

Dokumentation

Semesterprojekt - Softwaretechnik für autonome Roboterteams

*Jens Bork, Denis Erfurt, Lorenz Fichte,
Robert Fritz, Joseph Hufnagl, Sebastian
Günther, Sven Schröder, Jonathan Sielhorst
und Cordt Voigt*

Berlin, Sevilla, Zürich; den 14.10.2012

Inhaltsverzeichnis

I Allgemein	3
1 Vorwort	4
2 Einleitung	5
3 Aufgabenstellung	6
3.1 Formulierung der Aufgabe	6
3.2 Verfügbare Ressourcen	6
4 Anforderungserhebung	8
4.1 Systemanforderungen	8
4.2 Benutzeranforderungen	9
5 Projekt- / Zeitplanung	11
5.1 Gruppenaufteilung	11
5.2 Grobe Ablaufplanung	11
II Projektumsetzung	12
6 MCC - Mission Control Center	13
6.1 Statusorientierter Ablauf	13
6.2 Architekturmodell	14
6.2.1 Map Model	14
6.2.2 NAO/NXT Model	14
6.2.3 Controler / View	14
6.3 Verfahren zur Problemlösung	14
6.3.1 Kommunikation	14
6.3.2 Auswertung der Kalibrierung	15
6.3.3 Geführte Erkundung	15
6.3.4 Pfad zum Ziel	16
7 NXT	18
7.1 Entwurf	18
7.1.1 Software	18
7.1.2 Idee 1 → Modell 1	19
7.1.3 Idee 2 → Modell 2	20
7.1.4 Idee 3 → Modell 3	21
7.1.5 Fazit und Entscheidung	23
7.2 Kommunikation	23
7.2.1 Bluetooth	23
7.2.2 Kommunikationsprotokoll PC ↔ NXT	23
7.2.3 Kommunikation mit dem MCC	24
7.3 Logik	24
7.3.1 Explorationsalgorithmen	25
7.3.2 GoToPoint	26

8 NAO	28
8.1 Vorüberlegungen	28
8.2 Software NAO	28
8.2.1 NAOqi	28
8.2.2 Choreographie	29
8.2.3 Weitere Software	29
8.3 Hardware NAO	29
8.4 Messungen	30
8.4.1 Kamera	30
8.4.2 Sonar	30
8.4.3 NAO Walk	30
8.4.4 Auswertung	31
8.4.5 Fazit	31
8.5 Kalibrierung	32
8.5.1 NAOMarker	32
8.5.2 Marker Konstruktion für NXT	32
8.5.3 Auslesen der übergebenen Werte	33
8.5.4 Farberkennung	34
8.5.5 Triangulation	35
8.6 NAOWalk mit RedBall Tracking	35
8.7 Probleme & Lösungen	37
8.7.1 NAO	37
8.7.2 NAOMarker	38
8.7.3 Triangulation	38
8.7.4 Farberkennung	38
III Fazit	39
9 Fazit	40
9.1 Aufgabenstellung	40
9.2 Projektplanung	40
9.3 Aussicht	41

I Allgemein

1 Vorwort	4
2 Einleitung	5
3 Aufgabenstellung	6
3.1 Formulierung der Aufgabe	6
3.2 Verfügbare Ressourcen	6
4 Anforderungserhebung	8
4.1 Systemanforderungen	8
4.2 Benutzeranforderungen	9
5 Projekt- / Zeitplanung	11
5.1 Gruppenaufteilung	11
5.2 Grobe Ablaufplanung	11

Kapitel 1

Vorwort

Bei der vorliegenden Projektdokumentation handelt es sich um die abschließende Dokumentation des Semesterprojektes “Softwaretechnik für autonome Roboterteams” im Sommersemester 2012 am Institut für Informatik der Humboldt-Universität zu Berlin. Das Projekt ist verpflichtender Bestandteil zum Erlangen des Bachelor of Science Abschlusses am Institut.

