Softwareprojekt - Künstliche Intelligenz/FUmanoids

Schrittplanung

Manuel Zellhöfer
Mat.Nr. 4596223

WS 2011/12
22. April 2012

Inhaltsverzeichnis

1 Problemstellung und Aktueller Stand

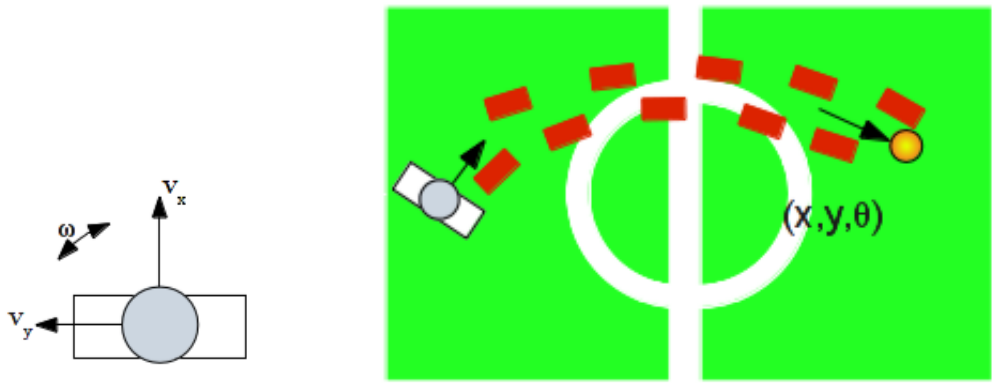
Humanoide Roboter zeichnen sich dadurch aus, dass Sie sich - nach Definition, wie der Mensch - mithilfe von zwei Beinen fortbewegen. Die Koordination die hierfür nötig ist, ist im aktuellen Stand des FUmanoid-Projekts einfach gehalten: Der Algorithmus, der die Beine des Roboters steuert (*Walker*) nimmt drei Situationsabhängige Geschwindigkeiten (Vorwärts v_x, Seitwärts v_y, Rotation ω, $\mathbf{v} = (v_x, v_y, \omega)$) vom *Verhalten*-Subsystem entgegen und berechnet daraus die Positionen für die Gelenke.

Probleme, die sich dadurch ergeben sind

- Instabilität durch Schwankungen in den Steuergrößen v_x, v_y und ω,
- ungünstige Trajektorien und
- wenig Kontrolle über die Trajektorie, die der Roboter vollführt.

Ein Schrittplaner soll dafür Abhilfe schaffen und eine flexiblere zusätzliche Ansteuerungsmöglichkeit für das *Verhalten*-Subsystem schaffen. Im Rahmen des Softwareprojekts, wird ein erster Prototyp entwickelt, der es erlaubt den Roboter durch Angabe einer relativen Zielposition ($\mathbf{p} = (x, y, \theta)$) zu steuern.

Die Bewegungssteuerung ist eine zentrale Komponente eines Roboters. Für humanoide Roboter ist diese aufgrund der komplexen Fortbewegungsart besonders aufwändig. Die aktuell verwendete Methode im FUmanoids-Projekt basiert auf dem Prinzip des passiv-dynamischen Gangs (*Passive Dynamic Walker*) unter Verwendung eines zentralen Mustergenerators (*Central Pattern Generator*) für die sich wiederholenden Bewegungsabläufe. Kurz gesagt: Mithilfe zweier alternierender Sinusschwingungen werden abwechselnd das linke und das rechte Bein des Roboters gehoben und je nach Steuergröße **v** bewegt. [4]



