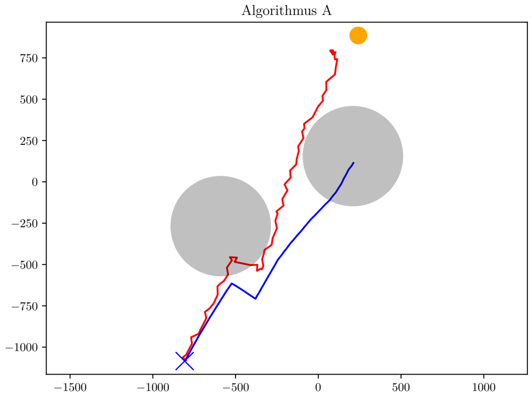

Algorithmus C
1000
500
0
−500
−1000
−2000
−1500
−1000
−500
0
500

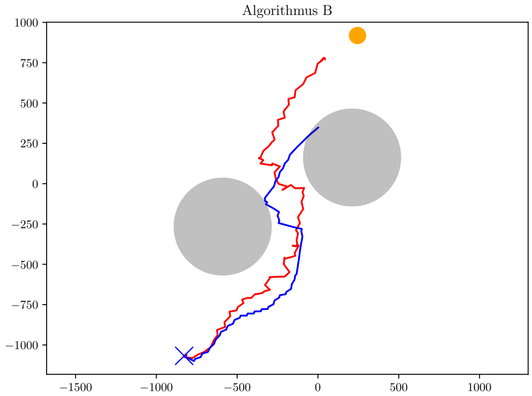
1.2 Zweite Experimentalreihe

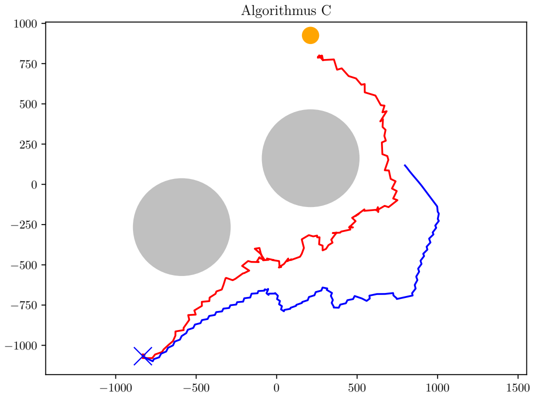


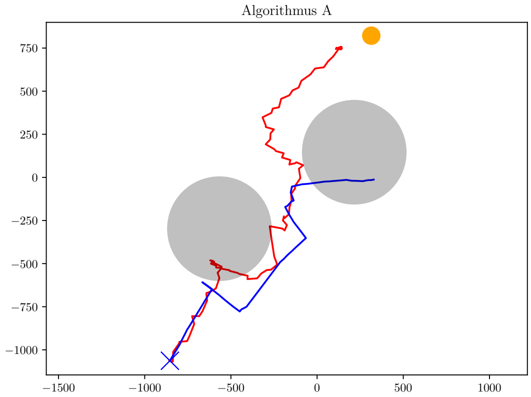


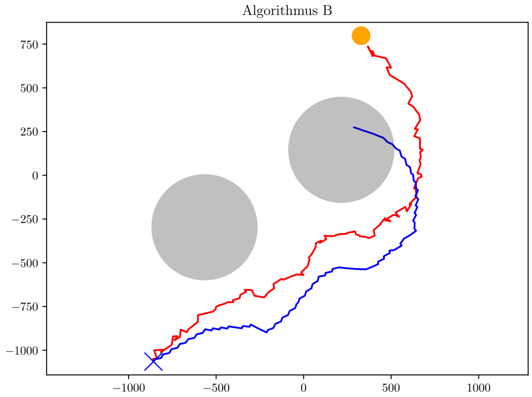


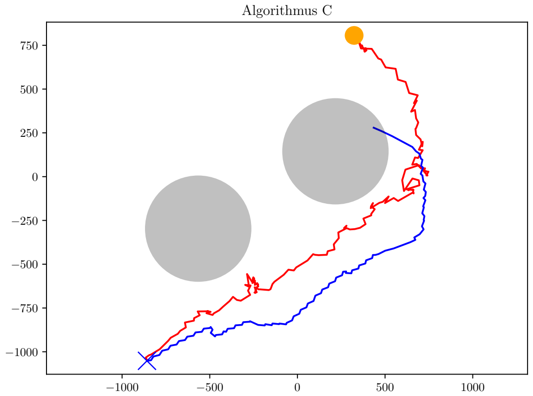

Algorithmus A


Algorithmus B


Algorithmus C


Algorithmus A


Algorithmus B


Algorithmus C

Literatur

[1] R. Deits and R. Tedrake. Footstep planning on uneven terrain with mixed-integer convex optimization. In *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*, pages 279–286. IEEE, 2014.

[2] K. Jolly, R. S. Kumar, and R. Vijayakumar. A bezier curve based path planning in a multi-agent robot soccer system without violating the acceleration limits. *Robotics and Autonomous Systems*, 57(1):23–33, 2009.

[3] M. Loetzsch, M. Risler, and M. Jungel. Xabsl-a pragmatic approach to behavior engineering. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 5124–5129. IEEE, 2006.

[4] H. Mellmann, B. Schlotter, and C. Blum. Simulation based selection of actions for a humanoid soccer-robot. In *RoboCup 2016: Robot Soccer World Cup XX*, 2016. to appear.

[5] H. Mellmann, B. Schlotter, S. Kaden, P. Strobel, T. Krause, and C.-N. Ritter. Berlin United - Nao Team Humboldt: Team Report 2016. Technical report, Humboldt-Universität zu Berlin, Adaptive Systems Group, 2016.

[6] H. Mellmann and Y. Xu. Adaptive motion control with visual feedback for a humanoid robot. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010)*, 2010.

[7] M. Missura. *Analytic and learned footstep control for robust bipedal walking.* PhD thesis, Universitäts-und Landesbibliothek Bonn, 2016.

[8] M. Missura and S. Behnke. *Balanced Walking with Capture Steps*, pages 3–15. Springer International Publishing, Cham, 2015.

[9] M. Nieuwenhuisen, R. Steffens, and S. Behnke. *Local Multiresolution Path Planning in Soccer Games Based on Projected Intentions*, pages 495–506. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[10] S. Rodríguez, E. Rojas, K. Pérez, J. López, C. A. Quintero, and J. M. Calderón. Fast path planning algorithm for the robocup small size league. In R. A. C. Bianchi, H. L. Akin, S. Ramamoorthy, and K. Sugiura, editors, *RoboCup 2014: Robot World Cup XVIII [papers from the 18th Annual RoboCup International Symposium, João Pessoa, Brazil, July 15*, volume 8992 of *Lecture Notes in Computer Science*, pages 407–418. Springer, 2014.

[11] B. Siciliano and O. Khatib. *Springer handbook of robotics.* Springer, 2016.

[12] Y. Xu. *From simulation to reality - migration of humanoid robot control.* PhD thesis, 2014.

[13] Y. Xu and H. Mellmann. Adaptive motion control: Dynamic kick for a humanoid robot. In R. Dillmann, J. Beyerer, U. Hanebeck, and T. Schultz, editors, *Proceedings of the 33rd Annual German Conference on Artificial Intelligence (KI 2010)*, volume 6359 of *Lecture Notes in Computer Science*, pages 392–399. Springer Berlin / Heidelberg, 2010.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 10. September 2017 ....