Hochschule Harz
Fachbereich Automatisierung und Informatik

Teamprojekt

für das Fach Programmierung mobiler Agenten
im Studiengang Master of Computer Science

lejON – LeJOS Odometric Navigator

vorgelegt von:

Jan Grohmann,
Annedore Rößling,
Florian Ruh,
Kai Schories

Inhaltsverzeichnis

1. Einleitung

Vorwort

Das vorliegende Dokument ist die Abschlussarbeit eines Teamprojekts im Masterstudiengang „Informatik/Mobile Systeme“, welches für das begleitende Seminar „Programmierung mobiler Agenten“ durchgeführt wurde. Es behandelt Probleme der odometrischen Navigation mobiler Roboter – auch als *dead reckoning* bezeichnet – und entwickelt Lösungen zur softwarebasierten Regelung und Kontrolle der Bewegung. Es wird ein Konzept einer Navigator-API für zweirädrige Lego-Mindstorms-Roboter mit LeJOS-Kernel vorgestellt und anhand einer Referenzimplementierung für Differential-Drive-Architekturen demonstriert, getestet und diskutiert.

Wir danken Herrn Prof. Dr. Frieder Stolzenburg für die ausgezeichnete Betreuung dieses Projektes, welches uns neue interessante Einblicke und Erfahrungen in Teamarbeit, Robotik und Softwareentwicklung eröffnet. Darüberhinaus danken wir ihm sehr für seine Geduld mit den häufigen, von uns hinausgeschobenen Terminen. Zudem danken wir Herrn Michael Wisse für die schnelle und kompetente Hilfe bei Fragen rund um LeJOS und die Lego Mindstorms.

Jan Grohmann, Annedore Rößling, Florian Ruh, Kai Schories
Wernigerode, den 21. Juni 2006.

1.1. Motivation und Aufgabenstellung

Ein mobiler Roboter ist ein System von verschiedenen physikalischen (Hardware) und logischen Komponenten (Software). Bezüglich der Hardware besitzt jeder autonom agierende, mobile Roboter Subsysteme für

Locomotion Fortbewegung in der Umwelt

Sensing Wahrnehmung von Reizen aus der Umwelt

Reasoning Umsetzung der wahrgenommenen Reize in Aktionen

Communication Kommunikation mit der Umwelt

Terrestrische Roboter bewegen sich auf einem festen Untergrund fort und benötigen dafür ständigen Bodenkontakt. Die meisten solcher Vehikel nutzen Räder zur Fortbewegung. Die Möglichkeit zur Fortbewegung ist Voraussetzung für Mobilität; die exakte Navigation, d. h. die genaue Planung und Ausführung von Fahrmanövern, ist Voraussetzung für höhere Funktionen, wie z. B. das autonome Operieren in entsprechenden Einsatzumgebungen. Eine häufig eingesetzte Technik zur Messung der Fortbewegung auf Basis von Rädern ist die Odometrie (vgl. Kapitel 1.2.2). Hier werden Wege und Winkel anhand der Umdrehungen der Räder gemessen und verarbeitet. Theoretisch kann man ausschließlich durch Anwendung von Odometrie die Position exakt bestimmen und somit den Roboter „blind“ navigieren. Praktisch treten jedoch Störungen wie z. B. Schlupf und Schlittern der Räder auf, was zu Messungenauigkeiten und somit zu einer Abweichung der Soll- zur Ist-Position führt. Navigationskomponenten auf odometrischer Basis müssen also neben der möglichst genauen Messung und Steuerung der Radbewegungen auch in der Lage sein, die praktisch zwangsläufig auftretenden Störungen der Radbewegungen zu erkennen und ggf. zu korrigieren.

Im KI-Labor der Hochschule Harz werden Grundlagen des Designs und der Programmierung von mobilen Robotern anhand von selbstgebauten Modellen auf Basis von Lego Mindstorms [1] mit LeJOS-Kernel [2] vermittelt. Lego stellt die Hardware für Aktorik und Sensorik bereit. LeJOS ist ein Framework auf Java-Basis bestehend aus einem Kernel (VM) zur Steuerung der Lego-Hardware und einer API bestehend aus Modulen für die Hardware-Abstraktion und Kommunikation mit dem Roboter.

Die Aufgabenstellung für dieses Teamprojekt ist die Untersuchung, Verbesserung und Erweiterung der *Navigator*-Klasse von LeJOS. Diese Klasse stellt Funktionen zur odometrischen Navigation von Differential Drives (vgl. Kapitel 1.2.1) bereit und besitzt einen einfachen Regelmechanismus zur Fehlerkorrektur. Eine Option ist die Definition und Implementierung einer komplett neuen Architektur, die hier auch tatsächlich erfolgt.

[1] http://www.lego.com/
[2] http://lejos.sourceforge.net/

Im folgenden Kapitel werden zunächst grundlegende Begriffe und Prinzipien erklärt, welche für das Verständnis des in Kapitel 1.3 vorgestellten Konzeptes notwendig sind. Am Ende werden daraus die projektspezifischen Anforderungen für eine neue, robuste Navigator-Architektur abgeleitet.

1.2. Theoretische Grundlagen und Begriffsklärungen

Terrestrische Roboter mit Radantrieb, im Folgenden kurz als WMR („wheeled mobile robot“) bezeichnet, lassen sich anhand der Art und Weise, wie Vortrieb und Lenkung über die Radbewegungen realisiert werden, in Klassen einteilen (vgl. [DJ00, S. 16ff.]):

- Differential Drive
- Synchronous Drive
- Bicycle, Tricycle
- Car Drive (Ackermann Drive)
- Omnidirectional Drive

Jede WMR-Klasse definiert eine eigene Kinematik. Die Kinematik beschreibt die Beziehung zwischen den Kontrollparametern und dem Bewegungsverhalten des Systems. Jede Klasse hat bezüglich der Manövrierfähigkeit unterschiedliche Freiheitsgrade, siehe dazu [Bru05, Kap. 3].

Definition 1.2.1 (Pose)**.** Die Bewegung eines WMR in der Ebene wird als Bewegung eines Punktes, dem *Referenzpunkt* des WMR, modelliert. Hinsichtlich der ebenen Bewegung besitzt der WMR 3 Freiheitsgrade: eine (x, y)-Position und einen Richtungswinkel θ. [3] Das Tripel (x, y, θ) wird als *Pose* oder auch *Bewegungsvektor* des Roboters bezeichnet.

Definition 1.2.2 (ICC)**.** Ein WMR in Bewegung besitzt zu jedem Zeitpunkt einen Punkt, um den er rotiert (vgl. Abb. 1.1). Dieser Punkt wird als *instantaneous center of curvature*, kurz ICC, bezeichnet. Der Abstand des Referenzpunktes zum ICC ist der Kurvenradius R. $R \to \infty$ impliziert eine reine Translation; $R = 0$ impliziert eine Punktrotation.

Über die Steuerung der Radgeschwindigkeiten oder der Lenkwinkel der Räder kann der WMR seinen ICC verändern. Dies wird als *steering* bezeichnet.

1.2.1. Differential Drives und ihre Kinematik

Differential Drives (kurz *DD*) sind bzgl. des Antriebsmechanismus die einfachste Klasse der WMRs. Wie in Abb. 1.1 dargestellt, ist ein DD aus zwei auf einer gemeinsamen[4] Achse montierten Rädern aufgebaut, welche von zwei unabhängigen

[3] Hier gilt die Vereinbarung $\theta = 0$ impliziert eine Ausrichtung entlang der $+x$-Achse der Ebene; $\theta > 0$ impliziert eine Drehung gegen den Uhrzeigersinn.

[4] „Gemeinsam“ ist nicht ganz richtig: praktisch sind die Antriebsräder auf separaten Antriebsachsen montiert, welche aber entlang einer gedachten Line senkrecht zur Fahrtrichtung ausgerichtet sind.

Motoren angetrieben werden. Im Allgemeinen besitzen DDs ein zusätzliches Rad (oder Kontaktpunkt), welches aber auf die Kinematik idealerweise keinen Einfluss hat. Man bezeichnet dieses Stützrad auch als *castor wheel*.

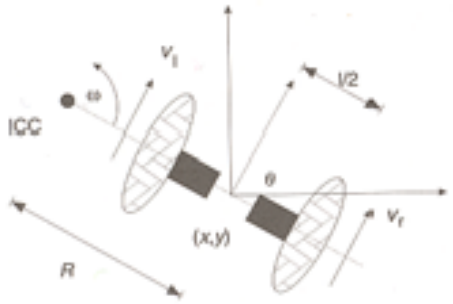


Abbildung 1.1.: DD-Modell
Quelle: [DJ00]

Kinematik Die Bewegung des DDs wird ausschließlich über die Geschwindigkeiten v_l und v_r der Räder gesteuert. Durch Variation der Relativgeschwindigkeit kann gleichzeitig eine Translation und eine Rotation des Drives erzeugt werden; das DD hat damit 2 Freiheitsgrade in der Bewegung. Tabelle 1.1 zeigt das Bewegungsverhalten bei verschiedenen Varianten der Relativgeschwindigkeit.

	Figur	
$v_l = v_r$	**Translation**	
$v_l < v_r$	**Linkskurve**	
	$v_l \geq 0$	$v_l = -v_r$
	Linkskurve mit $R \geq l/2$	Linkskurve mit $R = 0$, Linksrotation
$v_l > v_r$	**Rechtskurve**	
	$v_r \geq 0$	$v_r = -v_l$
	Rechtskurve mit $R \geq l/2$	Rechtskurve mit $R = 0$, Rechtsrotation

Tabelle 1.1.: Steuerung der Bewegung beim DD

Die Berechnung der Pose des DD erfolgt über die Gleichungen 1.2.1 bis 1.2.4, welche die die sog. *Vorwärtskinematik* des DD definieren.

Vorwärtskinematik des DD Seien v_r und v_l die Radgeschwindigkeiten und l die Achslänge, d. h. der Abstand zwischen den Achspunkten beider Räder, mit $l > 0$. ω

sei die Richtungswinkelgeschwindigkeit des DD. Für $\theta > 0$ gilt dann

$$\begin{aligned} \omega(R + l/2) &= v_r \\ \omega(R - l/2) &= v_l \end{aligned} \tag{1.2.1}$$

Für $\theta < 0$ werden v_r und v_l in Gleichung 1.2.1 vertauscht. Durch Lösung des Gleichungssystems nach R und ω ergibt sich

$$\begin{aligned} R &= \frac{l}{2} \cdot (\frac{v_r + v_l}{v_r - v_l}) \\ \omega &= \frac{v_r - v_l}{l} \end{aligned} \tag{1.2.2}$$

Liegt der Referenzpunkt des DD auf der Achsmitte, so berechnet sich die *Referenzpunktgeschwindigkeit* v zu

$$v = \frac{1}{2}(v_l + v_r) \tag{1.2.3}$$

Die Pose zu einem Zeitpunkt t ergibt sich durch Integration von ω und v über die Zeit. In Kapitel 1.3.2 wird erläutert, wie die Pose praktisch berechnet wird.

$$\begin{aligned} \theta(t) &= \int_0^t \omega(t)dt \\ x(t) &= \int_0^t v(t)\, cos\, \theta(t)\, dt \\ y(t) &= \int_0^t v(t)\, sin\, \theta(t)\, dt \end{aligned} \tag{1.2.4}$$

□

1.2.2. Odometrie

Odometrie (griech. *hodos*: der Weg, *metron*: das Maß) ist die Messung von Wegen und Winkel anhand der Radbewegungen. Hierfür werden Rotationssensoren verwendet, welche die Umdrehungen der Räder um ihre Achspunkte messen. Die Sensoren arbeiten als inkrementelle Kodierer, welche den Drehwinkel der Sensorwelle auf einen Code abbilden, vgl. [CH05]. Deshalb spricht man auch vom *shaft encoder*. Der Einfachheit halber nehmen wir hier an, dass in natürlichen Zahlen kodiert wird, so dass eine Raddrehung um 360° einer definierten Anzahl von Impulsen, sog. TICKS, entspricht.

Kenngrößen der Rotationssensoren Entscheidend für die Messgenauigkeit ist die *Winkelauflösung*. Diese hängt von zwei Parametern ab:

1. $\overline{\text{TICKS}}$ – Nominalticks des Rotationssensors
   Dieser Wert ist definiert als die Anzahl der TICKS pro Umdrehung der Sensorwelle und wird von der Sensor-Hardware vorgegeben.

2. N_G – Übersetzungsverhältnis (gear ratio)
   Eine (nachträgliche) Erhöhung der Winkelauflösung erhält man durch N:M-Übersetzung, mit $N > M$, der Radachse auf die Sensorwelle mittels Getrieberädern („gear train“); siehe dazu [Bir99].

Die tatsächliche Winkelauflösung wird über den Effektivwert TICKS# nach Gleichung 1.2.5 angegeben werden.

$$\text{TICKS\#} = N_G \cdot \overline{\text{TICKS}} \tag{1.2.5}$$

Odometrische Skalierungsfaktoren Zur Umrechnung der gemessenen TICKS auf Wege und Winkel, welche wir hier in Einheiten [cm] und [°] beschreiben, werden die Konstanten TPC, DTPD und TPD nach 1.2.6, 1.2.7 und 1.2.8 definiert.

Definition 1.2.3 (TPC, ticks per centimeter)**.** Sei d der konstante Raddurchmesser in [cm]. Dann ergibt sich die Anzahl der TICKS pro 1 Zentimeter zurückgelegtem Weg des Radmittelpunktes in Rollrichtung zu

$$\text{TPC} = \frac{\text{TICKS\#}}{\pi \cdot d} \tag{1.2.6}$$

□

Die bei einer Relativgeschwindigkeit $v_r - v_l \neq 0$ entstehende Richtungswinkeländerung um $\pm 1°$ entspricht einer konstanten Differenz $\Delta\text{TICKS}_{\text{r,l}}$ zwischen rechtem und linkem Rad. Anschaulich ausgedrückt dreht sich der Achspunkt des schnelleren Rades um den Achspunkt des langsameren Rades.

Definition 1.2.4 (DTPD, differential ticks per degree)**.** Sei l der konstante Abstand in [cm] zwischen den Achspunkten der beiden Räder des DD. Die Anzahl der Differenz-TICKS pro 1° Richtungswinkeländerung ist definiert als

$$\begin{aligned} \frac{\Delta\text{TICKS}_{\text{r,l}}}{1°} &= \text{konst.} = \frac{2\pi \cdot l}{360°} \cdot \text{TPC} \\ \text{DTPD} &= \frac{\pi \cdot l}{180} \cdot \text{TPC} \end{aligned} \tag{1.2.7}$$

□

Lässt man beim DD die Räder gegensinnig mit gleichem Geschwindigkeitsbetrag rotieren, so rotiert das DD um seinen Achsmittelpunkt. Der Kurvenradius ist $R = 0$, der ICC liegt im Achsmittelpunkt. Die Radachspunkte rotieren im Abstand $l/2$ um den ICC. Ersetzt man in Gleichung 1.2.7 l durch $l/2$, so erhält man die TPD-Konstante des DD.

Definition 1.2.5 (TPD, ticks per degree)**.** Wenn gilt: $v_r = -v_l \neq 0$, dann ist die Anzahl der TICKS pro Grad Achsdrehung des DD definiert als

$$\text{TPD} = \frac{\pi \cdot l}{360} \cdot \text{TPC} \tag{1.2.8}$$

□

Zusammenfassung Die Position eines sich in Bewegung befindenden WMR ist über seine Vorwärtskinematik bestimmt und kann im Idealfall durch Odometrie exakt berechnet werden. Praktisch treten jedoch systematische Messfehler und externe Störungen auf, welche die exakte Navigation erschweren. Der nächste Abschnitt geht auf dieses Problem ein und nennt mögliche Lösungen. Abschnitt 1.2.4 beschreibt ein allgemeines Konzept zur praktischen Fehlerkompensierung.

1.2.3. Fehlerquellen und Störung der Bewegung

Man unterscheidet in technischen Systemen generell 2 Arten von Fehlern:

1. Systematische Fehler
   Einseitige Abweichungen der Messwerte von ihrem Erwartungswert, deren Ursachen meist auf Messungenauigkeiten und schlechte Justierung zurückzuführen sind.

2. Nichtsystematische („zufällige“) Fehler
   Im Gegensatz zu systematischen Fehlern weisen nichtsystematische keinen erkennbaren Trend (*bias/offset*) auf

Systematische Fehler treten bei der odometrischen Navigation *immer* auf. Sie entstehen durch:

- endliche Auflösung der Rotationssensoren
- endliche Messfrequenz
- Toleranzen der Motoren (insbesondere beim DD kritisch)
- Komprimierung und Exzentrik der Räder
- Trägheit des WMR

Nichtsystematische Fehler entstehen durch

- Bodenunebenheiten
- Schlupf und Schlittern durch schlechten Bodenkontakt
- weitere, unbekannte Störquellen

Da Odometrie ein iteratives Verfahren ist, summieren sich kleine Fehler und mit wachsendem zurückgelegtem Weg wächst die Unsicherheit bzgl. der geschätzten Position. Ziel muss es also zunächst sein systematische Fehler zu minimieren. Maßnahmen sind u. a.

- Erhöhung der Winkelauflösung

- hohe Messfrequenz
- Verwendung nicht komprimierbarer Räder, welche senkrecht auf der Rollebene stehen und deren Kreisfläche mit der Achse einen rechten Winkel bildet
- Einkalkulieren der Trägheit des WMR bei Beschleunigung (insbesondere beim Stoppen). Dies ist im Allgemeinen sehr schwierig, da man hierfür das systemdynamische Modell des WMR benötigt, welches meist nicht vorab bekannt ist und experimentell ermittelt werden muss. Hierfür sind i. d. R. Beschleunigungs- und Drehratensensoren notwendig.

Des Weiteren müssen bleibende Abweichungen dynamisch kompensiert werden. Generell gibt es für WMRs zwei Ansätze, welche meist in Kombination verwendet werden.