(a) Aktuell (b) Schrittplanung

Abbildung 1: Prinzipskizze Aktueller Stand / mit Schrittplaner

2 Lösungsansätze

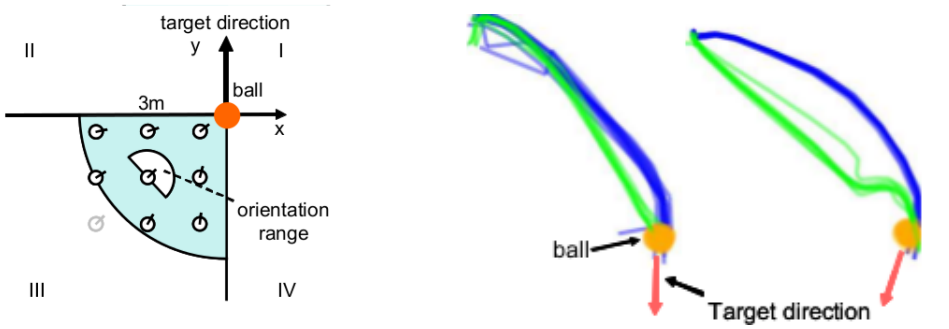
Im Umfeld des Robocup-Wettbewerbs sind elaborierte Schrittplanungsmethoden noch wenig verbreitet. Der Großteil der Teams arbeitet wie das FUmanoids-Team auch mit rein reaktiven Ansätzen [6]. Das Team NimbRo der Universität Bonn hat in [6] einen extensiv auf Maschinellem Lernen basierenden Ansatz vorgestellt (siehe 2.1).

In den übrigen Unterabschnitten werden Lösungen vorgeschlagen, die zuerst eine Bahn berechnen und davon ausgehend einzelne Schrittpositionen ermitteln.

2.1 Maschinelles Lernen

Der in [6] vorgestellte Ansatz versucht mithilfe von Maschinellen Lernens eine Bewegungsstrategie zu ermitteln, die in jeder Position des Roboters, relativ zu seiner Zielposition, den nächsten Schritt berechnet.

Hierfür werden offline für eine Reihe von Startpositionen $\mathbf{p}_i$ in einem Suchraum $R = X \subset \mathbb{R} \times Y \subset \mathbb{R} \times \Theta = [-\pi, \pi]$ mit dem A^*-Algorithmus gültige Schrittfolgen gesucht. Die ermittelten Schrittfolgen stellen dann den Trainingsdatensatz für die ML-Algorithmen (k-Nearest Neighbor, Multi-Layer-Perceptron und stückweise lineare Approximation) dar. Die Erstellung des



(a) Startpositionen im Suchraum

(b) Zwei Testläufe, blau mit / grün ohne Schrittplaner

Abbildung 2: Zur Veranschaulichung des ML-Ansatzes aus [6]

Trainingsdatensatzes erweist sich als sehr rechenintensiv. Die Auswertung der ML-Komponenten im Einsatz sind hingegen ausreichend effizient. Wie in Abbildung 2(b) zu sehen stellt die Verwendung des Schrittplaners eine deutliche Verbesserung dar. Alles in allem ist dieser Ansatz jedoch verhältnismäßig aufwändig und wird für das Projekt nicht in Betracht gezogen.

2.2 Splines

Eine einfachere Methode besteht darin, die gewünschte Trajektorie, die der Roboter vollführen soll zu berechnen und die Schritte die nötig sind davon ausgehend zu ermitteln. Abbildung 3 veranschaulicht die Idee.

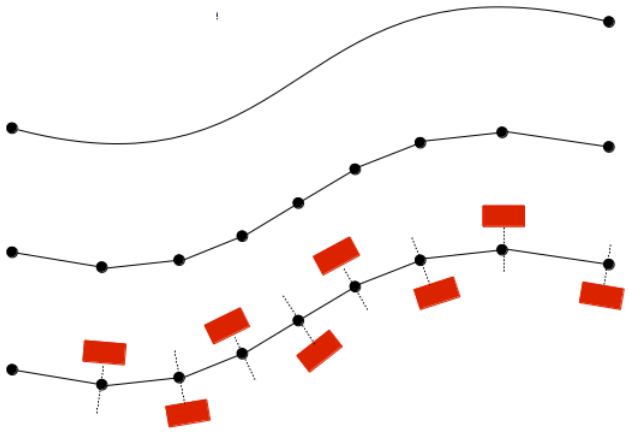


Abbildung 3: Prinzipielles Vorgehen zur Ermittlung einer Schrittfolge aus einer gegebenen Bahn