Diese Dokumentation wurde Frau Prof. Dr. sc. nat. Verena Hafner und Herrn Prof. Dr. rer. nat. habil. Bernd-Holger Schlingloff, den Betreuern des Projekts, vorgelegt. Durchgeführt wurde das Projekt von Jens Bork, Denis Erfurt, Lorenz Fichte, Robert Fritz, Sebastian Günther, Joseph Hufnagl, Jonathan Sielhorst, Sven Schröder und Cordt Voigt. Außerdem wurde das Projekt von Marcus Scheunemann vom Lehrstuhl Kognitive Robotik betreut. In der Studienordnung¹ ist das Semesterprojekt folgendermaßen beschrieben: “[...] Studierende üben die Fähigkeit, in einem Team ein komplexes System, das eine gegebene Aufgabenstellung löst, in Hard- und/oder Software zu entwerfen, zu entwickeln, zu testen und zu dokumentieren sowie die Ergebnisse in geeigneter Form zu präsentieren. Sie erlangen Kenntnisse über die typischen Probleme bei Projekten mit mehr als 2 Beteiligten. Sie erhalten die Fähigkeit zur selbstkritischen Präsentation des Erreichten und der vorgenommenen Entscheidungen.”

Diese Dokumentation soll also nicht nur den Zweck haben das Problem und die Lösungstrategie zu beschreiben und zu dokumentieren, sondern auch die während des Projekts aufgetretenen Probleme, Herangehensweisen und Schwierigkeiten zu beleuchten und somit anderen Studierenden die Möglichkeit geben daraus zu profitieren.

Berlin, Sevilla, Zürich, den 14.10.2012

¹<http://www.informatik.hu-berlin.de/institut/dokumente/ordnungen>

Kapitel 2

Einleitung

Das in den nachfolgenden Kapiteln beschriebene Projekt wurde in einer vollständig heterogenen Soft- und Hardwareumgebung durchgeführt, es wird daher weitestgehend auf betriebssystem-, software- oder hardwarespezifische Beschreibungen verzichtet - da es jedoch auch Teilaufgaben gab, bei denen ein homogenes Umfeld maßgeblich für einen korrekten Ablauf war, werden diese Notwendigkeiten gesondert erläutert. Insbesondere die Arbeitsumgebungen der in diesem Projekt verwendeten Roboter werden genauer erläutert.

Die Programmiersprache die durchgehend verwendet wird ist Python in der Version 2.7; lediglich bei der hardwarenahen Ansteuerung einer Robotergruppe wurde aus pragmatischen Gründen NXC (Not exactly C,¹) verwendet. Ein grundsätzliches Verständnis der Themengebiete Informatik und Programmierung wird für das vorliegende Dokument vorausgesetzt.

¹<http://bricxcc.sourceforge.net/nbc/>

Kapitel 3

Aufgabenstellung

3.1 Formulierung der Aufgabe

Die grobe Aufgabenstellung, wie sie ursprünglich ausgeschrieben war lautet: "Bei Erkundungsaktionen in für Menschen schwer zugänglichen Gebieten - etwa in Katastrophengebieten wie der Region Fukushima - können künftig Teams von autonomen Robotern eingesetzt werden, die miteinander kooperieren. Ein Flugroboter erkundet dabei das Gebiet aus der Luft und gibt den am Boden agierenden Rettungsrobotern Lage- und Geländeinformationen. Damit solch ein Szenario reibungslos funktioniert, müssen eine Reihe software-technischer Probleme gelöst werden, von der Modellierung der Anforderungen über die Implementierung der Kommunikation und Sensorverarbeitung bis hin zur Konstruktion geeigneter Sensorik, Motorik und Verhalten der mobilen Plattformen. In diesem Semesterprojekt soll ein autonomes RoboterTeam, bestehend aus einem Quadrocopter, humanoiden Nao-Robotern und mehreren NXT-Robotern, entworfen und realisiert werden, welches eine kooperative Aufgabe selbstständig in schwer zugänglichem Territorium durchführt. Schwerpunkt liegt dabei auf der Modellierung und dem Entwurf der Software für das verteilte System, welche nachweisbar bestimmte Korrektheitseigenschaften erfüllen muss. Begleitend zum Projekt werden das Seminar *Schwarmverhalten* und die Vorlesung *Software-Verifikation* angeboten. Die Teilnehmerzahl ist begrenzt. Von den Teilnehmern werden Vorkenntnisse in Programmierung und verteilten Systemen vorausgesetzt. Die Themen sind eng mit den Forschungsarbeiten der LS Kognitive Robotik und SVT verbunden und können zu Abschlussarbeiten führen. Besonders wichtig ist die Zusammenarbeit im Team."¹