1. Periodische Rekalibration der Position mit einem zweiten, odometrieunabhängigen Verfahren, z. B. GPS, Ultraschall
2. Einsatz von mechanischen, elektronischen oder softwarebasierten Regelkreisen zur Kontrolle der Radgeschwindigkeiten, vgl. Abschnitt 1.3.4

Definition 1.2.6 (Quantisierungsfehler bei Messung des Wegs)**.** Gegeben sei ein ideales Rad nach Standardradmodell[5] mit einem Durchmesser $d > 0$ in [cm] und ein Rotationssensor mit der Kenngröße TICKS#> 0. Der Quantisierungfehler q_s bei Messung des zurückgelegten Weges des Rads ist dann

$$q_s = \frac{1}{\text{TPC}} = \frac{\pi \cdot d}{\text{TICKS\#}} \tag{1.2.9}$$

□

Aus Gleichung 1.2.9 folgt: Um q_s zu minimieren, muss der Raddurchmesser möglichst klein oder TICKS# möglichst groß gewählt werden.

Definition 1.2.7 (Quantisierungsfehler bei Messung der Richtungswinkeländerung beim DD)**.** Gegeben sei ein DD mit einer Achslänge $l > 0$ in [cm] und einem TPC-Wert nach Gleichung 1.2.6. Der Quantisierungsfehler bei Messung des Richtungswinkels ist dann

$$q_\theta = \begin{cases} \frac{1}{\text{DTPD}} & = \frac{180}{\pi \cdot l} \cdot \frac{1}{\text{TPC}} \quad \text{falls Kurvenfahrt} \\ \frac{1}{\text{TPD}} & = \frac{360}{\pi \cdot l} \cdot \frac{1}{\text{TPC}} \quad \text{falls reine Rotation} \end{cases} \tag{1.2.10}$$

□

Aus Gleichung 1.2.10 folgt: Um q_θ zu minimieren, muss neben einem großen TPC-Wert auch die Achslänge möglichst groß gewählt werden.

[5] vgl. [AM88]

Bemerkung 1.2.1 (Gleichung 1.2.10)**.** Egal wie klein q_θ praktisch ist – bei der Punktrotation ist der Quantisierungsfehler immer doppelt so groß wie bei der Kurvenfahrt mit $R \geq l/2$, vgl. Tabelle 1.1.

Der nächste Abschnitt stellt ein allgemeines Konzept zur Fehlerkompensierung vor.

1.2.4. Regeltheorie

Das Verhalten von idealen physikalischen Systemen hängt nur von den Systemparametern und den Eingangsgrößen zum Zeitpunkt t ab. Wie im vorhergehenden Abschnitt erläutert, treten jedoch praktisch immer Störungen auf. Da die Störfaktoren meist nicht vollständig eliminiert werden können, ist eine aktive Regelkomponente notwendig, welche die Effekte der Störungen kompensiert. Abbildung 1.2 zeigt das Konzept einer solchen Regelung.

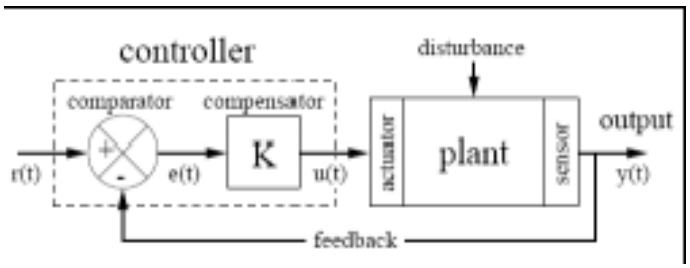


Abbildung 1.2.: Schema einer closed-loop-Regelung
Quelle: [Bir01]

Größen und Komponenten von closed-loop-Systemen Die Bezeichnung *closed-loop* leitet sich aus der Rückführung des Systemausgangs ab. Jeder Regelkreis nach diesem Schema besteht immer aus den Komponenten

- Reglstrecke (plant)
- Sensor(en)
- Komparator
- Regler (compensator)
- Stellglied (actuator)

Zwischen diesen Komponenten sind die Größen

- $r(t)$ Führungsgröße, Sollwert
- $y(t)$ Regelgröße, Istwert

- $e(t)$ Regelfehler, Regeldifferenz
- $u(t)$ Stellgröße

definiert. Der Istwert $y(t)$ wird mit dem Sollwert $r(t)$ verglichen, der Regelfehler $e(t)$ ermittelt und auf den Regler geleitet. Der Regler verstärkt den Regelfehler in ein proportionales Stellsignal $u(t)$ für den Aktuator, welches dem Regelfehler entgegen wirkt. Damit steht eine Methode zur Verfügung, um die gestörte Bewegung eines WMR zu kontrollieren und zu korrigieren.

Der PID-Regler Der einfachste Mechanismus für einen Regler nach obigem Prinzip ist der PID. PID steht für Proportional-Integral-Differential-Regler.

Am Beispiel der Fahrzeuglenkung kann das PID-Verhalten sehr gut erläutert werden. Kommt es zu einer Abweichung von der Spur, so wird unmittelbar gegengelenkt und zwar proportional zur Höhe der Abweichung (P-Anteil). Dieses Gegensteuern bewirkt, dass die Driftbewegung des Fahrzeugs gestoppt wird. Um jedoch auf den ursprünglichen Kurs zurückzukehren, ist ein weiteres Manöver notwendig, welches den aufgetretenen Spurfehler beseitigt (I-Anteil). Dieses Korrekturmanöver verursacht einen entgegengesetzten Drift des Fahrzeugs. Um das Schlingern um die eigentliche Spur zu dämpfen, werden die Ist-Fehler zweier aufeinander folgender Zeitpunkte differenziert. Sinkt die Fehlerrate, so wird auch die Reaktion des Reglers kleiner und die Stellgröße schwingt sich auf einen festen Wert ein. Abbildung 1.3 zeigt den typischen Verlauf der Stellgröße eines idealen PID-Reglers.

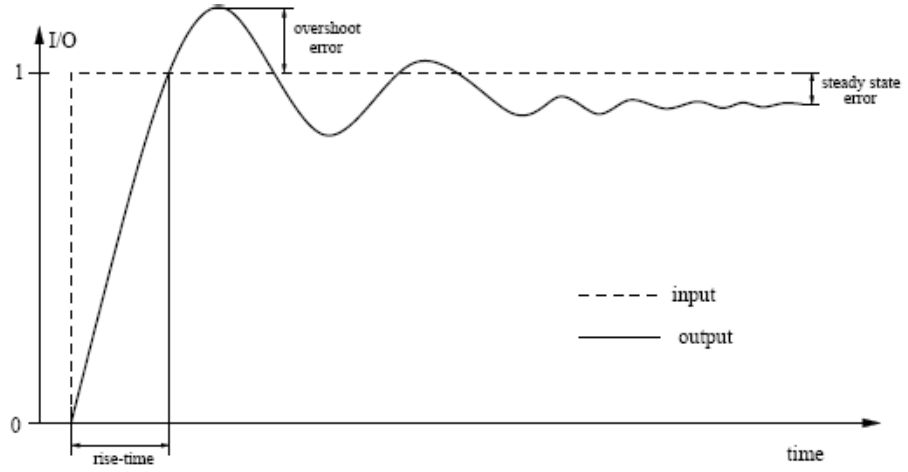


Abbildung 1.3.: Sprungantwort eines idealen PID-Reglers
Quelle: [Bir01]

Formal wird ein PID-Regler als lineares System mit der Grundblöcken P-Glied, I-Glied und D-Glied nach Gleichung 1.2.11 beschrieben.

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)d(t) + K_d \cdot \frac{d}{dt}e(t) \tag{1.2.11}$$

Die Konstanten K_p, K_i, K_d sind Verstärkungsfaktoren, welche die Reaktions- und die Ausregelzeit des PID-Reglers bestimmen.

Konstruktion eines Software-PID Um einen PID-Regler in Software zu realisieren wird der in Gleichung 1.2.11 für den zeitkontinuierlichen Fall beschriebene Regler nach Gleichung 1.2.12 diskretisiert.

$$u(k) = K_p \cdot e(k) + K_i \cdot \Sigma_{i=0}^{k} e(i) + K_d \cdot \triangle e(k) \qquad (1.2.12)$$
$$k = 1, 2, \dots \ , \ e(0) = 0$$

Die Ermittlung der Reglerkonstanten ist Teil des Konstruktionsprozesses. Praktisch ermittelt man diese bei *black-box*-Systemen experimentell nach der Ziegler-Nichols-Methode [LW98].[6] Der Regler kann als periodischer Prozess implementiert werden, welcher

1. den momentanen Regelfehler $e(k)$ ermittelt,
2. die internen Zustände $\Sigma_{i=0}^{k}$ und $e(k-1)$ aktualisiert,
3. die Stellgröße $u(k)$ berechnet und den Aktuator aufruft.

In Kapitel 1.3 wird eine Implementierung eines Software-PID zur Regelung der Motorleistungen beim DD vorgestellt.

1.2.5. Ableitung projektspezifischer Aufgaben

Anhand der Aufgabenstellung und den in diesem Kapitel erarbeiteten Grundlagen wurden für das weitere Vorgehen im Projekt folgende Teilaufgaben definiert.

1. Definition einer Hardware-Architektur.
   Das DD wird als WMR-Architektur verwendet, wobei auch spezielle Konzepte wie das DDD [Bau00] untersucht werden sollen. Dies ist motiviert durch die Tatsache, dass die unter LeJOS bereits vorhandene *Navigator*-Klasse bereits auf diesem Drive-Typ arbeitet und die im KI-Labor gebauten Lego-Roboter zumeist DDs sind.

2. Bau und Evaluierung von (D)DD-Prototypen.
   Beim Design sollten die in Abschnitt 1.2.3 angesprochenen Aspekte zur Minimierung von systematischen Fehlern beachtet werden. Die Evaluation soll die konstruktionsbedingten Vor-und Nachteile von DD und DDD untersuchen und die unbekannten Modellparameter experimentell bestimmen.

3. Erarbeitung eines Steuerkonzeptes zur Realisierung von Figuren nach Tabelle 1.1.

[6] Diese Methode betrachtet nur Systeme, bei welchen ein Überschwingen der Systemantwort zulässig ist.

4. Erarbeitung eines Konzeptes zur rein odometriebasierten Positionsschätzung.
5. Erarbeitung eines verbesserten Regelkonzeptes zur Kontrolle und Korrektur der Bewegung des DD.

Die Ergebnisse der Punkte 3-5 werden im folgenden Kapitel vorgestellt.

1.3. Konzept

Dieses Kapitel stellt das Konzept einer modularen Software-Architektur zur odometrischen Navigation zweirädriger WMRs auf Lego-Mindstorms-Basis mit LejOS-Kernel vor. Es hat den Namen lejON – LeJOS Odometric Navigator. Die Ziele bei der Entwicklung dieser Architektur sind:

1. Abstraktion eines zweirädrigen WMRs auf DD-Basis in eine Navigator-Schicht – der lejON-Schicht – mit einer erweiterbaren API (vgl. Abschnitt 1.3.1).

2. Implementierung einer rein odometriebasierten Positionsschätzung (vgl. Abschnitt 1.3.2).

3. Bereitstellung von Funktionen zur Realisierung von parametrisierbaren Figuren: Translation, Rotation und Kurve (vgl. Abschnitt 1.3.3).

4. Implementierung eines softwarebasierten Regelmechanismus zur Kontrolle und Korrektur der Bewegung (vgl. Abschnitt 1.3.4).

1.3.1. Die lejON-Schicht

In [KBM98, S.93,103ff.] wird eine abstrakte Schichtenarchitektur für mobile Indoor-Roboter beschrieben, welche sich grob in High-Level-*Planner* und Low-Level-*Navigation&Control* unterteilt. Dieses Konzept nutzt einen Command/Report-Mechanismus als Schnittstelle zwischen den Schichten, welche nach unten hin von den Details der Aktorik und Sensorik abstrahiert, nach oben hin Sensor-und Statusinformationen in aufbereiteter Form bereitstellt. Die Adaption für dieses Projekt liegt also nahe und führt zu dem in Abb. 1.4 dargestellten Architekturmodell.

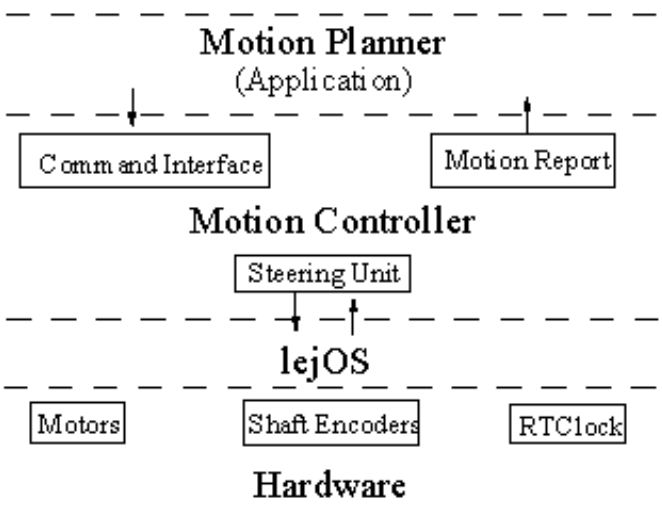


Abbildung 1.4.: Architektur der lejON-Schicht

Schnittstellen Die Kommunikation zwischen lejON und der Anwendung erfolgt über Nachrichtenblöcke. Das `Motion-Command` definiert eine Figur nach Tabelle 1.1, S. 8. Eine Folge solcher Kommandos definiert dann bestimmten Pfad. Solche Pfade sind dann auch wirklich realisierbar, da sie aus realisierbaren Pfadsegmenten zusammengesetzt sind. Im `Motion-Report` wird die geschätzte lokale Pose zusammen mit weiteren Statusinformationen bereitgestellt. Lokal bedeutet hier: Die Pose ist relativ zur Startpose der aktuellen Figur.

Motion Controller Die internen Aktionen vor, während und nach der Ausführung eines `Motion-Command` werden durch eine „state machine“ nach Abb. 1.5 koordiniert, die hier als `Motion-Controller` bezeichnet wird.

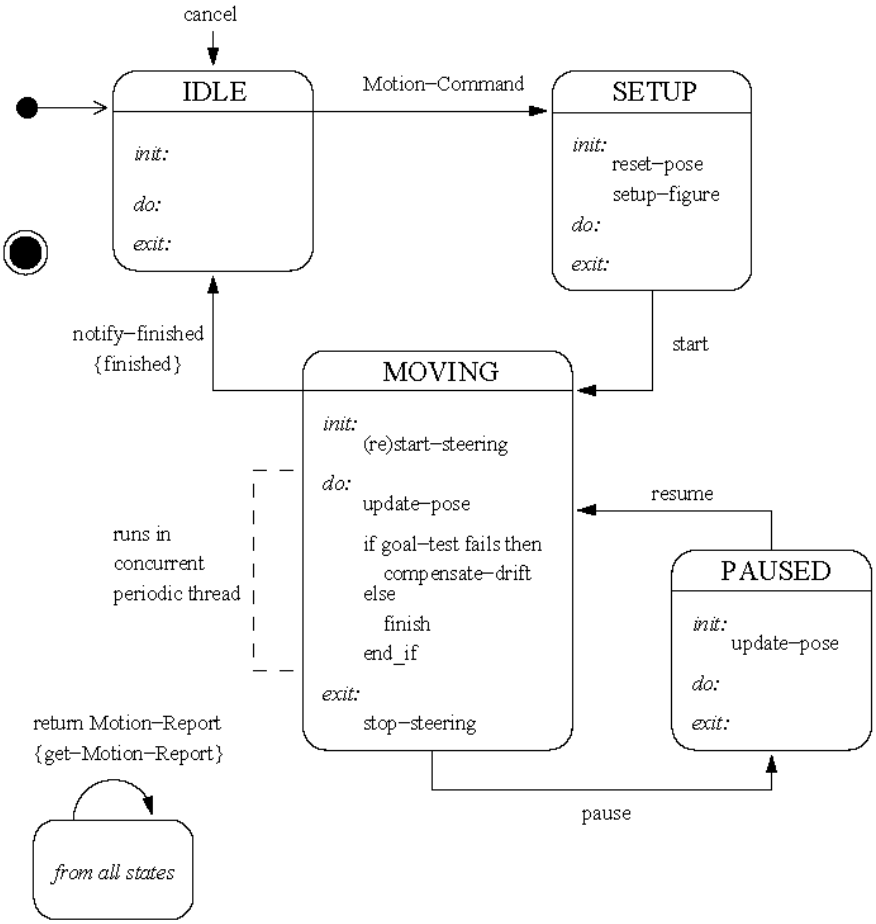


Abbildung 1.5.: Zustandsdiagramm des `Motion-Controller`

Steering Unit Die Schnittstelle zum LeJOS-Kernel und der Hardware des WMR ist die `Steering-Unit`. Sie stellt ein internes API zur Verfügung, welches in den Zuständen des `Motion-Controller` verwendet wird. Hier werden alle Aufgaben und

Funktionen gekapselt, welche mit der hardwareseitigen Vorbereitung, Durchführung und Terminierung einer Figur verbunden sind. Insbesondere die Positionsschätzung und die Fehlerkompensation sind hier implementiert. Ohne der eigentlichen Realisierung vorweg zu greifen, ist es notwendig, die beiden letztgenannten Aufgaben als periodische Tasks mit hinreichend hoher Frequenz zu realisieren.

Die lejON-Architektur ist vom Konzept her unabhängig vom WMR-Typ. Der typspezifische Teil dieser Architektur ist lediglich die `Steering-Unit`. Im Folgenden wird vom DD als WMR-Typ ausgegangen.

1.3.2. Implementierung der odometrischen Positionsschätzung

Um praktisch die Position eines sich bewegenden DDs nach Gleichung 1.2.4 zu berechnen, werden die Richtungswinkelgeschwindigkeit und die Referenzpunktgeschwindigkeit periodisch über die Rotationssensoren der Räder gemessen und aufsummiert.