Für nichtholonome Roboter, die bestimmten Einschränkungen im Bahnradius unterliegen existieren Ansätze, die Bézier Kurven oder Splines verwenden um eine Bahn zu generieren ([2], [5]). Ausgehend von diesen Bahnen werden dann Steuergrößen (i.A. Vorwärts- und Rotationsgeschwindigkeit) ermittelt. Prinzipiell ist das Vorgehen zur Bahngenerierung folgendes:

1. Gegeben Start und Endposition, ermittle Randbedingungen für die Kurve
2. Füge falls nötig Zwischenpunkte ein
3. Ermittle Parameter durch Lösen eines Optimierungsproblems je nach Anforderung an die Bahn

An dieser Stelle hat man eine parametrisierte Kurve $q(t)$ wobei der Parameter t nicht notwendigerweise einer günstigen physischen Größe (Zeit, Strecke) entspricht. Dies eignet sich, um Steuergrößen für nichtholonome Fahrzeuge zu ermitteln (wie in [2], [5] geschehen). Anschließend müssten aus der Bahn die einzelnen Schritte berechnet werden. Dies ließe sich erreichen, indem man die Bahn abfährt und bei Überschreitung der maximalen Schrittlänge / Fußrotation einen Schritt setzt.

Die Umsetzung dieser Lösung scheint ebenfalls eher aufwändig zu sein. Mit Optimierung und der anschließenden Schrittberechnung müssen zwei möglicherweise ähnlich rechenintensive Operationen durchgeführt werden. Es wäre von Vorteil, wenn man die Bahngenerierung und die Schrittberechnung kombinieren könnte.

2.3 Kinematischer Regler

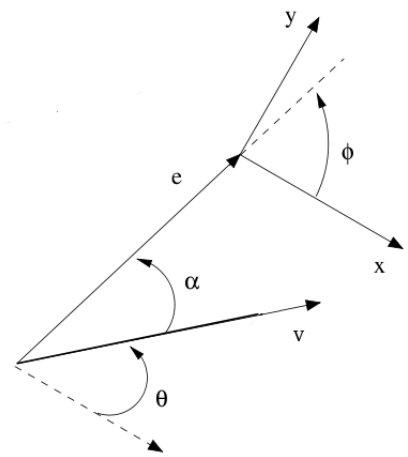
Eine andere Herangehensweise eine Bahn zu generieren, besteht darin basierend auf dem kinematischen Modell eines nichtholonomen Roboters (1) ein Regelgesetz abzuleiten. [1] und [3] sind Beispiele hierfür.

$$\begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= \omega \end{aligned} \tag{1}$$

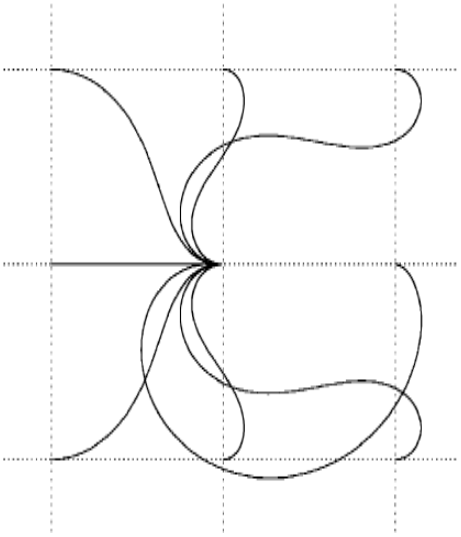
Da dies nicht ohne weiteres möglich ist, wird bei diesen Ansätzen zuerst eine Koordinatentransformation in Polarkoordinaten (siehe auch Abbildung 4(a)) durchgeführt:

$$\begin{aligned} e &= \sqrt{x^2+y^2} \\ \phi &= \text{atan2}(-y,-x) \\ \alpha &= \phi - \theta \end{aligned} \tag{2}$$

Zuvor wird der Koordinatenursprung in das Ziel verschoben. Die Größen e, ϕ und α werden dann als Regelabweichung betrachtet, die um den Roboter ins Ziel zu führen gegen 0 ausgeregelt werden müssen.