Im Verlauf der ersten Treffen der Gruppenmitglieder mit den Professoren wurde diese grobe Aufgabenstellung konkretisiert, indem verschiedene Möglichkeiten, das Problem zu lösen, diskutiert wurden. Einerseits wurde zunächst von einem Flugroboter abgesehen, da die Auswertung bewegter Bilddaten bereits ein sehr anspruchsvolles und umfangreiches Problem für sich allein ist - optional hätte dieser bei ausreichenden Ressourcen noch hinzu genommen werden können. Das Gebiet indem der Zielbereich gefunden werden soll, kann durch Wände oder andere glatte Oberflächen begrenzt werden; die Hindernisse sollen durch Ziegel oder ähnliche, gleich beschaffene, Steine modelliert werden. Auch hier ist eine Anpassung des Schwierigkeitsgrades durch die Anzahl der Steine und die Größe des Spielfelds möglich. Eine zeitliche Beschränkung besteht automatisch durch die Akkulebensdauer der Roboter (s. dazu Abschnitt NAO, bzw. NXT). Die erste Idee für die allgemeine Vorgehensweise ist, die NXT's für die Erkundung der Umgebung zu verwenden, die dadurch erhaltenen Daten durch ein steuerndes System auszuwerten und anschließend zu benutzen, um den NAO zum Ziel zu führen. Die genauen Rahmenbedingungen durch die Aufgabe und die Einschränkungen und Anforderungen an das System und die Roboter sind in der Anforderungserhebung formuliert (siehe Kapitel 4 auf Seite 8).

3.2 Verfügbare Ressourcen

Für das Projekt sind laut Studienordnung 12 SP angesetzt - dies entspricht etwa einem Aufwand von 360 Stunden, also etwa 25 bis 30 Stunden pro Person pro Woche. Zusätzlich haben Frau Professor Hafner, Herr Professor Schlingloff und Marcus Scheuenemann einmal wöchentlich an einer gemeinsamen Sitzung von 90 Minuten teilgenommen und waren darüber hinaus auch während des gesamten Semesters als Ansprechpartner verfügbar. Die für das Projekt eingeplanten Roboter waren zwei NAO's, sowie mehrere NXT-Bausätze, die je nach Bauart ausreichend für 3-5 Roboter sind. Für Test- und Arbeitszwecke wurde

¹<https://goya3.informatik.hu-berlin.de/goyacs/course/showCourseDetails.do?id=8363&caller=overview>

ein Raum im Fraunhofer Institut in Berlin-Adlershof zur Verfügung gestellt, der uneingeschränkt genutzt werden konnte. Weiterhin wurden zur Ausführung und Entwicklung der Software die privaten Rechner der Studenten verwendet - Lizenzen für eine Python-Entwicklungsumgebung wurden von der Universität zur Verfügung gestellt.

Kapitel 4

Anforderungserhebung

Die Anforderungserhebung wurde aus der groben Aufgabenstellung (s. dazu “Aufgabenstellung”) und einer Einigung der Projektteilnehmer über den konkreten Ablauf der Mission erstellt. Sie ist in System- und Benutzeranforderungen gegliedert maßgeblich für das gesamte Projekt:

4.1 Systemanforderungen

1. NAO

1.1. Der NAO soll helfen können die NXT's zu kalibrieren

- 1.1.1. Die NXT's tragen TAGs die gemäß der Bedienungsanleitung vom NAO optisch erkennbar sind, wenn der NXT in einem Abstand von 30 - 100 cm vom NAO ist
- 1.1.2. Der NAO muss das ganze ihm mögliche Sichtfeld absuchen können in dem er im Stehen sein Kopf bewegt
- 1.1.3. Wenn ein NAO ein freies Sichtfeld auf den NXT hat, sendet er dem MCC auf Anfrage Informationen über die relative Lage, sonst muss der NAO dem MCC den Fehler, dass der NXT nicht gesehen werden kann, melden