Die Referenzpunktgeschwindigkeit v und Richtungswinkelgeschwindigkeit ω des DD zum Messzeitpunkt $n > 0$ ergeben sich analog zu Gleichung 1.2.2 und 1.2.3 zu

$$v(n) = ds(n) = \frac{1}{2} \cdot \frac{\mathrm{dTICKS_r} + \mathrm{dTICKS_l}}{\mathrm{TPC}}$$

$$\omega(n) = d\theta(n) = \frac{1}{l} \cdot \frac{\mathrm{dTICKS_r} - \mathrm{dTICKS_l}}{\mathrm{TPC}}$$

Dabei ist dTICKS das gemessene Inkrement der TICKS des Rades bezogen auf den Messzeitpunkt $n - 1$. Analog zu Gleichung 1.2.4 ergibt sich dann die Pose zum Messzeitpunkt n zu

$$\theta(n) = \sum_{i=1}^{n} d\theta(i)$$

$$x(n) = \sum_{i=1}^{n} ds(i) \cdot cos(\theta(i))$$

$$y(n) = \sum_{i=1}^{n} ds(i) \cdot sin(\theta(i))$$

□

1.3.3. Implementierung von Figuren

Ziel des in Abschnitt 1.3.1 vorgestellten Konzeptes ist es u. a. eine minimale Schnittstelle in Form des `Motion-Command` bereit zu stellen, welche dem Anwender ermöglicht,

definierte Figuren im WMR auszuführen. Das DD erlaubt die Translation, die Punktrotation (im Folgenden nur als Rotation bezeichnet) und die Kurvenfahrt mit konstantem Radius $R \geq l/2$, vgl. Kapitel 1.2.1 .

Die Figuren sollen parametrisierbar sein über Vorgabe von Distanzen und Winkeln in Einheiten [cm] und [°] relativ zur Startpose der Figur. Durch die Ausführung der Figur erreicht das DD eine neue Pose, die Zielpose dieser Figur. Für den `Motion-Controller` ist die Startpose einer Figur immer als $(0, 0, 0)$ definiert und die Zielpose (x, y, θ) der Offset, der die relative Verschiebung des Referenzpunktes in der Bewegungsebene und den neuen Richtungswinkel angibt.

Die Ausführung der Figur wird in der `Steering-Unit` kontrolliert. Da hier die Bewegung ausschließlich durch Odometrie gemessen wird, müssen die Parameterwerte der Figur in geeignete Zielwerte auf TICKS-Basis umgewandelt werden. Weiterhin müssen im Falle der Kurvenfahrt die Geschwindigkeiten der Räder so gewählt werden, dass sich der gewünschte Kurvenradius ergibt (siehe unten).

Eine Figur wird für die `Steering-Unit` eindeutig durch die in Tabelle 1.2 angegebenen Größen beschrieben. Diese *Steering-Parameter* sind vor dem Start einer neuen Figur vom `Motion-Controller` zu berechnen; dies findet im Zustand `SETUP` statt, vgl. Abb. 1.5.

Parameter	*Beschreibung*	*Definition*
G_l, G_r	Zielticks links/rechts	$G_x \geq 0$
D_l, D_r	Rotationsrichtung links/rechts	$D_x = \begin{cases} 1 & , forward \\ -1 & , backward \end{cases}$
P_l, P_r	Leistungsstufen	$P_x \in \{0, 1, 2, \ldots, 7\}$
v_d	Driftgeschwindigkeit	$v_d = \lvert v_r \rvert - \lvert v_l \rvert$

Tabelle 1.2.: Steering-Parameter des DD

In der unter LeJOS vorhandenen *Navigator*-Klasse sind nur die Translation und die Rotation implementiert. Hier war die Aufgabenstellung zusätzlich auch Kurven mit konstanten Radien als Figur `CURVE` zu implementieren. Dies gestaltet sich unter LeJOS bzw. der zur Verfügung stehenden Lego Mindstorms-Hardware als schwierig, da dem Anwender zur Erzeugung von verschiedenen Radgeschwindigkeiten nur 8 Motor-Leistungsstufen zur Verfügung stehen. Die Leistungsstufen repräsentieren außerdem keine Geschwindigkeiten (Motordrehzahlen) sondern Drehmomente [Bau00]. Somit sind differenzierte Geschwindigkeiten nur unter Last messbar bzw. realisierbar. Abbildung 1.7 auf Seite 26 zeigt eine experimentell für den DD-Prototypen DIDI ermittelte Geschwindigkeitskurve.

Leistungsstufenmodulation Um Radgeschwindigkeiten „zwischen“ den Nenngeschwindigkeiten der Leistungsstufen zu realisieren, wird hier eine Modulation der

Leistungsstufen angewendet, welche implizit über den im nächsten Abschnitt vorgestellten PID-Regler realisiert wird. Das Prinzip ist simpel. Bei `CURVE` wird ein Solldrift $v_d \neq 0$ vorgegeben (welcher mit den vorhandenen Leistungsstufen bzw. Nenngeschwindigkeiten nicht realisiert werden kann). Der Regler versucht dann diesen Sollwert zu halten, indem er die Motorleistungen um die Anfangswerte herum moduliert.

Berechnung der `CURVE`-Einstellungen Sei $v(P)$ die Geschwindigkeitsfunktion des DD nach Abb. 1.7 und $R \geq l/2$, $\theta \neq 0$ und P_{max} die vorgegebenen Figurparameter nach Tabelle 1.5. Wir betrachten den Fall der Linkskurve ($\theta > 0$); der Fall der Rechtskurve ergibt sich analog. Aus Gleichung 1.2.1 und 1.2.2 kann man das Verhältnis der Radgeschwindigkeiten für die Kurvenfahrt mit R wie folgt ableiten.

$$\frac{v_r}{v_l} = \begin{cases} \frac{1-\alpha}{1+\alpha} & , \; v_r < v_l \\ \frac{1+\alpha}{1-\alpha} & , \; v_r > v_l \end{cases} \qquad \alpha = \frac{l}{2R} \tag{1.3.1}$$

Wähle nun die Geschwindigkeit des rechten (äußeren) Rades maximal zu $v_r = v(P_{max})$. Bestimme dann die Geschwindigkeit v_l des linken (inneren) Rades nach Gleichung 1.3.1. Als Anfangswerte der Leistungsstufen werden $P_r = P_{max}$ und $P_l = argmin_P \{ |v(P) - v_l| \}$ gewählt. Als Solldrift wird $v_d = v_r - v_l$ gewählt.

Die Tabellen 1.3 bis 1.5 zeigen, wie die in Tabelle 1.2 allgemein definierten Steering-Parameter konkret für die Figuren `TRANSLATE`, `ROTATE` und `CURVE` aus den vom Anwender vorgegebenen Parametern berechnet werden.

1.3.4. Implementierung eines softwarebasierten PID-Reglers

Das in Kapitel 1.2.4 allgemein vorgestellte Konzept von PID-Reglern wird hier zur Fehlerkompensierung bei der Bewegung des DD angewendet. Für die Implementierung eines PID in Software gibt es verschiedene Ansätze, siehe dazu [CMM03]. Der einfachste und effektivste Ansatz ist *ein* PID für beide Räder, welcher die Driftgeschwindigkeit v_d kontrolliert. Erfolgreiche Projekte, die diesen Ansatz nutzen, sind u. a. [CMM03, Luc01]. Eine anspruchsvollere Variante wird in [And98] vorgestellt.

Das Prinzip des hier vorgestellten Ansatzes soll am Beispiel der Geradeausfahrt erläutert werden. Die Soll-Driftgeschwindigkeit v_d^* ist in diesem Fall $v_d^* = 0$. Die Ist-Driftgeschwindigkeit v_d zum Zeitpunkt t kann über die Rotationssensoren als $v_d = \text{dTICKS}_\text{r} - \text{dTICKS}_\text{l}$ bestimmt werden (vgl. S. 19). Der zu kontrollierende Regelfehler zum Messzeitpunkt ist also

$$e = v_d^* - (\text{dTICKS}_\text{r} - \text{dTICKS}_\text{l}) \tag{1.3.2}$$

Ist beispielsweise $e < 0$, so eilt das rechte Rad dem linken voraus. e wird nun beseitigt, indem das linke Rad beschleunigt oder das rechte Rad abgebremst wird.

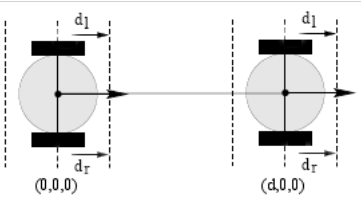
`TRANSLATE`	
Figurparameter:	$d > 0$ Distanz in Fahrtrichtung in [cm]
	P_{max} max. Leistungsstufe
Zielpose:	$(d, 0, 0)$
Figur:	
Steering-Parameter:	$G_l = G_r = d \cdot \text{TPC}$
	$P_l = P_r = P_{max}$
	$D_l = D_r = 1$
	$v_d = 0$

Tabelle 1.3.: Realisierung von `TRANSLATE`

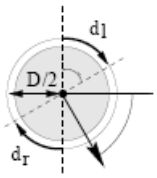
`ROTATE`	
Figurparameter:	$\theta \neq 0$ Richtungswinkel in [°]
	P_{max} max. Leistungsstufe
Zielpose:	$(0, 0, \theta)$
Figur:	
Steering-Parameter:	$G_l = G_r = \theta \cdot \text{TPD}$
	$P_l = P_r = P_{max}$
	$D_l = \begin{cases} -1 & \text{falls } \theta > 0 \\ 1 & \text{falls } \theta < 0 \end{cases}$
	$D_r = -D_l$
	$v_d = 0$

Tabelle 1.4.: Realisierung von `ROTATE`

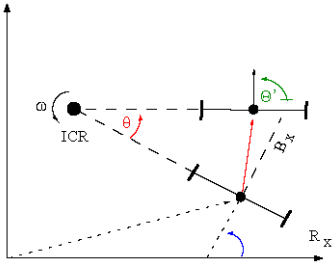
`CURVE` (vgl. Erläuterungen auf S. 21)	
Figurparameter:	$R \geq l/2$ Kurvenradius in [cm] $\theta \neq 0$ Richtungswinkel in [°] P_{max} max. Leistungsstufe
Zielpose:	$(R \cdot sin\,\theta,\ R \cdot (1 - cos\,\theta),\ \theta)$
Figur:	
Steering-Parameter:	$G_l = \begin{cases} \pi(R + l/2) \cdot \frac{\theta}{180} \cdot \text{TPC} & \text{falls } \theta < 0 \\ \pi(R - l/2) \cdot \frac{\theta}{180} \cdot \text{TPC} & \text{falls } \theta > 0 \end{cases}$ G_r analog
	$P_l = \begin{cases} P_{max} & \text{falls } \theta < 0 \\ < P_{max} & \text{falls } \theta > 0 \end{cases}$ P_r analog
	$D_l = D_r = 1$
	$v_d \neq 0$

Tabelle 1.5.: Realisierung von `CURVE`

Das Abbremsen und Beschleunigen realisiert die Stellfunktion

$$actuate : u \mapsto (P_l, P_r) \tag{1.3.3}$$

mit dem durch den PID berechneten Stellsignal u als Input (vgl. S. 14, Gleichung 1.2.11). Die Arbeitsweise von $actuate()$ ist in Algorithmus 1.3.4 abstrakt dargestellt. Die vielen Fallunterscheidungen tragen der Tatsache Rechnung, dass hier nur ein PID verwendet wird. Somit wird, je nach Vorzeichen des Sollwertes v_d^*, immer nur *ein* Rad geregelt. Im Falle der Kurvenfahrt ist es immer das innere Rad, was ja, wie in Abschnitt 1.3.3 erläutert, nötig ist. K_r ist ein Proportionalfaktor, welcher die Stellgröße u (des PID) in einen entsprechenden Leistungsoffset dP skaliert (siehe S. 26). dP wird dann zum Abbremsen bzw. zum Beschleunigen des geregelten Rades verwendet, indem die Motorleistung angehoben bzw. erniedrigt wird.

Algorithmus 1:
/* compute power offset against u */
$dP = K_r \cdot u$;

IF $v_d^* < 0$ **THEN**
  IF $dP < 0$ **THEN**
    /* compensate right drift */
    $accelerate{-}right(|dP|)$;
  ELSE
    /* compensate left drift */
    $brake{-}right(dP)$;
  END IF
ELSE
  IF $dP < 0$ **THEN**
    /* compensate left drift */
    $accelerate{-}left(|dP|)$;
  ELSE
    /* compensate right drift */
    $brake{-}left(dP)$;
  END IF
END IF

Ermittlung des Regelstreckenmodells Abbildung 1.6 zeigt das Schema eines Regelkreises.

Regelkreise können im Allgemeinen nur dann berechnet werden, wenn die Funktionsgleichungen der Regelkreiselemente bekannt sind. Zur Berechnung werden daher Kenntnisse über den Typ und die Parameter der Regelstrecke benötigt. Der Typ bestimmt das qualitative -, die Parameter das quantitative Regelstreckenmodell.

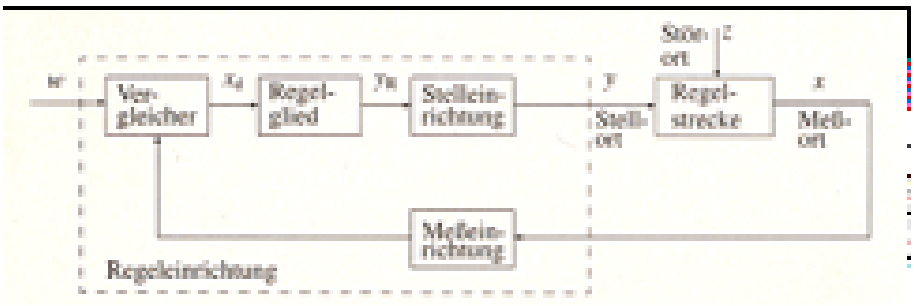


Abbildung 1.6.: Regelkreisschema
Quelle: [LW98]

Die Bestimmung von Typ und Parametern der Strecke wird als *Identifizierung* bezeichnet. Die Identifizierung muss bei fehlendem theoretischen Modell experimentell durch Messung des Input-Output-Verhaltens der Regelstrecke, dem *Übertragungsverhalten*, erfolgen. Die hier angewendete Vorgehensweise wird in [LW98, S. 295ff.] ausführlich beschrieben.

1. Experimentelle Ermittlung des Übertragungsverhaltens
2. Analyse und Ermittlung einer geeigneten Modellstruktur
3. Ermittlung der Modellparameter und Approximation des Übertragungsverhaltens durch die Modellfunktion
4. Modellinvertierung und Berechnung der Stellfunktion
5. Simulation des Regelkreises und Ermittlung der PID-Konstanten

Das messenbare Übertragungsverhalten ist in unserem Fall durch die Funktion $v : p \mapsto v(p)$ charakterisiert, wobei p die Leistungsstufe und $v(p)$ die messbare Geschwindigkeit des DD bzgl. einer Zeitbasis T_p ist.

Messaufbau Die Regelstrecke ist der in Kapitel 2.1.2 vorgestellte DD-Prototyp DIDI, in welchem der Software-PID später implementiert wird. Alle Messungen wurden unter Normalbedingungen (volle Batterie und ebener, fester Untergrund) durchgeführt. Wegen der geringen Auflösung der Rotationssensoren erfolgten die Messungen mit einer großen Periode von T_p=500 ms und 1000 ms, um differenzierte Messergebnisse zu erhalten. Es wurden 50 bzw. 25 Messwerte je Rad und Leistungsstufe erfasst und jeweils die Mittelwerte gebildet.[7]. Der Vergleich der Mittelwerte bei T_p=500 ms und 1000 ms zeigte lineare Abhängigkeit von der gewählten Zeitbasis. Somit können Mittelwerte für andere Zeitbasen, z. B. T_p=200 ms, durch Skalieren

[7]Die Messwerte haben die Dimension [TICKS]. Durch Bezug der Messwerte auf die Messperiode ergibt sich die Dimension (einer Geschwindigkeit) [TICKS/ms].

der Messwerte berechnet werden. Abbildung 1.7 zeigt den gemessenen Geschwindigkeitsverlauf bzw. das Übertragungsverhalten der Regelstrecke.

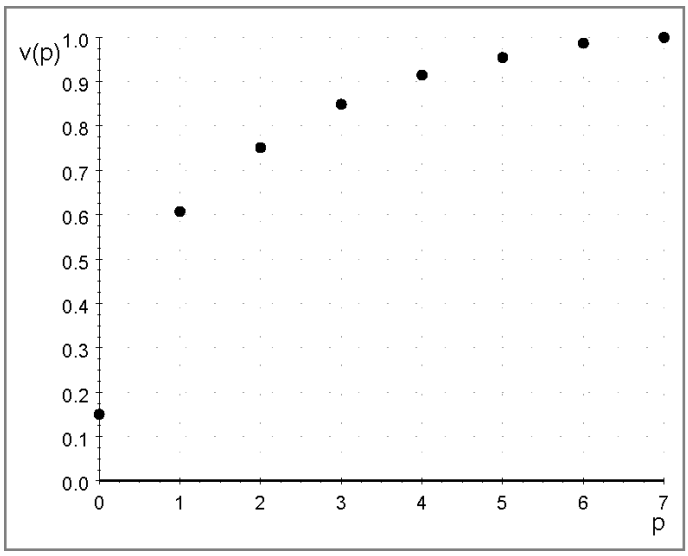


Abbildung 1.7.: Geschwindigkeitsverlauf DIDI (normiert)

Identifizierung der Strecke Der Geschwindigkeitsverlauf zeigt PPT1-Verhalten [LW98, S. 139 f.], wobei wir hier die Leistung und nicht die Zeit als Abszisse verwenden. Deshalb sind die im Folgenden berechneten Streckenparameter T_1 und T_v eigentlich Leistungen und keine Zeiten. Wir können also als Modell für die Übertragungsstrecke die Funktion der PPT1-Strecke verwenden.

$$v(p) = K_s \cdot \left[1 - (1 - \frac{T_v}{T_1}) \cdot e^{-\frac{p}{T_1}}\right] \tag{1.3.4}$$

Die Parameter des Modells sind T_1, T_v und K_s. T_1 ist die *Verzögerung* der Strecke – hier die Leistung, bei welcher ≈63% v_{max} erreicht wird. T_v ist der *Vorhalt* – hier der Leistungsoffset, bei welchem $v = 0$.[8] T_1 und T_v werden über die Tangente an $v(0)$ ermittelt (vgl. Abb. 1.8). K_s ist der Output der Strecke bei Sättigung – hier also v_{max}. Tabelle 1.6 zeigt die für DIDI ermittelten Parameterwerte. Abbildung 1.8 zeigt die mit diesen Parametern approximierte Streckenfunktion .