(a) Zur verdeutlichung der Koordinatentransformation. Am Ursprung des Koordinatensystems befindet sich das Ziel ($x = y = \theta = 0$) (aus [3])



(b) Bahnen für verschiedene Startpositionen bei Simulation (aus [3])

Abbildung 4: Details zur Bahngenerierung mit kinematischem Regler

Die beiden angeführten Arbeiten stellen dabei zwei unterschiedliche Regelgesetze vor. Um eine Bahn mithilfe eines solchen Regelgesetzes zu generieren, wird einfach das Modell (1) mit den berechneten Steuergrößen simuliert (Abb. 4(b)). Der Vorteil dabei ist der, dass während

der Simulation also dem „entlangfahren der Kurve", die Schritte je nach gegebenen Einschränkungen gesetzt werden können. Der Nachteil der Spline-Methode entfällt also. Zusätzlich sind die auszuwertenden Ausdrücke zur Bestimmung der Regelgrößen relativ einfach. Dieser Ansatz wurde folglich im Zuge des Softwareprojekts umgesetzt. Details hierzu folgen im nächsten Abschnitt.

3 Umsetzung

Bei der Umsetzung wurde darauf Acht gegeben, die Modularisierung, der die Software des FUmanoids-Projekts derzeit unterzogen wurde so gut wie möglich zu folgen. Dem Modul `motion/walking` wurde eine neue Klasse `FootstepPlanner` hinzugefügt. Der existierende *Walker* wurde geringfügig modifiziert, damit er die Schrittfolge, die der Planer berechnet, verwendet.

Die `FootstepPlanner`-Klasse stellt folgende Methoden zur Verfügung:

`setTarget` Setzt das Ziel relativ zur aktuellen Position des Roboters (x nach vorne, y nach rechts bzw. negativ nach links, θ Rotation rechts positiv bzw. links negativ, in mm bzw. $^\circ$)

`setMax{X|NegX|Y|Yaw}` Setzt Höchstwerte für Schrittpositionen[1]. *NegX* steht dabei für die Rückwärtsgeschwindigkeit.

`setPlanner` Setzt den zu verwendenden Planer

`calcSteps` Berechnet die Schritte

`getSteps` Gibt die Schritte als `deque`-Struktur zurück, so dass sie vom modifizierten *Walker* verwendet werden kann.

In den nächsten zwei Abschnitten wird die Funktionsweise der `FootstepPlanner`-Klasse beschrieben. Danach werden die Modifikationen des *Walkers* erläutert. Im letzten Abschnitt werden einige Probleme erläutert, die auftraten bzw. noch bestehen.

3.1 Berechnung der Bahn

Zur Berechnung der Bahn wird das Modell (1) in den privaten Methoden `calcStepsControl*` auf einfache Art und Weise zeitdiskret simuliert:

$$\mathbf{p}(t+1) = \mathbf{p}(t) + \mathbf{v} \cdot \Delta t$$

also

$$\begin{aligned} x(t+1) &= x(t) + v \cdot \cos\theta(t) \cdot \Delta t \\ y(t+1) &= y(t) + v \cdot \sin\theta(t) \cdot \Delta t \\ \theta(t+1) &= \theta(t) + \omega \cdot \Delta t \end{aligned} \tag{3}$$

[1] Wird voreingestellt gemäß den Konfigurationsparametern `motions.walker.max*Speed`. Bei der Einheit handelt es sich wohl um Millimeter.