1.2. Der NAO muss sich zu Beginn nach dem Aufstehen und wenn er hinfällt mit Hilfe der NXT's auf 2 cm genau kalibrieren können

- 1.2.1. Der NAO hat ein freies Sichtfeld auf 2 NXT's, sonst muss der NAO dem MCC den Fehler, dass er einen oder beide NXT's nicht sehen kann, melden
- 1.2.2. Die NXT's sind in einem Abstand von 30 - 100 cm, sonst muss der NAO dem MCC den Fehler, dass ein oder beide NXT's nicht im gewünschten Radius sind, melden

1.3. Der NAO muss sich anhand von Orientierungspunkten (Steine, NXT's) einem gegebenen Pfad mit einem Fehler von höchstens 5 cm folgen können

1.4. NAOWalk

- 1.4.1. Das MCC muss einen Pfad zum Ziel bestimmen und diesen an den NAO übermitteln können
- 1.4.2. Der NAO muss mit Hilfe von einem NXT mit einem roten Ball zum Ziel laufen können
 - A. Der NAO muss dem 1 Meter entfernten NXT mit dem roten Ball folgen können und ihm mitteilen, dass er angekommen ist
 - B. Der NAO muss sich das letzte Teilstück zum NXT merken und auf die ehemalige Position des NXT laufen können wenn dieser sich weiterbewegt hat.

1.4.3. Der NAO muss den Becher am Ziel platzieren können

2. NXT

2.1. Der NXT muss zu einem gegebenen Punkt fahren können mit einer maximalen Abweichung von 10 cm

2.2. Der NXT darf für 1 m Luftlinie maximal 1 min brauchen

2.3. Der NXT muss das Ziel identifizieren können wenn er darüber fährt

2.4. Der NXT muss selbstständig Kollisionen verarbeiten können, so dass er anschließend weiter fahren kann

- 2.5. Der NXT muss sich selbstständig, effektiv in einem unbekannten Gebiet bewegen können
- 2.6. Der NXT muss seine Position, den Winkel seiner aktuellen Ausrichtung und ob er sich auf dem Ziel befindet oder nicht laufend an das MCC übermitteln können

3. MCC

- 3.1. Das MCC muss mit 2 NAO's kommunizieren können
- 3.2. Das MCC muss mit bis zu 5 NXT's kommunizieren können
- 3.3. Das MCC muss die 5 Zustände *Initial*, *AutonomicExploration*, *GuidedExploration*, *PathVerification* und *NAOWalk* mit folgenden Zustandsübergangsanforderungen verwalten können:

Initial

Tritt nach Beginn der Mission ein

AutonomicExploration

Tritt ein nachdem die NAO's stehen, kalibriert sind und die NXT's ihre absolute Position erhalten haben

GuidedExploration

Tritt nach 5 Minuten des vorherigen Zustands ein

PathVerification

Tritt ein nachdem das Ziel gefunden wurde, ein Pfad dorthin berechnet wurde und 80

NAOWalk

Tritt ein nachdem die vorherige Phase erfolgreich abgeschlossen wurde; sonst wird die *GuidedExploration* wiederholt

- 3.3.1. Das MCC muss den NAO's und den NXT's die aktuelle Missionsphase mitteilen können
- 3.4. Das MCC muss eine Karte aus den Messungen der NXT's generieren können
 - 3.4.1. Das MCC kann die Karten in vereinfachter Form an die Roboter schicken können
 - 3.4.2. Das MCC muss nach einer Kalibrierung die Messfehler mit Hilfe der Abweichung, die die Kalibrierung ergeben hat, verringern können
- 3.5. Das MCC muss mit Hilfe der beiden NAO's einen NXT kalibrieren können
 - 3.5.1. Das MCC muss die Funktion kalibrieren bereit stellen, Die eine Anfrage an die beiden NAO's stellt einen bestimmten NXT zu kalibrieren
 - 3.5.2. Wenn beide NAO's relative Koordinaten für den NXT liefern, dann ermittelt das MCC die absolute Position des NXTs und teilt ihm diese mit
 - 3.5.3. Nach erfolgter Kalibrierung, sendet das MCC dem NXT seine aktuelle Koordinaten.
 - 3.5.4. Das MCC muss den NXT auf 2 cm genau identifizieren können
 - 3.5.5. Wenn ein oder beide NAO's den Fehler melden, dass der NXT nicht in Sichtfeld ist, muss das MCC dem NXT eine neue Position geben können, die das Problem möglicherweise behebt