Der nächste Schritt ist die Berechnung der Stellfunktion durch Invertierung der Streckenfunktion. Es wird die Funktion $K_r|_A : \Delta v \mapsto \Delta p$ ermittelt, welche den gemessenen Regelfehler im Arbeitspunkt A für kleine Werte von Δv in eine proportionale Leistungsdifferenz Δp „zurückrechnet“. Diese wird dann vom Stellglied

[8] In der LeJOS-Motor-API ist $v(p = 0) > 0$!

PPT1-Parameter für „DIDI“	
$T_1 = 1.7371$	$v(T_1) \approx 0.63 \cdot v_{max}$
$T_v = 0.2786$	$v(-T_v) = 0$
$K_s = 15.3$	$K_s = v_{max}$, bei $T_p = 200ms$

Tabelle 1.6.: Regelstreckenparameter für „DIDI“

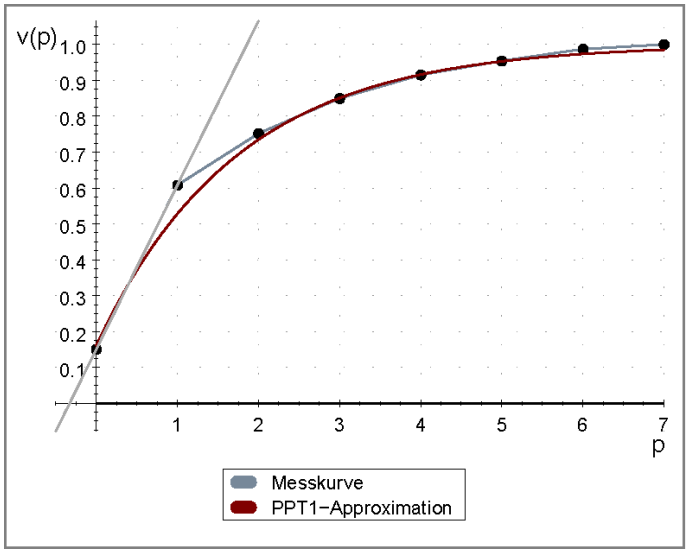


Abbildung 1.8.: Approximierter PPT1-Verlauf (normiert)

kompensiert; man sagt hier: der Regelfehler wird verstärkt und bezeichnet K_r als *Regelverstärkung*. Die Berechnungsmethode linearisiert die Streckenfunktion $v(p)$ nach Gleichung 1.3.4 in A durch eine Taylorreihen-Zerlegung mit Abbruch nach dem 1. Glied, vgl. [LW98, S. 52 ff.].

$$K_r|_A = \left.\frac{\Delta p}{\Delta v}\right|_{A=p} \approx \frac{1}{\dot{v}(p)} = K_{r_0} \cdot e^{\frac{p}{T1}} \tag{1.3.5}$$

Abbildung 1.9 zeigt den Verlauf der Regelverstärkung für verschiedene Arbeitspunkte.

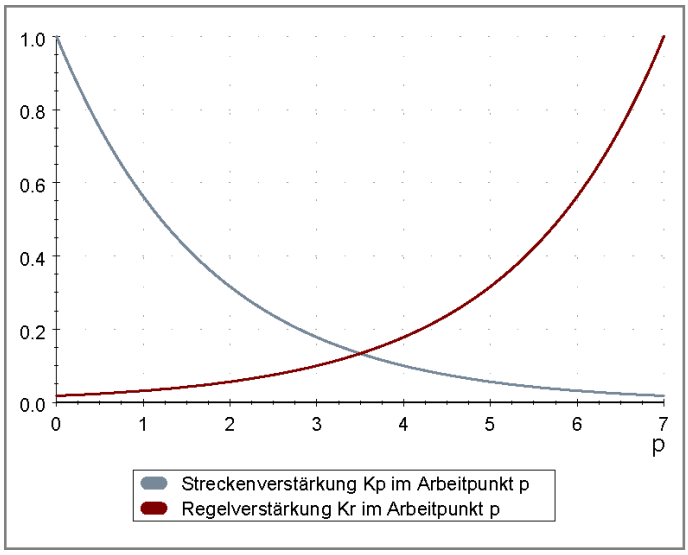


Abbildung 1.9.: Arbeitspunktabhängige Regelverstärkung (normiert)

Architektur des PID-Reglers Der Regelkreis besteht aus 4 Modulen bzw. Funktionen.

1. Odometer, *update*()
   Ermittlung des Wegdrifts der Räder (Istwert)

2. Comparator, *compare*()
   Ermittlung des Regelfehlers e nach Gleichung 1.3.2

3. Compensator, *compensate*(e)
   Erzeugung des Stellsignals u

4. Actuator, *actuate*(u)
   Einstellen der Motorleistungen nach Algorithmus 1.3.4

Der Signalfluss zwischen diesen Komponenten ist in Abbildung 1.10 dargestellt.

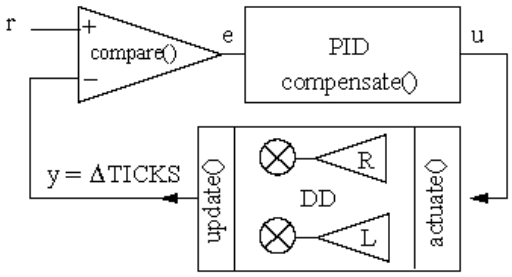


Abbildung 1.10.: Architektur des PID-Reglers

Simulation Um das Reglermodell und die ermittelten Parameter vor der eigentlichen Implementierung zu überprüfen, wurde ein Simulationsprogramm unter MuPAD[9] erstellt, welches die Regelstrecke mittels $v(p)$ modelliert und die oben vorgestellte PID-Architektur implementiert. Abbildung 1.11 zeigt zwei Simulationsergebnisse.

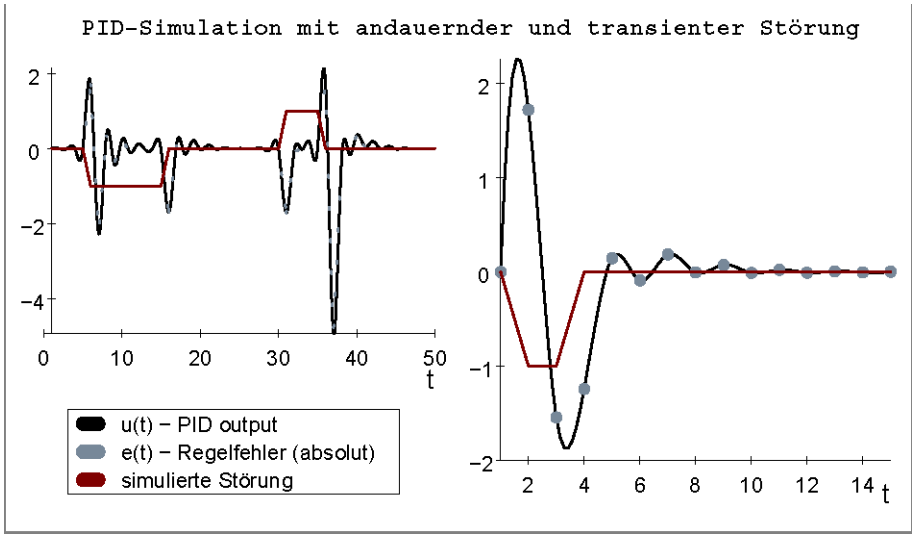


Abbildung 1.11.: PID-Simulation

Man kann erkennen, dass bei Auftreten einer Störung (simulierter Drift) eine proportionale PID-Reaktion erfolgt (Gegenlenken), so dass der momentane Drift gestoppt wird. Weiterhin stellt sich der PID so ein, dass die entstandene Abweichung (in Form des Absolutfehlers) wieder ausgeregelt wird.

Die hier verwendeten PID-Konstanten K_p, K_i und K_d wurden für das Simulationsprogramm wie folgt bestimmt. Die Stellgröße eines PID kann analog zu Gleichung 1.2.11 auch als

$$u(t) = K_r \left[e(t) + \frac{1}{T_N} \int e(t)\, dt + T_v \frac{\delta e(t)}{\delta t} \right] \tag{1.3.6}$$

[9] http://www.sciface.com/

dargestellt werden, vgl. [LW98, S.162]. K_r wird vom Aktuator nach Gleichung 1.3.5 berechnet. T_N ist die Nachstellzeit, T_v der Vorhalt. Die Werte von T_v und T_n sollten ungefähr mit den oben ermittelten Streckenparametern T_1 und T_v übereinstimmen. Tabelle 1.7 zeigt die in der Simulaltion verwendeten PID-Konstanten.

PID-Konstanten der Simulation	
K_p	1
K_i	0.57
K_d	0.15

Tabelle 1.7.: PID-Konstanten der Simulation

Bemerkung 1.3.1. In der Simulation wird *nicht* berücksichtigt, dass nur ganze Leistungsstufen zur Verfügung stehen. In der Zielimplementierung wird es deshalb durch das Runden der Stellgröße im Actuator zu einem etwas unruhigeneren bzw. ungenaueren Regelverhalten kommen. Ggf. ist eine Feinabstimmung der Konstanten nötig.

Zusammenfassung In diesem Kapitel wurden die Kernbestandteile von lejON

- Systemarchitektur
- Realsierung der odometrischen Positionsschätzung
- Realisierung von Figuren
- Kontrolle und Korrektur der Bewegung

konzeptuell vorgestellt. Die hardware- und softwareseitige Umsetzung wird in den folgenden zwei Kapiteln beschrieben.

Bemerkung 1.3.2. Die bestehende Implementierung des PID weicht (noch) von dem hier vorgestellten Konzept ab! Es wurde erst zum Ende des Projekts wirklich klar, dass eine Approximation der Strecke durch das PPT1-Modell möglich ist. Die aktuelle Implementierung verwendet ein einfacheres, lineares Streckenmodell und einen etwas anderen Stellalgorithmus – wobei auch damit schon relativ gute Ergebnisse erzielt werden (vgl. Kapitel 4 und 5). Durch die Simulationen konnte jedoch die Machbarkeit des obigen Konzepts zumindest theoretisch gezeigt werden. Eine Ersetzung der alten PID-Implemetierung hat auf die anderen Bestandteile von lejON keine Auswirkungen. Lediglich die PID spezifischen Parameter müssen geändert und erweitert werden. Eine Umsetzung des obigen Konzeptes in der bestehenden Implementierung würde auf jeden Fall eine Verbesserung des Regelverhaltens hinsichtlich der Genauigkeit und der Reaktionszeiten bedeuten.

2. Architektur

Die Architektur des zweirädrigen Roboters für dieses Projekt soll eine sichere und zuverlässige Navigation erlauben. Dafür ist es erforderlich, möglichst viele Störfaktoren bereits in der Planungsphase zu beseitigen. Insbesondere soll der Roboter ohne Drift geradeausfahren können. Dies ist bei herkömmlichem Antrieb über zwei Motoren, die mit jeweils einem Antriebsrad verbunden sind, ohne Software-Regelung nicht möglich. Hierbei spielen die unterschiedlichen Motorenbeschaffenheiten eine zu große Rolle. Eine mögliche Lösung für dieses Problem bietet ein so genanntes *dual differential drive* (DDD).

Die für dieses Projekt in Erwägung gezogenen Architekturen sind zum einen das *(Single) Differential Drive* und das *Dual Differential Drive*. Letzteres lehnt sich an die in [Bau00, S. 237ff] vorgestellte an. Darin wird ein mit Lego-Technic gebautes DDD beschrieben. Die für dieses Projekt realisierte Architektur ist jedoch bislang das *DD*.

2.1. Differential Drive

2.1.1. Allgemeine Architektur

Bei einem *(Single) Differential Drive* (DD) werden die Antriebsräder nicht miteinander verbunden sondern von jeweils einem Motor betrieben. Infolgedessen ist sowohl die Geradeaus- als auch die Kurvenfahrt von beiden Motoren gleichermaßen abhängig. Dieses Verhalten ist deswegen ein Problem, weil durch kleinste Abweichungen im Fertigungsprozess, durch Materialfehler oder durch Verschleiß nicht gewährleistet werden kann, dass sich – trotz gleichmäßiger Ansteuerung – die Motoren mit exakt demselben Drehmoment drehen.

2.1.2. DIDI als Prototyp eines DD

Der Prototyp, der in diesem Projekt zu Einsatz kommt, heißt *DIDI*. *DIDI* ist die Abkürzung für *(Single)* ***D****ifferential* ***D****rive*.

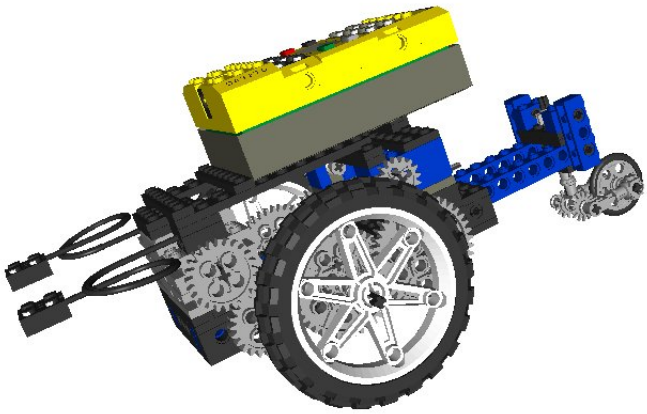


Abbildung 2.1.: DIDI, Prototyp eines Differential Drive

Die getrennt voneinander arbeitenden Antriebsräder werden von jeweils einem Motor über Zahnräder angetrieben. Die Rotationssensoren sind mit den Antriebsachsen ebenfalls über Zahnräder verbunden. Die Übersetzungsverhältnisse Motor-Antriebsachse und Antriebsachse-Rotationssensor können auf einfache Weise verändert werden. Das Stützrad befindet sich in einiger Entfernung zum Fahrzeugschwerpunkt. Seine Aufhängung kann sowohl in der Höhe als auch in der Entfernung zum Schwerpunkt leicht geändert werden. Dies ist dann besonders sinnvoll, wenn z. B. die Antriebsräder durch kleinere ersetzt werden.

2.2. Dual Differential Drive

2.2.1. Allgmeine Architektur

Bei einem solchen Antriebssystem hängen die Antriebsräder nicht mehr an einer einzigen Achse, sondern an zwei mit einem Differenzial direkt und einem zweiten indirekt verbundenen. Ein DDD sorgt dafür, dass ein Motor den gemeinsamen Antrieb beider Achsen in eine Richtung übernehmen kann; dies geschieht über das primäre Differenzial. Ein weiterer Motor am sekundären Differenzial sorgt für gegensätzliches Drehen der Antriebsachsen mit gleicher Geschwindigkeit.

Soll der Roboter also geradeaus fahren, wird nur das primäre Differenzial benötigt. Bei Kurvenfahrten wird dann das sekundäre zugeschaltet oder allein betrieben – je nachdem, ob sich der Roboter auf der Stelle drehen oder in einem größeren Radius fahren soll.

Die beiden Differenziale sind einzeln wie herkömmliche Differenzialgetriebe zu sehen: Die Leistung wird dem Gehäuse in der Mitte zugeführt, so dass die Achsen

sich mit derselben Geschwindigkeit wie die des Ausgleichgehäuses drehen. (vgl. /1/, S. 238)

Verbunden werden die beiden Getriebe durch Zahnradübersetzungen an ihren beiden Achsen. Dabei ist die Anzahl an Zahnrädern auf der einen Seite gerade, auf der anderen ungerade. Dies bewirkt, dass die Achsen am sekundären Differenzial in entgegengesetzter Richtung gedreht werden, wenn am primären die beiden Halbachsen in gleiche Richtung angetrieben werden. Durch das DDD besitzt der Roboter nun die Möglichkeit, einen korrekten Geradeauslauf zu vollziehen, ohne dabei zur Seite wegzudriften. Würde je ein Motor für ein Antriebsrad benutzt, könnte durch unterschiedliche Motorenleistung nicht gewährleistet sein, dass das Fahrzeug tatsächlich auch geradeaus fährt.

2.2.2. HILDE als Prototyp eines DDD

HILDE ist der Prototyp dieses Projektes für ein DDD. *HILDE* steht für ***H****igh-****I****ntelligence* ***L****ego* ***D****ual Differential Drive* ***E****ngine*.