wobei Δt als Konstante zu $0,001$ festgelegt wurde. Größere Werte sollten ohne Probleme möglich sein und würden sich positiv auf die Laufzeit auswirken. Die Größen v und ω werden mithilfe der vorher erwähnten Regelgesetze ermittelt. Zwei wurden hierfür implementiert:

- nach [3] ohne Rückwärtsbewegung, wählen mit `setPlanner(ControlInd)`:

$$\begin{aligned} v &= \gamma e & \gamma > 0 \\ \omega &= v\left(\tfrac{\sin\alpha}{e} + h\tfrac{\phi}{e}\tfrac{\sin\alpha}{\alpha} + \beta\tfrac{\alpha}{e}\right) & h > 1, 2 < \beta < h+1 \end{aligned} \tag{4}$$

- nach [1] mit Rückwärtsbewegung, wählen mit `setPlanner(ControlAic)`:

$$\begin{aligned} v &= \gamma \cdot \cos\alpha \cdot e & \gamma > 0 \\ \omega &= k\alpha + \gamma\tfrac{\cos\alpha\sin\alpha}{\alpha}(\alpha + h\phi) & k, h > 0 \end{aligned} \tag{5}$$

mit den transformierten Größen e, α, ϕ nach (2). Die Parameter wurden nach den in den jeweiligen Veröffentlichungen vorgeschlagenen Werten festgelegt.

3.2 Ermitteln der einzelnen Schritte

Der *Walker* verfügte bereits über rudimentäre Funktionalität zur Verwendung eines Schrittplaners. So entspricht der Geschwindigkeitsvektor **v** den er vom *Verhalten*-Subsystem erhält der gewünschten Position des Mittelpunkts des Roboters nach dem nächsten Schritt. Die Berechnung der Schrittfolge erfolgt nun parallel zur Berechnung der Bahn: An dem Punkt in der Bahn zu dem der Roboter einen Schritt machen müsste, der an der Grenze des zulässigen Bereichs liegt, wird ein Schritt gesetzt (siehe auch Abb. 5).

Um dem Roboter langsames Beschleunigen bzw. Abbremsen zu ermöglichen wird dabei diese Grenze dynamisch unter Berücksichtigung der vorhergehenden Schrittlänge definiert. Die private Methode `checkStep` entscheided dabei ob ein Schritt, bei gegebenem Abstand zum Ziel und fehlendem Winkel zur gewünschten Orientierung der Schrittfolge hinzugefügt werden soll oder nicht. Sie wird in jedem Simulationsschritt aufgerufen.

3.3 Anpassung des Walker-Algorithmus

Die Änderungen die am *Walker* durchgeführt wurden waren eher geringfügig. Rudimentäre Funktionalität war wie oftmals erwähnt bereits vorhanden. Die Abstrakte `MotionWalker`-Klasse enthielt Attribute und Methoden (`steps`, `setNextStep`) die quasi direkt übernommen wurden[2].

Um besser auf schwankende Eingaben vom *Verhalten*-Subsystem zu reagieren und um den Gang zu stabilisieren implementiert der *Walker* einen Mechanismus, der die Bewegungen in

[2] Die Struktur von `steps` wurde von einer `queue` zu einer `deque` geändert.

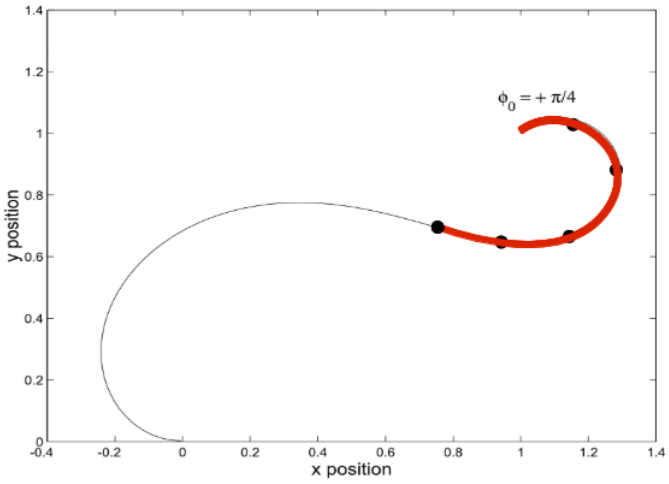


Abbildung 5: Ermitteln einer Schrittfolge während der Simulation der Roboterbewegung. Hier nicht berücksichtigt ist der Beschleunigungsvorgang, der die Schrittlänge stufenweise vergrößert.