4.2 Benutzeranforderungen

1. Der Benutzer muss die Roboter auf initiale Positionen bringen, so dass alle NXT's im Sichtfeld beider NAO's sind
2. Der Benutzer muss die Mission starten können
3. Die Mission muss bis auf den Systemstart ohne menschlichen Eingriff ablaufen und nach maximal 30 Minuten erfolgreich beendet sein
4. Es sollen 2 NAOs und bis zu 5 NXT's verwendet werden
5. Das Spielfeld soll eine Größe von 3m * 3m haben
6. Auf dem Spielfeld befinden sich Hindernisse
 - 6.1. Die verwendeten Hindernisse sind so zu plazieren, dass mindestens ein möglicher Weg vom NAO zum Ziel existiert
 - 6.2. Die Hindernisse sind von den NXT's erkennbar, d.h. sie dürfen sich nicht verschieben oder umfallen, wenn der NXT dagegen fährt

- 6.3. Die Größe der Hindernisse entsprechen den Kalkziegelsteinen von Hellweg für 89 Cent
7. Am Ende der Mission muss ein Becher auf dem Zielpunkt positioniert sein
 - 7.1. Der Durchmesser des Ziels beträgt mindestens 15 cm.
 - 7.2. Im Umfeld von 30 cm um das Ziel befinden sich keine Hindernisse.

Kapitel 5

Projekt- / Zeitplanung

5.1 Gruppenaufteilung

Ausgehend von der Anforderungserhebung kann das Projekt in drei große Teilaufgaben gegliedert werden:

- Implementation des Mission Control Centers
- Konstruktion der NXT's und Implementation der für die Roboter benötigten Routinen
- Implementation der für die NAO's benötigten Routinen

Diese drei Teilaufgaben wurden durchgehend von jeweils einer Gruppe von 2-4 Studenten bearbeitet, wobei ein Student im speziellen die Kommunikation zwischen dem MCC und dem NAO-Roboter bearbeitet hat. Durch diese Einteilung konnte - nachdem die exakten Schnittstellen zwischen den Gruppen, bzw. den Programmen festgelegt wurden - die Projektarbeit auf drei Kleingruppen heruntergebrochen werden. Innerhalb dieser Kleingruppen wurden die anstehenden Aufgaben in der Regel gemeinschaftlich oder zumindest in durchgängiger Absprache durchgeführt.

5.2 Grobe Ablaufplanung

In der Anforderungserhebung ist bereits die zu entwickelnde Funktionalität beschrieben - darauf basierend wird die Mission mit folgendem Ablauf implementiert:

1. Die NXT's erkunden die Umgebung in ausreichendem Maße, so dass für den NAO ein Weg zum Ziel existiert. Die Informationen werden dem MCC übermittelt und in einer Karte visualisiert, um den Ablauf der Mission verfolgen zu können.
2. Das MCC hat die Möglichkeit die auftretenden Ungenauigkeiten zu reduzieren, indem eine Kalibrierung eines NXT's durch einen NAO erfolgt.
3. Nachdem die Karte ausreichend erkundet ist und ein Weg zum Ziel berechnet wurde, wird der NAO, geführt von einem NXT, zum Ziel gebracht um dort den Becher abzustellen. Formal ist der Ablauf der Mission in einer "State-Machine" implementiert (s. dazu "Statusorientierter Ablauf").