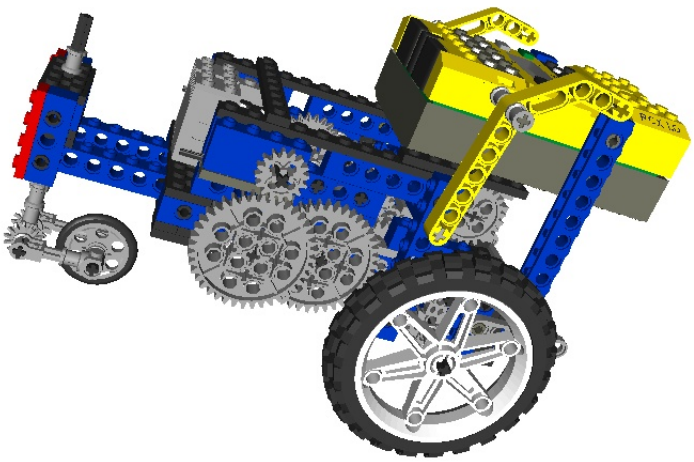


Abbildung 2.2.: HILDE, Prototyp eines DDD

Die Anordnung der Rotationssensoren ähnelt der von *DIDI*. Bei *HILDE* kann allerdings nur das Übersetzungsverhältnis Antriebsachse-Rotationssensor geändert werden. Durch die Eigenschaften des DDD ist das Verhältnis Motor-Antriebsachse unveränderbar. Direkt am primären Differenzial befinden sich die Antriebsachse und darüber der „primäre“ Motor. Über dem sekundären Differential ist der „sekundäre“ Motor angebracht. Aus diesem Grund ist der schwere RCX-Baustein auch (weg vom sekundären Differential) über den Fahrzeugmittelpunkt gerutscht als bei *DIDI*. Das Stützrad ist, genau wie bei *DIDI*, in Höhe und Entfernung zum Fahrzeugschwerpunkt variabel.

2.3. Einschränkungen und Berücksichtigungen

Die o. g. allgemeine Architektur kann nicht ohne Weiteres mit Lego-Technic-Bauteilen konstruiert werden. Hierzu müssen die Grenzen der eingesetzten Materialien berücksichtigt werden.

Da sich ein Kettenantrieb mit Lego aus anderen Projekten heraus als nicht geeignet für eine präzise Navigation herausstellte, ist für dieses Projekt ein dreirädriger Roboter vorgesehen. Somit muss das Design aus [Bau00] an drei Räder angepasst werden. Dabei ist zu beachten, dass der Roboter über zwei Räder angetrieben wird und das dritte als Stützrad nutzt. Letzteres muss aber von zu großer Last befreit werden, damit seine Drehung bei Richtungsänderung des Roboters nicht die gesamte Konstruktion dreht und somit die Fahrtrichtung negativ beeinflusst.

Aus diesem Grund ist der schwerste Baustein, der RCX selbst, direkt über der Antriebsachse angebracht, so dass er auf das Stützrad möglichst keinen Druck ausübt. Das Stützrad lässt sich somit fast frei bewegen und passt sich (passiv) der Bewegungsrichtung des Roboters an. Zudem ist eine Aufhängung des Stützrades angebracht, die es ermöglicht, den Radstand sowie die Höhe des Stützrades für weitere Tests zu variieren.

Das Eigengewicht ist allein durch den RCX und die Batterien sehr hoch. Deswegen sollte darauf verzichtet werden, den Roboter mit übermäßig breiten Reifen oder mit hoher Profilstärke fahren zu lassen, um möglichst wenig Reibung auf dem Untergrund zu erzeugen. Generell lässt sich allerdings auch solch eine Konfiguration realisieren.

Berücksichtigt werden muss ebenso, dass (fast) alle Bauteile des Roboters aus Kunststoff bestehen und nur eine eingeschränkte Stabilität bieten. Insbesondere ist festzustellen, dass sich gerade die Antriebsachsen je nach Gewicht des Roboters stark durchbiegen und eine präzise Navigation beeinflussen.

Zur Überprüfung, ob die angetriebenen Achsen auch die gewünschte Drehgeschwindigkeit erhalten, die ihnen die Motoren vorgeben, ist an jedem Antriebsrad je ein Rotationssensor angebracht. Da allein schon die Übersetzung durch Zahnräder mit einem gewissen „Spiel“ verbunden ist, wird das präzise Auslesen bei geringen Geschwindigkeiten besonders wichtig. Architekturbedingt können die Rotationssensoren jedoch nicht direkt an der Antriebsachse angreifen, so dass eine Übersetzung notwendig wird.

Viele Störfaktoren lassen sich somit schon während der Fahrzeugkonstruktion minimieren. Der Idealfall, nämlich die exakte Geradeausfahrt ohne Kursabweichung, lässt sich aber dennoch nicht ohne Software-Regelung realisieren – auch beim *DDD* ist dies der Fall.

3. lejON-API

Um dem Anwender von lejON eine Schnittstelle zu bieten und die Nutzung von lejON für beliebige in Kapitel 2 beschriebene Architekturen erweitern und nutzen zu können, wurde die lejON-API entwickelt. Die Schnittstelle zum Anwender bietet – den Anforderungen aus der Aufgabenstellung entsprechend – Methoden zum Fahren bestimmter Figuren:

- Translate – eine gerade Strecke vorgegebener Länge
- Rotate – eine Rotation um einen vorgegebenen Winkel
- Curve – eine Kurve mit vorgegebener Bogenlänge und Radius

Im Folgenden wird die Entwicklung der Schnittstelle über die Definition von Anwendungsfällen bishin zur Analyse von Aktionen und damit der Grundlage der erfolgten Implementierung für die DIDI-Architektur in Abschnitt 3.1 beschrieben. Weiterhin erfolgt in Abschnitt 3.2 eine Einordnung der lejON-API in ein Schichtenmodell, welches die Trennung zur Anwendungs- und zur Hardware-Schicht aufzeigt. In Kapitel 3.3 wird dann der konkrete Aufbau und Implementierung der lejON-API aus den verschiedenen Klassen und damit auch die Möglichkeiten der Erweiterung von lejON für eigene Architekturen erläutert.

3.1. Definition von Anwendungsfällen

Im Rahmen der Entwicklung der Anwenderschnittstelle von lejON wurden Anwendungsfälle identifiziert, die die Vorbereitung, Ausführung, Unterbrechung und Auswertung einer Figur beschreiben. Ausgehend von diesen Anwendungsfällen werden später Methoden der Anwendungsschnittstelle festgelegt und darin auszuführende Aktionen abgeleitet.

Beschreibung Anwendungsfall „set up a motion“:	
Name	„set up a motion“
Kurzbeschreibung	Anwender erzeugt ein Kommando für die gewünschte Figur mit entsprechenden Parameter und sendet das Kommando an das Drive zur Vorbereitung der Figur
Akteure	Anwendungsprogramm/User
Auslöser	Anwender möchte eine Figur ausführen
Ergebnis	Das Kommando für die Figur ist verarbeitet und die Figur kann gestartet werden.
Eingehende Daten	Parameter für Figur
Vorbedingungen	Es wird derzeit keine Figur ausgeführt, pausiert oder vorbereitet.
Nachbedingungen	Die Figur ist vorbereitet und kann gestartet werden.

Essenzielle Schritte	1. Erstellung eines Kommandos 2. Nachricht mit dem erzeugten Kommando an das Drive 3. Prüfen des Kommandos auf Gültigkeit 4. Wechsel des inneren Zunstands des Drives von „Idle“ nach „Setup“ 5. Vorbereitung der Figur durch das Drive entsprechend der Kommando-Parameter z. B. Berechnung der Motorleistungsstufen beider Räder für eine Kreisfahrt a) Berechnung der Leistungsstufen der Motoren b) Festlegung der Laufrichtung der Motoren c) Berechnung des Sollwertes für den Regler d) Berechnung der Zielticks für Beendigung der Figur 6. Nachricht an die Hardwareschicht zur Vorbereitung der Figur (setup) und damit Konfiguration der Hardware a) Setzen der Motorleistungen b) Setzen der Laufrichtungen der Motoren c) Setzen des Sollwertes des Reglers
Ausnahmen/Fehlersituationen	Der innere Zustand des Drives ist nicht „Idle“ – das Kommando kann somit nicht angenommen werden. Eine laufende Figur muss gegebenenfalls abgebrochen und die neue Figur vorbereitet werden.

Beschreibung Anwendungsfall „start a motion“:	
Name	„start a motion“
Kurzbeschreibung	Anwender startet die Figur für ein vorher abgesetztes Kommando (Translation, Rotation oder Kurvenfahrt), ist die Figur abgearbeitet, wird ein Ereignis generiert
Akteure	Anwendungsprogramm/User
Auslöser	Anwender möchte eine Figur ausführen
Ergebnis	Figur wird gestartet und die Anwendung bei Beendigung der Figur benachrichtigt
Eingehende Daten	Nachricht zum Starten der Figur
Vorbedingungen	Kommando für die Figur wurde erzeugt (Angabe der Parameter und der Art der Figur)
Nachbedingungen	Figur wird ausgeführt
Essenzielle Schritte	1. Benachrichtigung des Drives, die Figur zu starten. 2. Wechseln des inneren Zustands der Roboterarchitektur von „Setup“ nach „Moving“ 3. Nachricht an die Hardwareschicht zum Starten der Figur (move) 4. Starten der Figur durch die Hardwareschicht a) Starten der Motoren b) Start der Regelung c) Periodischer Vergleich der gemessenen Ticks mit den Zielticks d) Bei Übereinstimmung der gemessenen Ticks mit den Zielticks ist die Figur beendet und ein entsprechendes Ereignis wird ausgelöst

Ausnahmen/Fehlersituationen	Der innere Zustand des Drives ist nicht „Setup“ – das Kommando kann somit nicht gestartet werden. Eine laufende Figur muss gegebenenfalls abgebrochen und die neue Figur vorbereitet werden.

Beschreibung Anwendungsfall „pause a motion“:	
Name	„pause a motion“
Kurzbeschreibung	Eine laufende Figur wird angehalten
Akteure	Anwendungsprogramm/User
Auslöser	Anwender möchte eine Figur pausieren
Ergebnis	Figur wird angehalten
Eingehende Daten	Nachricht zum Pausieren der Figur
Vorbedingungen	Eine Figur wurde gestartet und wird noch ausgeführt
Nachbedingungen	Figur ist angehalten
Essenzielle Schritte	1. Wechsel des inneren Zustands des Drives von „Moving“ zu „Paused“ 2. Nachricht an die Hardwareschicht zum Pausieren der Figur (pause) a) „Floaten“ der Motoren b) Stoppen der Regelung
Ausnahmen/Fehlersituationen	Der innere Zustand des Drives ist nicht „Moving“ – es gibt keine laufende Figur, die pausiert werden kann.

Beschreibung Anwendungsfall „resume a motion“:	
Name	„resume a motion“
Kurzbeschreibung	Eine pausierte Figur wird fortgesetzt.
Akteure	Anwendungsprogramm/User
Auslöser	Anwender möchte eine pausierte Figur fortsetzen
Ergebnis	Figur wird fortgesetzt
Eingehende Daten	Nachricht zum Fortsetzen der Figur
Vorbedingungen	Kommando für die Figur wurde erzeugt (Angabe der Parameter und der Art der Figur)
Nachbedingungen	Figur wird weiter ausgeführt
Essenzielle Schritte	1. Wechsel des inneren Zustands des Drives von „Paused“ nach „Moving“ 2. Nachricht an die Hardwareschicht zum Fortsetzen der Figur (move) a) Starten der Motoren b) Starten der Regelung
Ausnahmen/Fehlersituationen	Der innere Zustand des Drives ist nicht „Paused“ – es gibt keine pausierte Figur, die wieder fortgesetzt werden kann.

Beschreibung Anwendungsfall „cancel a motion“:	
Name	„cancel a motion“
Kurzbeschreibung	Anwender bricht eine laufende, pausierte oder eine vorbereitete, noch nicht gestartete Figur ab
Akteure	Anwendungsprogramm/User
Auslöser	Anwender möchte eine Figur abbrechen
Ergebnis	Figur wird abgebrochen
Eingehende Daten	Nachricht zum Abbruch der Figur
Vorbedingungen	Eine abzubrechende Figur wird ausgeführt, pausiert oder ist zum Starten vorbereitet
Nachbedingungen	Figur wurde abgebrochen und kann nicht wieder gestartet werden

Essenzielle Schritte	1. Wechsel des inneren Zustand des Drives von „Paused“, „Setup“ oder „Moving“ nach „Idle“ 2. Nachricht an die Hardwareschicht zum Abbruch der Figur (cancel) a) Stoppen der Motoren b) Stoppen des Reglers
Ausnahmen/Fehlersituationen	Der innere Zustand des Drives ist nicht „Setup“, „Paused“ oder „Moving“ – es gibt keine laufende, pausierte oder vorbereitete Figur.

Beschreibung Anwendungsfall „get a motion report“:	
Name	„get a motion report“
Kurzbeschreibung	Anwender fordert einen Report der gefahrenen Figur an
Akteure	Anwendungsprogramm/User
Auslöser	Anwender möchte den Report zu einer Figur erhalten
Ergebnis	Es wird ein Report geliefert
Eingehende Daten	Nachricht zur Anforderung eines Reports
Ausgehende Daten	Report Objekt
Essenzielle Schritte	1. Nachricht zur Anforderung eines Reports an das Drive 2. Erstellung des Motion Reports durch das Drive a) Berechnung der aktuellen (lokalen) Pose b) Berechnung der Fehler (lateraler und direktionaler Fehler) c) Setzen der Werte für Fehler und die errechnete Pose im Report

Die Aktion „start a motion“ soll hier beispielhaft näher erläutert werden: Das Aktionsdiagramm umfasst sowohl Aktionen der High-Level- als auch der Low-Level-Schicht. Auf die einzelnen Schichten wird im folgenden Abschnitt noch näher eingegangen. Nach der Annahme der Nachricht zum Starten der Figur werden innerhalb der Low-Level-Schicht die Bewegungsregelung sowie die periodische Prüfung der zurückgelegten Strecke aktiviert. Hat das Drive die für die Figur berechneten Wege zurückgelegt, wird die High-Level -Schicht entsprechend benachrichtigt und generiert wiederum eine Nachricht an die Anwendung. Der unten stehende Ausschnitt aus dem Aktionsdiagramm „start a motion“ stellt diesen Ablauf entsprechend dar:

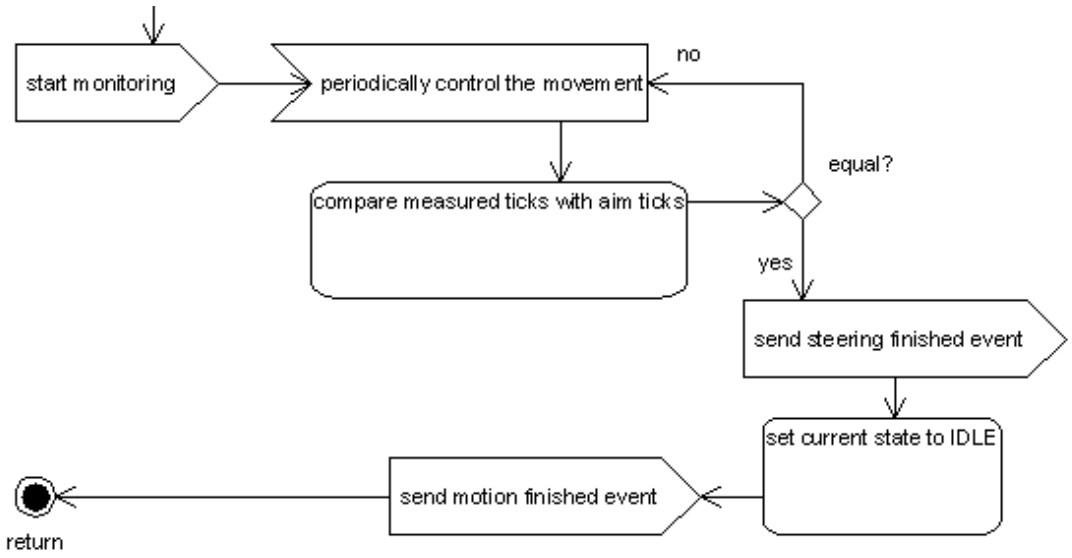


Abbildung 3.1.: Ausschnitt Aktion „start a motion“

3.2. Schichtenmodell

Die lejON-API wird, unterhalb der Anwendungsschicht angeordnet, in zwei Schichten aufgeteilt: Der High-Level-Schicht und der Low-Level-Schicht.

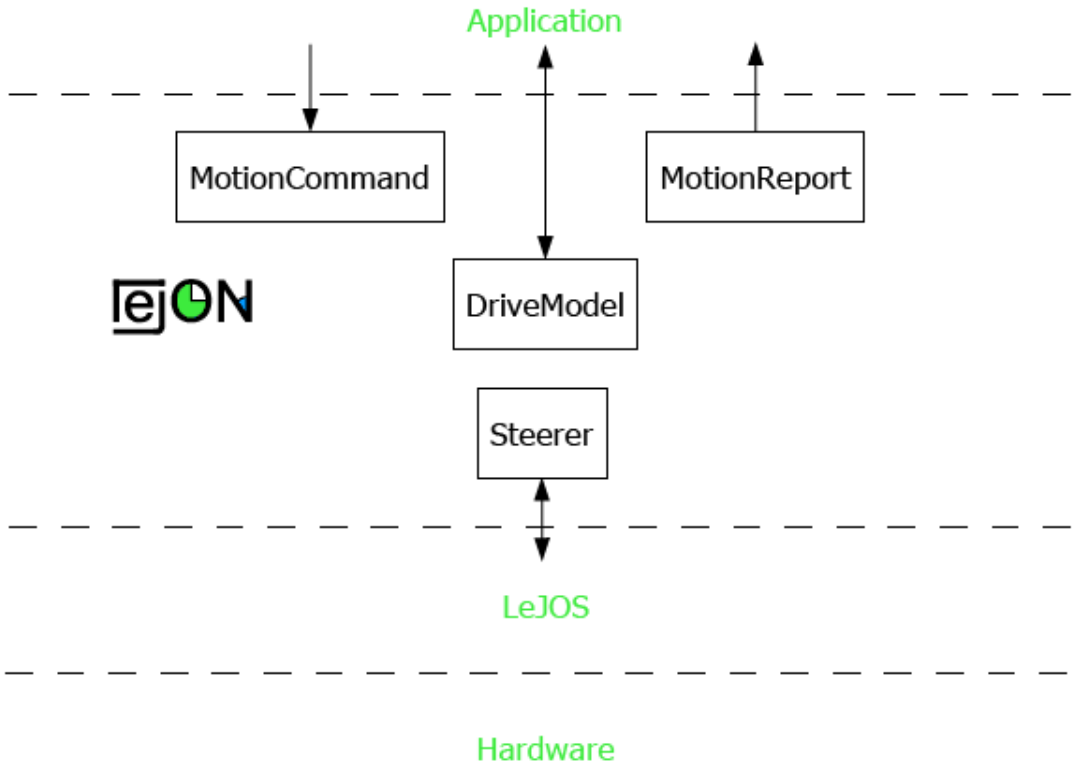


Abbildung 3.2.: lejON-Schichten

In der High-Level-Schicht werden Nachrichten der Anwendung entgegengenommen und z. B. bei Beendung einer Figur Nachrichten an die Anwendung gegeben. Die Low-Level-Schicht bildet die Schnittstelle zur Hardware des Drives unter Verwendung der LeJOS-API.