den ersten Sekunden des Gehens klein hält und mit der Zeit Vergrößert. Dies geschieht durch einen `startingFactor` der in einem definierten Zeitraum von 0 auf 1 linear wächst.

Da der Schrittplaner diese Beschleunigungsphase berücksichtigt, und die Schrittlängen sukzessive vergrößert, entfällt die Notwendigkeit für einen solchen Mechanismus. Befindet sich der *Walker* nun also im `footPlanningMode` (ein Attribut der `MotionWalker`-Basisklasse), so wird dieser `startingFactor` konstant auf 1 gesetzt. Lediglich die Schritthöhe unterliegt noch dieser Verzögerung.

Eine Klasse zur Repräsentation eines einzelnen Schrittes (`Step`) wurde erweitert, um Operationen wie Addition, Subtraktion und Skalierungen zu ermöglichen.

3.4 Aufruf des Schrittplaners

Es hat sich herausgestellt, dass der Walker mit diesen geringfügigen Veränderungen gut mit den geplanten Schrittfolgen funktioniert. Ein Beispiel für den Aufruf ist in der Datei `test.cpp` zu finden. Testen des Planers ist durch den Aufruf von

```
FUmanoid --minimal test --x=int --y=int --yaw=int --planner={aic|ind}
```

möglich. Im Grunde verläuft der Aufruf wie in Abb. 6.

Die `FootstepPlanner`-Klasse ist noch nicht in das modularisierte Framework eingebunden. Der Aufruf in Zeile 10 ist mit der aktuellen Version aus dem git-Repository nicht mehr mög-

```
1  FootstepPlanner *fsp = new FootstepPlanner();
2
3  fsp->setPlanner(ControlAic);              // nicht zwingend nötig
4  fsp->setTarget(x,y,yaw);
5  fsp->calcSteps();
6
7  MotionWalker::setFootPlanningMode( true );
8  MotionWalker::setSteps( fsp->getSteps() );
9
10 robot.switchMotion(ID_MOTION_WALKER, false);
```

Abbildung 6: Aufruf des Schrittplaners

lich. Der gezeigte Quelltext berechnet in jedem Fall die Schritte und gibt diese auf der Konsole aus[3].

3.5 Probleme

Abgesehen von einigen organisatorischen Schwierigkeiten, die nichts mit dem Projekt zu tun haben gestaltete sich die Entwicklung des Schrittplaners verhältnismäßig einfach. Die Tatsache, dass der *Walker* den Geschwindigkeitsvektor den er vom *Verhalten*-Subsystem empfängt als Schrittlänge umsetzt und im Zuge dessen quasi völlig auf dem existierenden *Walker* aufgebaut wurde, trug dazu nicht von ungefähr bei. Probleme zeigten sich lediglich bei der Implementierung der Simulation und des Reglers, ließen sich jedoch auf einfache Verständnisschwierigkeiten, Vorzeichenfehler und ähnliche Dummheiten zurückführen. Ansonsten soll noch auf folgende fortbestehende Probleme eingegangen werden:

Schlupf, Ungenauigkeit

In der aktuellen Version berechnet der Schrittplaner eine komplette Schrittfolge von der Start- in die gewünschte Zielposition. Der Walker nimmt zwar die gewünschten Schrittlängen als Input entgegen, konstruktionsbedingt (da sich auf dem Prinzip des *Passive Dynamic Walkers* fortbewegt wird) werden aber immer, für gewöhnlich sogar deutliche Abweichungen von der gewünschten Schrittlänge auftreten. Diese Abweichungen werden bisher überhaupt nicht berücksichtigt.