II Projektumsetzung

6 MCC - Mission Control Center	13
6.1 Statusorientierter Ablauf	13
6.2 Architekturmodell	14
6.2.1 Map Model	14
6.2.2 NAO/NXT Model	14
6.2.3 Controller / View	14
6.3 Verfahren zur Problemlösung	14
6.3.1 Kommunikation	14
6.3.2 Auswertung der Kalibrierung	15
6.3.3 Geführte Erkundung	15
6.3.4 Pfad zum Ziel	16
7 NXT	18
7.1 Entwurf	18
7.1.1 Software	18
7.1.2 Idee 1 → Modell 1	19
7.1.3 Idee 2 → Modell 2	20
7.1.4 Idee 3 → Modell 3	21
7.1.5 Fazit und Entscheidung	23
7.2 Kommunikation	23
7.2.1 Bluetooth	23
7.2.2 Kommunikationsprotokoll PC ↔ NXT	23
7.2.3 Kommunikation mit dem MCC	24
7.3 Logik	24
7.3.1 Explorationsalgorithmen	25
7.3.2 GoToPoint	26
8 NAO	28
8.1 Vortüberlegungen	28
8.2 Software NAO	28
8.2.1 NAOqi	28
8.2.2 Choreographie	29
8.2.3 Weitere Software	29
8.3 Hardware NAO	29
8.4 Messungen	30
8.4.1 Kamera	30
8.4.2 Sonar	30
8.4.3 NAO Walk	30
8.4.4 Auswertung	31
8.4.5 Fazit	31
8.5 Kalibrierung	32
8.5.1 NAOMarker	32
8.5.2 Marker Konstruktion für NXT	32
8.5.3 Auslesen der übergebenen Werte	33
8.5.4 Farberkennung	34
8.5.5 Triangulation	35
8.6 NAOWalk mit RedBall Tracking	35
8.7 Probleme & Lösungen	37
8.7.1 NAO	37
8.7.2 NAOMarker	38
8.7.3 Triangulation	38
8.7.4 Farberkennung	38

Kapitel 6

MCC - Mission Control Center

6.1 Statusorientierter Ablauf

Um die Lösung der Aufgabe zu strukturieren, aber auch um die Möglichkeit zu haben, die Mission in Teilaufgaben zu unterteilen, ist der Ablauf grundsätzlich in 5 verschiedene Status unterteilt. Jeder einzelne Status besteht aus einem definierten Ziel und dementsprechenden Abbruch-, bzw. Abschlusskriterien. Die Implementierung ist in Form einer “META-State-Machine” vorgenommen, die den (inneren) Zustand der Mission kennt und bei Änderung diesen an die einzelnen Teilsysteme propagiert. Die im folgenden beschriebenen Status geben die konzeptionellen Bedingungen und Aktionen innerhalb dieses Status wieder, es werden jedoch keine konkreten Abläufe, bzw. Implementierungen dieser Aktionen erläutert. Eine genaue Beschreibung der Abläufe wird in späteren Kapiteln nachgeholt. Die einzelnen von der State-Machine verwalteten Status sind:

1. Initial

Der Initial-Zustand wird automatisch von allen (Teil-)Systemen zu Beginn der Mission eingenommen ohne von der State-Machine propagiert werden zu müssen. Der Initial-Zustand stellt grundsätzliche einige Vorbedingungen sicher, damit die eigentliche Mission gestartet werden kann. Damit die Kalibrierung (s. u.) der NXT's durch den NAO möglichst geringe Fehler produziert, wird zunächst der NAO aufgestellt (falls er nicht bereits steht) und anschließend mit Hilfe von zwei NXT's auf eine Position innerhalb der Karte kalibriert. Anhand dieser Daten werden auch die Positionen der NXT's bestimmt - sollte es noch weitere NXT's geben, werden diese im Abschluss ebenfalls kalibriert. Für den Übergang in den nächsten Status muss der NAO stehen und alle NXT's und der NAO müssen kalibriert sein.

2. Autonomic Exploration

Die Autonomic Exploration ist die erste Phase in der das unbekannte Gebiet erkundet wird. Zur Erkundung verwenden die NXT's verschiedene Explorationsalgorithmen (s. dazu die Kapitel zum NXT) mit dem Ziel, innerhalb einer vorgegebenen Zeit einen möglichst großen Bereich zu erkunden. Die Zeit die für diese Phase veranschlagt ist beträgt 5:00 Minuten. Dieser Wert ist allerdings sehr von der gestellten Aufgabe abhängig - je nach Größe des zu erkundenden Gebietes sollte der Wert entsprechend angepasst werden, damit nach Ablauf der Zeit ein adequater Bereich erkundet wurde.