3.3. Definition der Klassen

Unter Berücksichtigung der logischen Teilung der lejON-API in die erläuterten Schichten wurden folgende Klassen identifiziert, die im Folgenden erläutert werden:

3.3.1. Klassen der High-Level API

Die Klassen *DriveParameters*, *DriveModel*, *Pose*, *MotionCommand* *MotionReport* sowie *MotionEventListener* gehören logisch zur High-Level-API.

DriveParameters

Die Klasse *DriveParameters* stellt alle wichtigen Parameter des Drives zur Verfügung:

- Referenzen zu Motor- und Sensorobjekten

- Reglerkonstanten KP, KI, KD
- maximale und minimale Leistungsstufe für Motoransteuerung
- Zeitbasis
- Ticks pro Zeitbasis für jede Motorleistungsstufe (experimentell ermittelt)
- Nominalticks
- Achslänge
- Raddurchmesser
- Übersetzungsverhältnis

Methoden zum Abrufen und Setzen der Parameter werden ebenfalls zur Verfügung gestellt.

Pose Als *Pose* werden die

- Position in x-Richtung (Fahrtrichtung),
- Position in y-Richtung und
- die Ausrichtung des Roboters als Winkelangabe

zusammengefasst. Zusätzlich ermöglichen entsprechende set- und get-Methoden das Abrufen und Setzen der einzelnen Attribute.

DriveModel Die Klasse *DriveModel* stellt eine abstrakte Klasse zur Modellierung des Drives dar und verwaltet die inneren Zustände, die durch Vorbereitung beziehungsweise Ausführung einer Figur beschrieben sind. Dies sind:

- IDLE – In diesem Zustand wird auf ein Kommando gewartet.
- SETUP – In diesem Zustand wird eine Figur auf die Ausführung vorbereitet.
- MOVING – In diesem Zustand führt das Drive eine Figur aus.
- PAUSED – In diesem Zustand ist eine ausgeführte Figur unterbrochen.

Das *DriveModel* stellt des Weiteren die Schnittstelle zur Anwendung und zur Hardware zur Verfügung und bietet daher Methoden zur

- Registrierung und Entfernung eines MotionEventListeners,
- Kontrolle der Vorbereitung und Ausführung einer Figur,
- Einholung eines MotionReport zur aktuell ausgeführten Figur.

Des Weiteren implementiert die Klasse den *SteeringEventListener*, um so Nachrichten von der Low-Level-API bei Beendung der Figur zu erhalten. Alle Implementierungen einer konkreten Roboterarchitektur sollten von dieser Klasse abgeleitet sein.

MotionEventListener Die Klasse *MotionEventListener* dient zur Benachrichtigung der Anwendung das Ereignis der Beendung einer Figur, die zuvor gestartet wurde und wird durch die Klasse *DriveModel* verwaltet. Daher muss sie im *DriveModel* registriert werden.

MotionCommand Mit Hilfe der Klasse *MotionCommand* werden Kommandos zur Ausführung der verschiedenen Figuren erstellt, die anschließend durch das *DriveModel* vorbereitet und gestartet werden können. Es müssen daher bei Erzeugung eines *MotionCommand*-Objektes entsprechend der Figur Parameter gesetzt werden.

MotionReport Objekte der Klasse *MotionReport* stellen Informationen über die ausgeführte Figur zur Verfügung. Dies sind

- die aktuelle Position des Drives (*Pose*-Objekt)
- sowie Fehler der Translation und Rotation, die bei der Ausführung der Figur im letzten Messintervall errechnet wurden.
- Erstellungszeit des Reports (Zählung der Zeitintervalle)
- Zustand der Bewegung (siehe Zustände des *DriveModel* in Abschnitt 3.3.1)

Ein *MotionReport*-Objekt wird durch das *DriveModel* verwaltet und periodisch durch innerhalb der Low-Level-API aktualisiert. Der Anwender kann damit z. B. zur Verfolgung der Roboterposition regelmäßig Informationen über die Bewegung des Roboters abrufen.

MoveMentor Die Klasse *MoveMentor* bietet die Möglichkeit zur Korrektur einer Bewegung. Nach Beendigung eines *MotionCommand* wird durch Vergleich der lokalen Roboterpose mit den Bewegungsparametern versucht, durch Störungen und Ungenauigkeiten enstandene Fehler auszugleichen. Es wird die Korrektur von Winkelfehlern bei Rotation und Kurvenfahrt unterstützt.

3.3.2. Klassen der Low-Level-API

Die Klassen *DDModel*, *Steerer*, *PIDController*, *SteeringEventListener*, dienen zur Realisierung der Hardware-Ansteuerung, Regelung und Berechnungsroutinen für die odometrische Positionsbestimmung. Sie verwenden Objekte der Klassen *Motor*, *Sensor*, *Timer* und *TimerListener* der LeJOS-API.

DDModel Die Klasse *DDModel* ist eine beispielhafte Implementierung des abstrakten *DriveModel* für ein Differential-Drive. Es Implementiert die Methoden

- *translate*
- *rotate* und

- *curve*

zur Vorbereitung der Figuren und legt damit Parameter für die Motorenansteuerung fest. Außerdem werden die Anzahl der Ticks des linken und rechten Rades für die jeweilige Figur berechnet, die insgesamt zurückgelegt werden müssen. Als Komponente beinhaltet das *DDModel* den *Steerer*, der die Regelung der Bewegung und weitere periodische Berechnungen ausführt.

Steerer Die

- Regelung der Bewegung des Roboters und die damit erforderliche differenzierte Ansteuerung der Motoren sowie
- die periodiche Berechnung der Roboter-Pose
- als auch der periodische Vergleich der durch die Rotationssensoren gemessenen Ticks mit den berechneten Zielticks

übernimmt der *Steerer*.Um periodisch Berechnungen auszuführen, implementiert die *Steerer*-Klasse den LeJOS-*TimerListener* und besitzt einen *Timer* als Member.

SteeringEventListener Die Klasse dient dazu, das *DriveModel* bei Beendung einer Figur zu benachrichtigen.

3.3.3. Zustände des DriveModel

Bei der Vorbereitung und Ausführung einer Figur kann das *DriveModel* bestimmte Zustände beschreiben (siehe Abschnitt 3.3.1). Daraus ergibt sich die in folgender Abbildung dargestellte Zustandsmaschine:

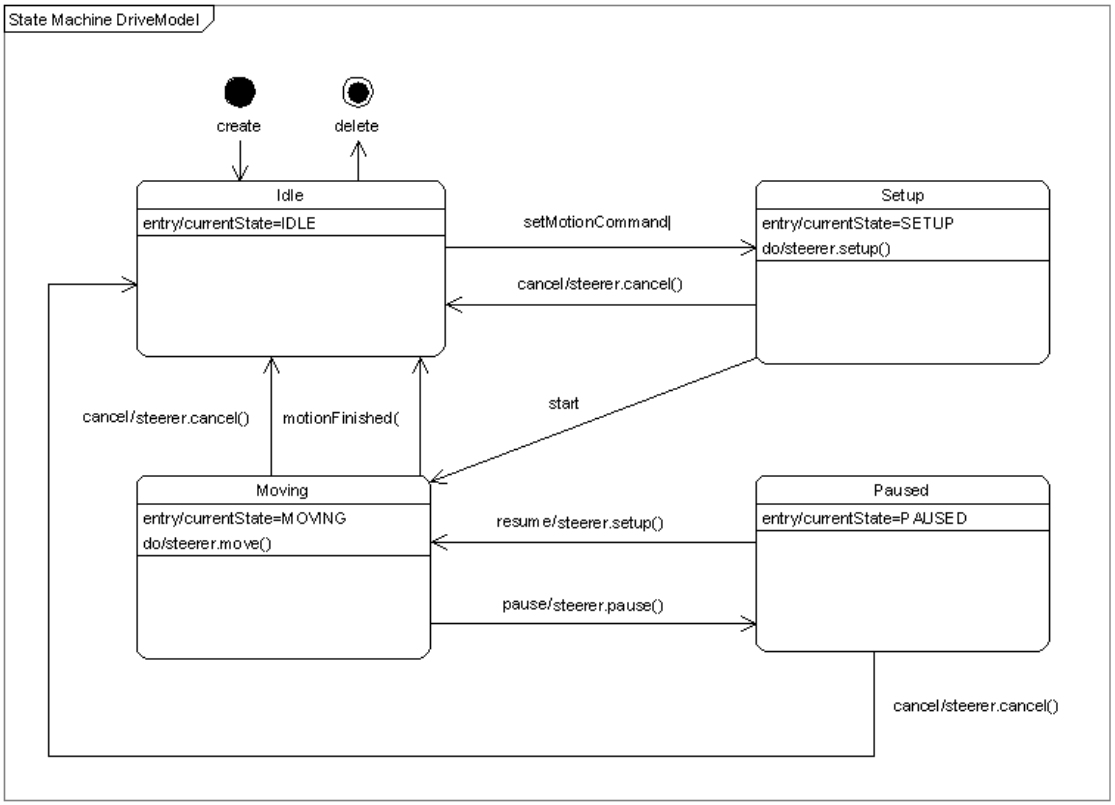


Abbildung 3.4.: Zustände

Es sind – wie in den Anwendungsfällen erläutert – nur bestimmte Zustandsübergänge möglich.

3.3.4. Kommunikation zwischen DriveModel und Steerer

Das folgende Interaktionsdiagramm zeigt die Kommunikation zwischen Anwendung, *DriveModel* und *Steerer*. Beispielhaft wird der Ablauf

- Vorbereitung der Figur,
- Starten der Figur,
- Pausieren der Figur,
- Wiederaufnahme der Bewegung und
- Beendung der Figur

für die Figur „Geradeausfahrt“ (translate) gezeigt. Entsprechend werden zwischen den Objekten Nachrichten ausgetauscht.

Nach der Initialisierung (1-3) wird eine Figur erstellt und parametriert, indem ein *MotionCommand* Objekt angelegt wird (4). Anschließend wird die Figur durch das *DriveModel* vorbereitet (5), was zu einer Initialisierung der Hardware und des Reglers im *Steerer* führt (6). Danach wird die Bewegung gestartet (7). Während der Bewegungsausführung initiiert die timedOut-Methode des Timers alle periodischen Berechnungen, die durch den Steerer vorgenommen werden. Beim Pausieren der Figur (8) wird der Timer gestoppt und bei der Wiederaufnahme der Bewegung (11) wieder gestartet. Ist der Vergleich der gemessenen Ticks mit den berechneten Zielticks für die Figur positiv, so ist die Figur beendet. Die Bewegung wird gestoppt und ein *SteeringFinishedEvent* ausgelöst (Nachricht an das *DriveModel*), woraufhin ein *MotionFinishedEvent* der Anwendung das Ende der Figur signalisiert.

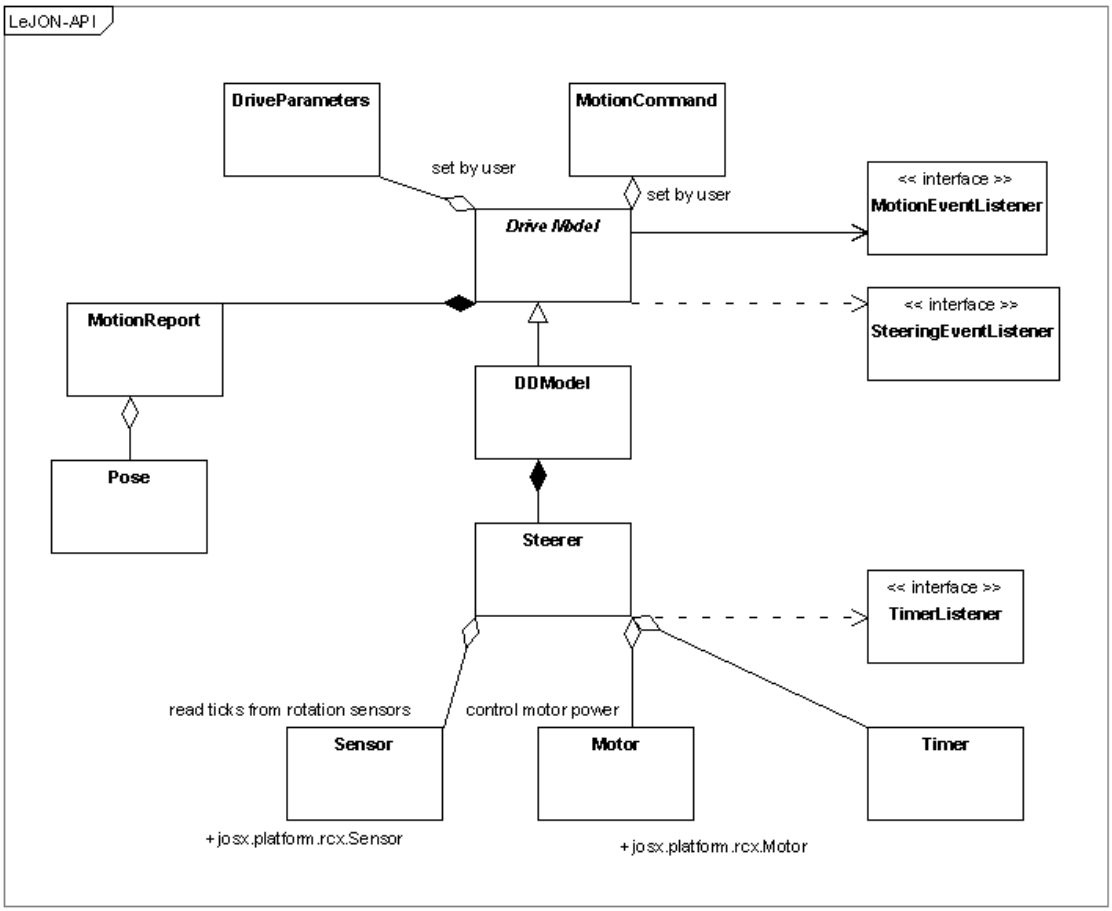


Abbildung 3.3.: lejON-Klassen

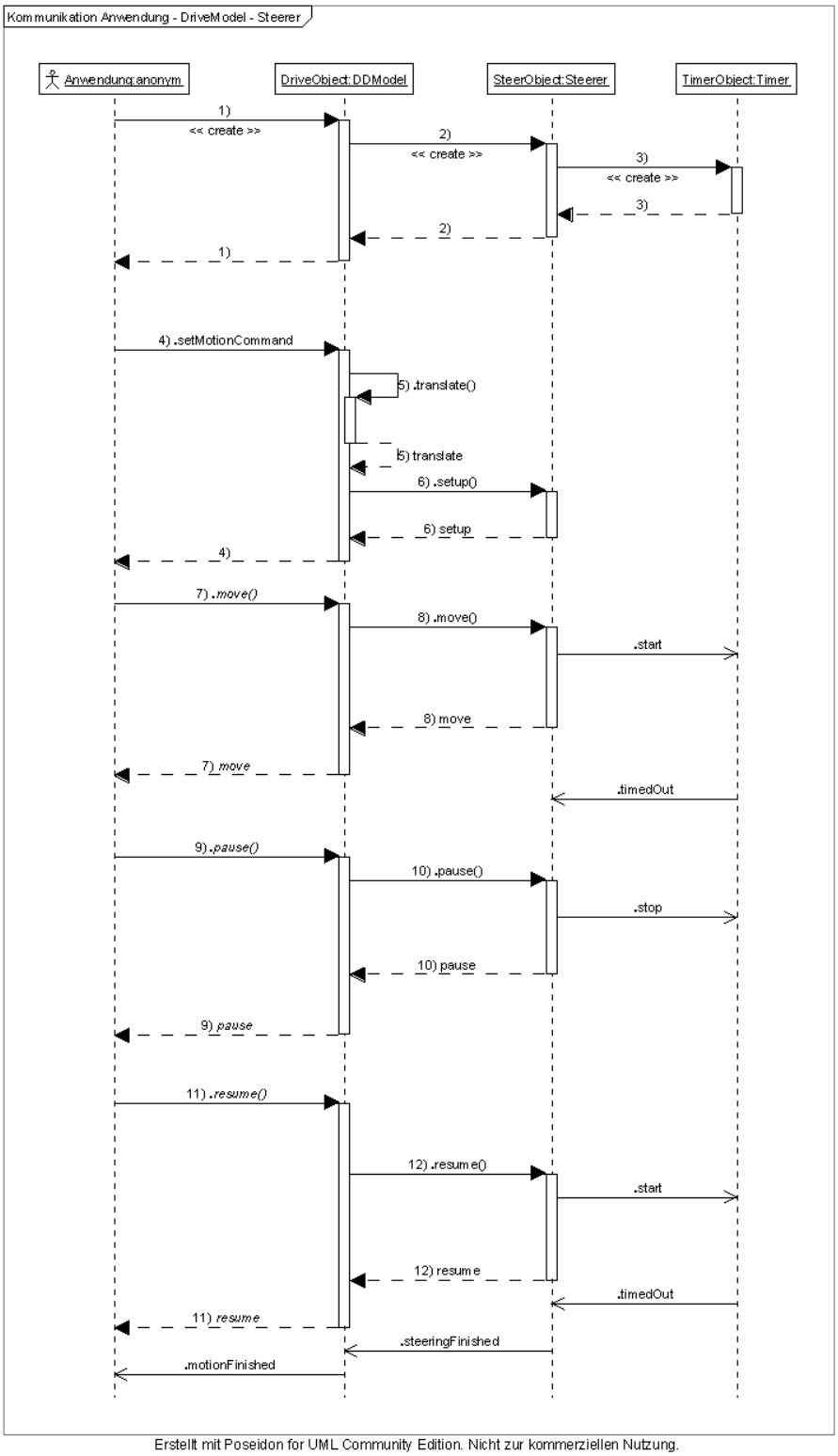


Abbildung 3.5.: Interaktion DriveModel und Steerer

4. Evaluation

In diesem Kapitel werden Evaluationsergebnisse vorgestellt und ausgewertet, um hieraus Rückschlüsse in Bezug auf die Genauigkeit der odometrischen Steuerung ziehen zu können. Zunächst sollen die Sensorrohdaten betrachtet werden. Im Weiteren werden die Einzelfiguren der lejON-API untersucht und im Anschluss daran der Testparcours dargestellt.