[3] Ob der *Walker* diese dann auch abläuft ist mir zum Abgabetermin unklar.

Ungünstige Bahnen vom Regler

Bei den Reglern die zur Bahngenerierung verwendet werden, handelt es sich um Regler für nichtholonome, sich üblicherweise auf Rädern fortbewegende Roboter. Für gewisse Startpositionen berechnen diese unerwartet ungünstige Bahnen. Soll sich der Roboter beispielsweise einfach einen Meter zurück bewegen, würde er bei beiden Reglern versuchen über pirouettenhaft anmutende Bahnen seine Zielposition einzunehmen.

4 Ausblick

Der Schrittplaner ist ein erster Prototyp. Ein erster Versuch zeigte bereits vielversprechende Ergebnisse. Von einer funktionstüchtigen Einbettung in die übrige Software ist er allerdings noch ein gutes Stück entfernt. Folgende Maßnahmen könnten sich dabei als nützlich erweisen:

On-line Schrittplanung

Die Schrittplanung erfolgt off-line im vornherein. Um zu garantieren, dass der Roboter sein Ziel auch wirklich findet, ist die Verwendung von Informationen der Selbstlokalisation unerlässlich. Mit der aktuellen Vorgehensweise, im vornherein die komplette Schrittfolge zu berechnen, ist dies nicht umzusetzen. Der Schrittplaner müsste so implementiert werden, dass er zu jeder Position den/die nächsten Schritt/e berechnet. Hierfür könnte es bereits genügen, die Simulation nach der Berechnung einer gewissen Anzahl von Schritten anzuhalten und diesen Vorgang regelmäßig durchzuführen.

Einfache Bahngenerierung

Es stellt sich auch die Frage, ob die Bahngenerierung nicht noch einfacher erfolgen könnte. Die Methode `calcStepsBasic` stellt eine sehr einfache Methode dar, das Ziel durch die Befehlsfolge

1. Rotation bis die Zielposition in der Gangrichtung des Roboters liegt
2. Geradeausgehen bis kurz vor der Zielposition
3. Rotation bis gewünschte Orientierung erreicht
4. Einnehmen der Zielposition

zu erreichen. Inwiefern sich dieses vorgehen für On-line Schrittplanung eignet bliebe herauszufinden. Eine andere Möglichkeit bestünde darin, den Regler durch einige Fallunterscheidungen robuster zu gestalten[4].

[4] Dies geschieht in Teilen bereits in der Methode `calcStepsControlInd`. Dort ist Rotation am Platz implementiert, ohne das das Regelgesetz dies definitionsgemäß unterstützt.

Literatur

[1] M. Aicardi, G. Casalino, A. Bicchi, and A. Balestrino. Closed loop steering of unicycle like vehicles via lyapunov techniques. *Robotics & Automation Magazine, IEEE*, 2(1):27--35, 1995.

[2] Ji-wung Choi, E. Curry Renwick, and Gabriel Hugh Elkaim. Smooth path generation based on bezier curves for autonomous vehicles. In *Proceedings of the World Congress on Engineering and Computer Science Vol II*, 2009.

[3] G. Indiveri. Kinematic time-invariant control of a 2d nonholonomic vehicle. In *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, volume 3, pages 2112--2117, 1999.

[4] Johannes Kulick. Ein stabiler gang für humanoide, fußball spielende roboter. Master's thesis, Freie Universität Berlin, Oktober 2011.

[5] Aurelio Piazzi, Corrado Guarino Lo Bianco, and Massimo Romano. *Smooth Path Generation for Wheeled Mobile Robots Using η^3-Splines.* Motion Control. 2010.

[6] A. Schmitz, M. Missura, and S. Behnke. Real-time trajectory generation by offline footstep planning for a humanoid soccer robot. 2011.