4.1. Evaluation der Sensorrohdaten

Die Impulse der Rotationssensoren stellen für einen Drive die einzige Grundlage für eine Navigation dar. Diese sogenannten Ticks müssen daher untersucht werden, um Auswirkungen von Übersetzungsverhältnis, Motorspannung und Reibungskoeffizient auf das Navigieren festzustellen. Hierzu sind folgende Tests durchgeführt worden:

1. Messung der Ticks im reibungsfreien Zustand bei Batteriebetrieb
2. Messung der Ticks im reibungsfreien Zustand bei Netzbetrieb
3. Messung der Ticks mit DIDI-Architektur auf Laminat bei Batteriebetrieb
4. Messung der Ticks mit HILDE-Architektur auf Laminat bei Batteriebetrieb

Im reibungsfreien Zustand ist erkennbar, dass die Ticks nicht mehr richtig gezählt und vergessen werden können. Dieses Phänomen tritt bei Übersetzungen mit einem Verhältnis von größer 1:2 auf, sowohl bei Batterie- als auch bei Netzbetrieb. Weiterhin zeichnet sich der Batteriebetrieb bei der Messung über den einzelnen Leistungsstufen durch einen exponentiellen Verlauf aus, wohingegen sich beim Netzbetrieb eine eher lineare Funktion abbildet (Abbildung 4.1). In beiden Betriebsarten werden bei Übersetzungsverhältnissen von kleiner 1:2 bei einer Messdauer von 4 Sekunden Messwerte zwischen 400 und 490 Ticks erreicht.

Auch die Daten aus den Testläufen mit der DIDI- und HILDE-Architektur zeigen einen exponentiellen Verlauf. Aufgrund der Reibung von Rad zu Boden werden hier rund 200 Ticks weniger im Messinterval gezählt. Weiterhin ist eine Abhängigkeit von Batteriespannung zur Motorenleistung erkennbar, die jedoch in einer weiteren Evaluation genauer untersucht werden könnte.

Dieser Test zeigt, dass ein Übersetzungsverhältnis von über 1:2 für unsere Architektur unter LeJOS nicht sinnvoll ist. Darüberhinaus benötigt man für jede Architektur einen Velocities-Vektor. Dieser entspricht der Ticksanzahl für jede Motorleistungsstufe zur Zeitbasis des Reglers (vgl. Listing A.1) zur ordentlichen Berechnung der Stellgröße.

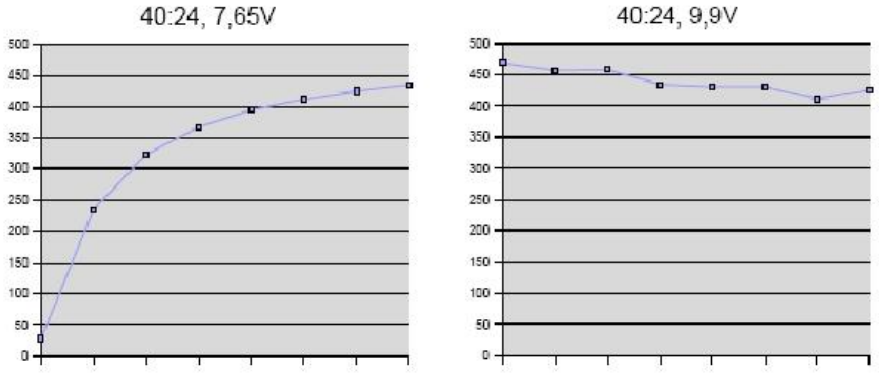


Abbildung 4.1.: Gemessene Ticks im reibungsfreien Zustand über den 8 Leistungsstufen der Motoren

Im Folgenden wird nun die Evaluation der Basisfiguren der lejON-API vorgestellt und ausgewertet.

4.2. Evaluation der Basisfiguren

Zur qualitativen Beurteilung der Navigation mittels lejON-API werden die bereitgestellten Fahrfiguren getestet. Als Architektur dient hierbei das DIDI-Modell. Im Wesentlichen soll geklärt werden, inwieweit eine vorgegebene Figur vom Drive gefahren wird, wie groß die gemessene Abweichung vom Zielpunkt ist und wo das Drive sich zu befinden glaubt.

4.2.1. Basisfigur „Translate“

Die Geradeausfahrt ist mit den Weglängen 50 cm, 100 cm, 200 cm und 400 cm getestet worden. Der zu erreichende Zieltupel lag bei folgenden Positionen: (x, y, θ) = (Weglänge, 0, 0). Die Stichprobe bestand aus fünf Einzelmessungen pro Weglänge und Motorleistungsstufe. Im Mittel wurden die in Tabelle 4.1 aufgeführten Werte für die x-Koordinate ermittelt. Die Drehung des Fahrzeugs zum Bezugskoordinatensystem entsprach einer Änderung im Maximum von $\pm 13°$. Die Änderung der y-Koordinate schwankte zwischen $+10\%$ und -10% von der zu fahrenden Wegstrecke. Eklatant ist jedoch der Unterschied zwischen der Messung y-Koordinate im Koordinatensystem zum einen und der Berechnung des Drives zum anderen. Beispielsweise wurde bei einer Distanz von 400 cm eine Strecke von 30 cm in y-Richtung zurückgelegt. Vom Drive wurden jedoch nur 3 cm berechnet. Grund hierfür ist die Auflösung der Rotationssensoren und die DIDI-Architektur, mit der die Genauigkeit auf 9° beschränkt ist. Eine direkte Abhängigkeit zwischen Motorleistungsstufe und Genauigkeit der Zielposition kann für die Fahrfigur *TRANSLATE* nicht festge-

	Distanz							
	50 cm		100 cm		200 cm		400 cm	
Leistungsstufe	M	B	M	B	M	B	M	B
0	55	54	107	106,4	202,6	203,2	399,8	403
1	57,6	55	105,8	103,6	204	205	399,8	403,6
2	58,6	57	105	104	204,6	205	398,8	403
3	56,8	54,8	105	103,8	204,4	204,8	399,6	403,4
4	56,2	54,4	106,4	105,4	203,2	203,8	400,4	403,6
5	56	54,2	107,6	106,4	203,4	204,4	399,4	403,8
6	57	55	107	106	203,2	204,6	398,8	402,2
7	57	55,4	106,4	105,6	204,2	205,6	397,8	401,4

Tabelle 4.1.: Mittelwerte der gefahrenen Distanzen. Links befinden sich die gemessenen Werte (M), rechts die vom Drive berechneten (B).

stellt werden. Hierzu bedarf es einer weitergehender Evaluierung mit einer größeren Stichprobe.

Im Vergleich zur Geradeausfahrt mit der lejON-API konnten mit dem *RotationNavigator* von LeJOS die in Tabelle 4.2 aufgelisteten Mittelwerte aus drei Messreihen erzielt werden. Auffallend ist, dass ein enormer Unterschied zwischen wirklich gefahrener und der vom Drive ermittelten Strecke liegt. Dieser Fehler liegt bei den gemittelten Werten im Maximum bei etwa 15%. Zudem scheinen Abweichungen in y-Richtung gar nicht registriert zu werden, ebenso wie θ, die Richtungsänderung des Drives. Die festgestellte Richtungsänderung zum Bezugkoordinatensystems wurde im Maximum mit 10° ermittelt.

4.2.2. Basisfigur „Rotate“

Eine weitere grundlegende Fahrfigur ist das Rotieren. Die Testreihen umfassen hierbei die Rotationswinkel 45°, 90°, 180° und 360° mit jeweils fünf Messungen pro Leistungsstufe. Die Änderung der x- und y-Koordinate kann vernachlässigt werden. Für die ersten drei Winkel variiert diese Änderung zwischen 1 cm und 2 cm. Bei der Rotation um 360° wurde als Maximalwert eine Verschiebung der Position von 6 cm festgestellt. Im Mittel liegen die Werte jedoch um den Koordinatenursprung, so dass im Weiteren nur der Rotationswinkel betrachtet wird.

Aus den Messreihen ersichtlich erhält man einen Fehler von 25° bis 50°. Dieses Ergebnis ist sicherlich nicht zufrieden stellend. Die Ursachen hierfür sind wiederum in Architektur und der Auflösung der Rotationssensoren zu suchen. Weiterhin ist zu bedenken, dass beim Motorenstopp das Fahrzeug aufgrund der Trägheit weiterrollt. Bei einem Nachrollen von 3 *ticks* und einer Genauigkeit von $9\frac{ticks}{degree}$ pro Sensor entspricht dies einem Fehler von 54°. Positiv ist aber, dass das Drive den Trägheitseffekt registriert und die berechneten Positionen diesen Fehler ebenfalls enthalten. Der Anwender kann somit darauf reagieren. Auch hier ist es schwer, eine Aussage über die

		Distanz			
		50 cm		200 cm	
Leistungsstufe		M	B	M	B
0	x	43	49	168	199
	y	0,67	0	-1,33	0
	θ	0	0	0	0
1	x	45,33	50,3	171	200
	y	-0,67	0	-5	0
	θ	0	0	-1,67	0
2	x	47	50,67	170	200,67
	y	0	0	1	0
	θ	-3,33	0	0	0
3	x	46,67	50,67	174	201
	y	-1	0	-9	0
	θ	-3,33	0	-1,67	0
4	x	47,67	51	173,67	200,67
	y	0	0	-14	0
	θ	-3,33	0	-3,33	0
5	x	48	51	173,67	200,67
	y	-1,67	0	0,67	0
	θ	-5	0	1,67	0
6	x	47,67	49	174	200,67
	y	-1,67	0	-13,33	0
	θ	-1,67	0	-3,33	0
7	x	47,67	50,67	175	201,67
	y	-4,67	0	1,67	0
	θ	-5	0	5	0

Tabelle 4.2.: Mittelwerte der ereichten Positionen mit *RotationNavigator*. Links befinden sich die gemessenen Werte (M), rechts die vom Drive berechneten (B).

	Rotation							
	45 °		90 °		180 °		360 °	
Leistungsstufe	M	B	M	B	M	B	M	B
0	68,4	71,6	133,4	133	209,4	216,4	386,4	395,6
1	68,4	71,8	135,6	134,4	201,2	203	385,6	394,2
2	84,4	84,2	120	121,2	212,2	213	395,6	405,4
3	95,4	96,8	132,4	136,6	222,6	227,6	382,6	388,6
4	94	98	143	147,2	226,6	225,6	388	397,4
5	74,2	77,2	114	116	210,2	213,2	395,4	406,6
6	69,8	68,2	110	114	208	212,2	406,4	417,4
7	74,4	74,4	115,4	116	208,6	214,8	397,8	409,2

Tabelle 4.3.: Mittelwerte der Drehwinkel. Links befinden sich die gemessenen Werte (M), rechts die vom Drive berechneten (B).

Genauigkeit der Navigation bezüglich den Leistungsstufen zu treffen. Eine größere Stichprobenmenge könnte hierüber Aufschluss geben.

	Rotation							
	45 °		90 °		180 °		360 °	
Leistungsstufe	M	B	M	B	M	B	M	B
0	148,33	45	n.e.	n.e.	n.e.	n.e.	n.e.	n.e.
1	50	45	95	90	213,3	180	385	360
2	50	45	98,3	90	188,3	180	341,6	360
3	53,3	45	101,6	90	180	180	331,6	360
4	55,67	45	100	90	183,33	180	330	360
5	55,67	45	101,67	90	181,67	180	330	360
6	59	45	103,33	90	181,67	180	331,67	360
7	58	45	99	90	180	180	338,33	360

Tabelle 4.4.: Mittelwerte der Drehwinkel mittels *RotationNavigator*. Links befinden sich die gemessenen Werte (M), rechts die vom Drive berechneten (B).

In Tabelle 4.4 sind Testreihen auf Basis des *RotationNavigator* erfasst. Die Werte wurden aus drei Messungen pro Leistungsstufe gemittelt, wobei unter Leistungsstufe 0 nicht alle Werte evaluiert werden konnten (n.e. – nicht evaluiert). Vergleicht man die Ergebnisse des *RotationNavigator* mit denen der lejON-API, so fällt auf, dass mittels *RotationNavigator* eine am Sollwert näher liegende Fahrfigur realisiert werden konnte. Wie bei der Basisfigur „Translate“ hat man auch hier das Problem, dass das Drive den gefahrenen Winkel nicht berechnet. Dies führt dazu, dass kein optimales Eingreifen des Anwenders erfolgen kann, um dem Fehler entgegenzuwirken.

4.2.3. Basisfigur „Curve“

Da das Aufnehmen der Testreihen für die Basisfigur *CURVE* sehr zeitaufwendig ist, soll sich die Evaluation der Kurvenfahrt hier auf einen Kurvenradius von 50 cm mit den Kurvenwinkeln 45°, 90°, 180° und 360° beschränken. Gemessen worden ist über den acht Motorleistungsstufen mit jeweils fünf Messreihen (n.e. – nicht evaluiert). Die gemittelten Positionen sind in Tabelle 4.5 aufgelistet. Die besten Ergebnisse konnten hier mit den Leistungsstufen 4, 5, 6 und 7 erreicht werden. Bei der Leistungsstufe 0 konnte zum Zeitpunkt der Evaluation die Kurvenfahrt nicht richtig ausgeregelt werden. Die Ursache dafür liegt in der Arbeitsweise des PID-Reglers (vgl. 1.3.4). Kurven mit einem großen Kurvenradius werden grundsätzlich zu kurz gefahren. Ursache hierfür ist, dass zur Berechnung der Zielticks der Rotationssensoren eine ideale Kurvenfahrt angenommen wird, durch die Reglung jedoch ein Schwingen um die Ideallinie erfolgt und somit mehr Strecke zurückgelegt wird. Die relative Abweichung der gemessenen Position des Drives zur berechneten liegt in einem akzeptablen Toleranzbereich, so dass die fehlende Wegstrecke vom Anwender nachgefahren werden kann.

4.3. Definition des Testparcours

Mit Hilfe der lejON-API ist es möglich Figuren zu fahren, welche sich aus den unter Punkt 4.2 vorgestellten Basisfiguren zusammensetzen. Aufgrund der dort erhaltenen Werte, ist es sinnvoll auf Anwenderebene eine Methode zu implementieren, welche den vorhandenen Fehler nach Fahrtende ausgleicht. Gerade bei einer Kreisfahrt konnten in den Tests nur Kurven im Mittel bis 285° erreicht werden. Die Abbildung 4.2 zeigt einen Parcours, welcher mit dem zur Verfügung stehenden lejON-API problemlos gefahren werden kann. Im Gegensatz hierzu kann mit dem herkömmlichen *RotationNavigator* aus dem LeJOS-API dieser Parcours nicht ohne weiteres gefahren werden, da dieser keine Funktionalität für eine Kurvenfahrt bereitstellt.

Stufe		Kurvenwinkel							
		45 °		90 °		180 °		360 °	
		Zielposition							
	x	35		50		0		0	
	y	15		50		100		0	
	θ	45		90		180		360	
		M	B	M	B	M	B	M	B
0	x	43,8	42,2	76,2	75,6	145	143,4	n.e.	n.e.
	y	3,6	3,4	9	10,6	27,4	32,8	n.e.	n.e.
	θ	14,4	10,6	15,6	14	25,4	18,6	n.e.	n.e.
1	x	42,6	40,8	71,6	68,4	117,2	110,6	90,2	68,6
	y	10,6	11,2	29,8	33,2	88,4	90,2	216,2	225,2
	θ	25,2	22,2	43,8	42,2	69,8	69,2	133	135,8
2	x	40,2	37,4	60,6	55,6	55,8	49	-74,8	-83,2
	y	18,2	19	48	51,8	117	120,2	118,8	110,2
	θ	45,2	37,6	78	76,2	123,6	124,8	237,8	240,2
3	x	41,2	38,6	60,8	56	58	50,6	-70	-79
	y	18,2	19	48,4	52,6	116,4	119,4	122	116,4
	θ	43,6	38,4	72,2	75,4	114,2	115,8	236	237,4
4	x	41,2	38	52	47,6	33	22	-68,8	-70,8
	y	17,8	20,8	53,6	57,8	118,2	119,2	50	39
	θ	48,4	42,8	84,8	83,4	146,2	148,4	277,2	285,2
5	x	42,4	39,2	53	48,2	31	20,8	-72,8	-74,6
	y	20,8	21,8	53,8	58,8	118,6	118,6	48,2	40,8
	θ	56,2	47,4	89	88	147,2	150	285,2	280,4
6	x	40,6	37,2	50,4	48,2	32,6	22	-71,4	-69,8
	y	18,8	19,8	56	58	117,8	118,6	53,8	41,6
	θ	49,6	40,2	87,4	84,4	141	145,6	282,8	289,6
7	x	41	37,8	52,2	48,6	34,4	24,4	-73,6	-72,8
	y	18,4	20	53,8	56	116,4	118	49	40,8
	θ	51	44,8	84,6	81,6	134,2	138,2	277,4	281,4

Tabelle 4.5.: Mittelwerte der gefahrenen Kurvenpositionen bei einem Kurvenradius von 50 cm. Links befinden sich die gemessenen Werte (M), rechts die vom Drive berechneten (B).

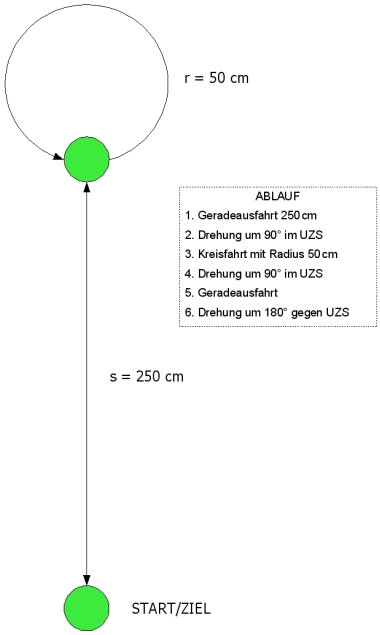


Abbildung 4.2.: Definition des Testparcours

5. Diskussion

Ein Roboter kann unter Nutzung von lejON hinreichend präzise fahren.

Aufgrund der Einschränkungen, welche die eingesetzte Hardware mit sich bringt, ist es jedoch nicht möglich, eine Fahrfigur exakt zu fahren. Dazu wäre es notwendig, die Genauigkeit der Rotationssensoren und gleichzeitig die Anzahl an Motorleistungsstufen zu erhöhen.

Die Evaluation der lejON-API hat ergeben, dass die Navigation gegenüber der in LeJOS verfügbaren nicht nur etwas genauer und die Kurvenfahrt nun als Fahrfigur verfügbar ist, sondern die Odometrie an sich konnte auch enorm verbessert werden. Begründet ist dies durch die hohe Anzahl an einstellbaren Fahrzeugparametern, insbesondere für den PID-Regler. Dadurch „weiß“ der Roboter nun fast exakt, wo er sich befindet. Er kann sich also bei entsprechender Programmierung selbst korrigieren. Zu diesem Zweck gibt es in lejON eine optionale Korrekturfunktion. Dass diese nicht automatisch durchgeführt wird, liegt zum großen Teil daran, dass der verfügbare Speicherplatz auf dem Rechnerbaustein durch die Java Virtual Machine, LeJOS und lejON beinahe gänzlich erschöpft ist und nur wenig Speicherplatz verbleibt[1]. Der Anwender soll selbst entscheiden können, wie wichtig ihm diese Eigenschaft ist oder ob er lieber anderen Funktionen höhere Priorität einräumt.

Allerdings ist besonders bei der Rotation die Korrektur ratsam. Gerade hierbei lässt die Auflösung der Rotationssensoren und das Übersetzungsverhältnis nur eine unzureichende Kontrolle der Drehung um die eigene Achse zu. Die angesprochene exakte Selbstlokalisierung kann jedoch diese Ungenauigkeit sehr gut ausgleichen. Es wird sich einfach wieder um den zu weit gedrehten Winkel zurückgedreht. Dies geschieht solange, bis das Fahrzeug innerhalb eines definierten Toleranzbereichs um den anvisierten Winkel zum Stehen kommt.

Während der Evaluation kam nur eine Fahrzeugkonfiguration zum Einsatz. Für eine genauere und allgemeinere Betrachtung sollten Testläufe über alle möglichen Werte der Hardwareparameter – wie Radgröße, Übersetzungsverhältnisse und Reifenprofil – durchgeführt werden. Den theoretischen Ansatz dazu liefert der Abschnitt 1.2.3 auf Seite 11 über Fehlerquellen, Störfaktoren der Bewegung und deren Reduzierung. In diesem Zusammenhang muss auch deutlich gemacht werden, dass die bestehende Implementierung des PID-Reglers vom Konzept abweicht. Für die lejON-API kommt ein simplerer Algorithmus zum Einsatz (s. Bemerkung 1.3.2 auf Seite 30), wohingegen das Konzept im Abschnitt 1.3 auf Seite 17 eine komplexere Berechnung vorsieht. Dies stellte sich aber erst zum Ende der Entwicklungszeit heraus und

[1] Die Code-Optimierung hinsichtlich des Speicherplatzes ist während der Entwicklung der API nicht erfolgt.

konnte nicht mehr berücksichtigt werden. Dank des modularen Aufbaus kann der Regler aber jederzeit ohne größeren Aufwand ausgetauscht werden.

Die lejON-API eignet sich durch ihren modularen Aufbau für verschiedene Antriebsarchitekturen. Während der Realisierung konnte jedoch nur eine, nämlich das DD, entwickelt und genutzt werden. Für weitere Architekturen war die Entwicklungszeit zu kurz.

Ein Grund hierfür ist die schwierige und langwierige Programmierung der Software, da es zu wenig Debug-Möglichkeiten für die Lego Mindstorms gibt und Programmübertragung mittels Infrarotschnittstelle sehr langsam und störanfällig ist. Die Fehlersuche dauerte dadurch äußerst lang.

6. Zusammenfassung

Das hier vorgestellte Projekt lejON untersucht die präzise Navigation an zweirädrigen Lego-Mindstorms-Robotern mit Hilfe der Odometrie, also der Weg- und Winkelmessung. Diese Roboter besitzen Rotationssensoren für ihre Antriebsräder und werden über den LeJOS-Kernel angesteuert. Ursprüngliches Ziel war die Verbesserung und Erweiterung vorhandener Software zur Navigation, wobei im Projektverlauf deutlich wurde, dass die Entwicklung einer neuen Architektur, der lejON-API, sinnvoll ist. Deren Ziel ist es nun, sowohl High-Level-Methoden zur Bewegungssteuerung der Roboter zur Verfügung zu stellen als auch die Verknüpfung zur unteren Hardwareschicht von LeJOS sicher zu stellen. Die in der Praxis auftretenden Störfaktoren werden mit einem PID-Regler kontrolliert, der im Abschnitt 1.3 auf Seite 17 vorgestellt wird. Dieser softwareseitige Regler wird für eine Hardwarearchitektur eingesetzt, welche bereits in ihrer Entwurfsphase von potenziellen Störquellen befreit ist. Zwei mögliche Architekturen, das *(Single) Differential Drive* (DD) und das *Dual Differential Drive* (DDD), werden in Kapitel 2 auf Seite 31 behandelt. Die Funktionsweise der lejON-API wird im folgenden Kapitel ab Seite 35 skizziert. Sie stellt eine Zwischenschicht dar, über die das Anwenderprogramm mit der Hardware kommunizieren kann. Dabei kommt das Konzept der Zustandsautomaten in lejON zur Geltung. Dies vereinfacht die Validierung und Handhabung von lejON, so dass dem Anwender komfortable Navigationsfunktionen zur Verfügung stehen. Die im Rahmen dieses Projekts entwickelte lejON-API wurde mittels eines DD-Prototyps getestet. Die getätigten Testläufe werden im Kapitel 4 auf Seite 48 vorgestellt und evaluiert. Die Navigation über die lejON-API und die hier vorgestellte Odometrie ist also trotz der hardware- und softwareseitigen Beschränkungen ausreichend präzise, was in Kapitel 5 auf Seite 56 diskutiert wird. Ein Beipiel, wie lejON in den Anwendercode eingebettet werden kann, liefert der Anhang A auf Seite 61. Diese API ist zukunftssicher. Da im Herbst 2006 eine neue Generation der Lego Mindstorms namens NXT verfügbar sein wird, wird im Kapitel 7 auf Seite 59 ein Ausblick auf eine mögliche Anpassung der lejON-API an NXT gegeben. Die lejON-API ist also nicht nur eine Verbesserung der bisherigen Navigation mit LeJOS sondern bietet darüberhinaus eine höhere Genauigkeit bei der eigenen Positionsbestimmung sowie neue Funktionen, insbesondere die Kurvenfahrt.

7. Ausblick

Die für dieses Projekt eingesetzte Hardware basiert auf dem Lego-Mindstorms-Baukasten „Robotic Invention System 2.0“ aus dem Jahre 1998. Wie bereits erörtert, ist diese Generation aufgrund ihrer mittlerweile veralteten Technik in ihrer Anwendung deutlich begrenzt.

Die Lego Group hat jedoch bereits angekündigt, zum August 2006 eine neue Mindstorms-Generation namens *NXT* zu veröffentlichen (vgl. [McN06]). Die deutsche Version wird voraussichtlich Mitte Oktober ausgeliefert werden. Neben vielen kleineren Verbesserungen ist insbesondere die Hardware erneuert worden. So können selbstentwickelte Programme wesentlich effizienter und genauer agieren. Es steht nun ein 32-bit-Microcontroller mit größerem Speicher zur Verfügung, welcher nun vier Eingänge und drei Ausgänge bedienen kann. Durch die Ausgabe auf dem Matrix-Display, der Klangerzeugung mit Hilfe richtiger Lautsprecher und den Verbindungmöglichkeiten über USB 2.0 und Bluetooth haben sich die Debug-Möglichkeiten enorm verbessert. Damit können die Programme nun genauer und komfortabler analysiert werden.

Dieses Projekt kann bei den neuen *Mindstorms NXT* auf Motoren zurückgreifen, deren Rotationssensoren bereits im Gehäuse integriert sind. Die Auflösung der Sensoren soll von 22° auf 1° verfeinert werden, wobei auch die Leistungsstufen der Motoren von acht auf 100 (zumindest in der beigelegten Programmierumgebung) steigen soll.

Laut den Entwicklern des LeJOS-Projekts wird auch LeJOS selbst an die neue Generation angepasst werden (vgl. [And02]). Dies kann wesentlich schneller geschehen, da Lego plant, die Firmware samt Quellcode und entsprechendem „Software Development Kit“ von vornherein offenzulegen und somit den Zugriff für Entwickler anderer Betriebssysteme und Programmierumgebungen zu erleichtern.

Aus diesem Grund bietet es sich an, auch lejON für NXT bereitzustellen. Durch die effiziente Schichtenstruktur ist es lediglich notwendig, die Low-Level-Funktionen anzupassen, wie etwa die Methoden zur Ansteuerung der Motoren oder zum Auslesen der Rotationssensoren. Die High-Level-Funktionen können hingegen unverändert bleiben. So ist es für den Anwender bequem und einfach, neue Programme mit Hilfe von lejON zu entwerfen. Wie in der aktuellen API muss er lediglich für neue zweirädrige Roboter auch neue PID-Konstanten bestimmen, die er dann in den Initialisierungsmethoden einsetzt. Mit Hilfe der NXT-Hardware kann dann eine deutlich präzisere Navigation erfolgen als bisher.

Zusätzlich zu den Möglichkeiten, welche die neue Hardware bietet, wird das Projekt lejON unter der GNU Public License (GPL) auf der Open-Source-Platform

„sourceforge“ veröffentlicht[1]. Damit können alle Interessierten an der Weiterentwicklung von lejON aktiv teilnehmen oder für eigene Anwendungen den Quellcode und die Dokumentation der lejON-API herunterladen.

Darüberhinaus ist dieses Projekt als Beitrag für die „2. Wernigeröder Automatisierungs- und Informatiktage (WAIT)“[2] 2006 eingereicht. Der Artikel (vgl. [GRR06]) behandelt insbesondere den Themenbereich der Mobilität und der Softwaretechnik und stellt dabei lejON vor als ein aktuelles, anwendungsorientiertes Forschungsergebnis.

[1] Siehe dazu: http://lejon.sourceforge.net/
[2] Siehe dazu: http://fstolzenburg.hs-harz.de/wait2006/

Anhang A.

lejON nutzen

Im Folgenden soll die Handhabung der leJON-API am Beispiel eines single differential drive (DIDI) dargestellt werden. Eine genauere Beschreibung der Funktionen und Methoden dieses API kann im zugehörigen Javadoc entnommen werden. Zunächst müssen wichtige Parameter des Fahrzeugs initialisiert werden. Diese sind von Fahrzeug zu Fahrzeug unterschiedlich und müssen somit für jede Fahrzeugarchitektur neu bestimmt werden. Zu diesen Parametern zählen alle physikalischen Eigenschaften, die für die Odometrie nötig sind. Die Initialisierung der Parameter wird in A.1 aufgeführt. Hierbei handelt es sich um die Achslänge, den Raddurchmesser, das Übersetzungsverhältnis von Rad zu Sensor sowie die Anzahl der Nominalticks der Rotationssensoren bei einer vollen Umdrehung. Weiterhin werden die ID's der angeschlossenen Motoren und Sensoren sowie ein für das Drive bestimmter Geschwindkeitskeitvektor übergeben. Abschließend werden die Konstanten für den PID-Regler gesetzt, welche ebenfalls für jedes Fahrzeug selbst bestimmt werden müssen.

Tafel A.1: Initialisierung der Drive-Parameter

```
// experimentally defined velocity vector for DIDI
int [] velocities = new int [] {1, 3, 5, 5, 6, 6, 6, 6};

// PID constants for DIDI
float [] pidConstants = new float [] {1.56f, 0.48f, 0.8f};

// set dd parameters
DriveParameters params = new DriveParameters
(
    16,                     // ticks nominal
    10.5f,                  // axle length
    7.0f,                   // wheel diameter
    40.0f / 24.0f,          // gear ratio
    Motor.C.getId(),        // motor ID left
    Motor.A.getId(),        // motor ID right
    Sensor.S3.getId(),      // rotation sensor ID left
    Sensor.S1.getId(),      // rotation sensor ID right
    true,                   // rotation sensors count backward
    false,                  // wheels and motors rotate in the same direction
    pidConstants,           // PID constants
    100,                    // time base in [ms]
    6,                      // maximum ticks per time base
    velocities              // velocity vector
);
```

Im nächsten Schritt (siehe A.2) muss nun das DriveModel initialisiert und optional ein *EventListener* angemeldet werden. Da das Fahrzeug einer bestimmten Architektur zugeordnet ist, muss die richtige Instanz geholt werden. In diesem Beispiel wird ein Single Differential Drive Model verwendet, welches als Singleton hinterlegt ist und über die Methode `instance()` zugewiesen werden kann.

Tafel A.2: Initialisierung des DriveModel

```
// create a model for single differential drive
DriveModel myDrive = DDModel.instance(params);

// tell who is listener
myDrive.addMotionEventListener(this);
```

Somit ist die Initialisierung abgeschlossen. Nachfolgend soll nun dargestellt werden, wie eine Bewegung ausgelöst werden kann. Hierzu wird ein sogenanntes MotionCommand benötigt, um dem DriveModel mitzuteilen, welche Figur als nächstes gefahren werden soll. Insgesamt können vier verschiedene Kommandos erteilt werden entsprechend der vier Basisfiguren. Im Listing A.3 wird ein solches Kommado für eine 100 cm lange Geradeausfahrt aufgezeigt. Nachdem das Kommando an das DriveModel mit der Methode `sendMotionCommand()` gesendet worden ist, kann mit dem Befehl `move()` die Bewegung ausgelöst werden.

Tafel A.3: Bewegung auslösen

```
// create a command for moving
MotionCommand cmd = new MotionCommand();

cmd.setAction(MotionCommand.TRANSLATE); // action
cmd.setDistance(100);                   // move 100 cm straight
cmd.setAngle(0);                        // ignored for action TRANSLATE
cmd.setRadius(0);                       // ignored for action TRANSLATE
cmd.setMaxPower(4);                     // set motor power

// prepare moving
myDrive.sendMotionCommand(cmd);

// now move
myDrive.move();
```

Anhang B.

lejON-CD

Literaturverzeichnis

[AM88] Alexander, J.C und J.H Maddocks: *On The Kinematics Of Wheeled Mobile Robots*. Online Paper, 1988.

[And98] Anderson, David P.: *SR04 Mobile Robot*. Website, 1998.

[And02] Andrews, P. et al.: *LeJOS, Java for the RCX*. URI: http://lejos.sourceforge.net/, Stand: 22.03.2006, 2002.

[Bau00] Baums, David: *Daves Baums Lego Mindstorms Roboter*. Galileo Press, 1. Auflage, 2000.

[Bir99] Birk, Andreas: *Autonomous Systems 1999/2000, Slides PARTV*. 1999.

[Bir01] Birk, Andreas: *Autonomous Systems, Draft-Version Alpha 0.2*. no publisher, 2001.

[Bru05] Bruyninckx, Herman: *The Robotics WEBook*. Online Book, 2005.

[CH05] Cheng, Chian und Martin Hirzer: *Odometrie, Wo bin ich?* Slides, 2005.

[CMM03] Corseaux, Dominique, Gegoire Maillet und Jean-Baptiste Maillet: *BOTZILLA Projekt*. Website, 2003.

[DJ00] Dudek, Gregory und Michael Jenkin: *Computational principles of mobile robotics*. Cambridge University Press, 2000.

[GRR06] Grohmann, Jan, Annedore Rössling und Florian Ruh: *lejon – LeJOS Odometric Navigator, Artikel für WAIT 2006*. noch nicht veröffentlicht, 2006.

[KBM98] Kortenkamp, David, Peter Bonasso und Robin Murphy (Herausgeber): *Artificial intelligence and mobile robots: case studies of successfull robot systems*. AAAI Press/ The MIT Press, 1998.

[Luc01] Lucas, G. W.: *Using a PID-based Technic For Competitive Odometry and Dead-Reckoning*. Website, 2001.

[LW98] Lutz, Holger und Wolfgang Wendt: *Taschenbuch der Regelungstechnik, 2. Aufl.* Thun, 1998.

[McN06] McNally, M.: *Lego Mindstorms NXT Key Product Features*. URI: http://mindstorms.lego.com/press/2057/LEGO%20MINDSTORMS%20NXT%20Key%20Product%20Features.aspx, Stand: 22.03.2006, 2006.