3. Guided Exploration

In dieser Phase wird zielgerichtet versucht, nicht erkundete Gebiete von den NXT's abfahren zu lassen. Da nicht bekannt ist, wie groß das gesamte zu erkundende Gebiet ist oder welche Ausmaße es hat, muss man Areale, die noch abgefahrene werden sollen, nach gewissen Kriterien aussuchen (s. hierzu Geführte Erkundung). Die Punkte die dann letztendlich noch abgefahrene werden sollen, werden direkt vom MCC an die entsprechenden NXT's geschickt. Maßgeblich für den Übergang in die nächste Phase ist, dass das Ziel gefunden wurde. Mögliche Adaptionen sind bei Kenntnis des Gebietsumfangs die Erkundung bis ein bestimmter Schwellwert an erkundetem Gebiet relativ zum Gesamtgebiet überschritten wird und das Vorhandensein eines ausreichend großen Weges zum Ziel.

4. Path Verification

Nach der zweiten Erkundungsphase wird der vom MCC berechnete Pfad zum Ziel (s. hierzu Pfad zum Ziel) durch einen NXT verifiziert. Es soll dabei sichergestellt werden, dass der berechnete Weg ausreichende Ausmaße hat, um vom NAO abgelaufen zu werden. Sollte der Weg als ungenügend erkannt werden, wird zur vorigen Phase zurück gesprungen. In dieser kann zielgerichtet versucht

werden entweder den berechneten Pfad zu ändern, einen Neuen zu finden oder ein größeres Gebiet zu erkunden, um neue Wege zu finden.

5. NAO Walk

In der letzten Phase wird der NAO von einem NXT zum Ziel geführt. Der NXT wird dabei vom NAO als Fixpunkt genutzt und bewältigt die Strecke abschnittsweise (s. hierzu NAOWalk).

6.2 Architekturmodell

Bevor eine grobe Architektur entworfen werden kann, muss man sich klar machen, welche Funktionen das zu implementierende Stück Software leisten soll. Aus der Anforderungserhebung geht hervor, dass eine Karte existieren muss, in die die erhaltenen Erkundungsinformationen des NXT eingetragen werden können und die in vereinfachter Form an die NXT's geschickt werden kann. Zusätzlich ist es hilfreich, wenn auch nicht zwingend gefordert, dass während des Missionsablaufs die aktuelle gesamte Karte visualisiert wird, um den Missionsfortschritt und das Verhalten der Roboter transparent zu machen. Aus diesen Vorüberlegungen ist eine Implementierung der Model-View-Controller Architektur naheliegend. Im Modell werden NXT, NAO und die Karte abstrakt repräsentiert; Es gibt verschiedene Sichten auf die enthaltenen Daten, abhängig davon für wen die Information bestimmt ist.

6.2.1 Map Model

Im Map Model wird eine vereinfachte Darstellung der Karte und die Zielposition gespeichert. Bei der Karte wird dabei lediglich modelliert ob ein gewisser Bereich frei ist (also nicht durch ein Hindernis belegt) und wie wahrscheinlich es ist, dass er frei ist. Die Karte besteht aus einem Grid von gleichgroßen Kästchen die folgendermaßen geändert werden: In regelmäßigen Abständen erhält das MCC vom NXT eine Information über seine aktuelle Position sowie seinen Ausrichtungswinkel, relativ zu seiner Startposition. Die Informationen werden insofern in der Karte eingetragen, als dass die durch den NXT belegten Kästchen zum Zeitpunkt der Übermittlung seiner Position, um 1 erhöht werden (initial: 0). Bei der Visualisierung der Karte bestimmt die Zahl auf den einzelnen Kästchen die Farbgebung und repräsentiert die Wahrscheinlichkeit, dass das Kästchen nicht durch ein Hindernis belegt ist. Die Strategie der Erkundung besteht also nicht darin Hindernisse zu umfahren, sondern sich auf als frei vermuteten Flächen zu bewegen. Die Größe der Karte ist unabhängig von der Maßgabe in der Anforderungserhebung dynamisch, d.h. sie ist vollkommen losgelöst von der Größe des tatsächlichen Areals (unter anderem auch, weil nicht bekannt ist an welcher Position innerhalb des Gebiets die Roboter initial platziert werden). Bei Bedarf wird die Karte automatisch um eine quadratische, so genannte MapSection erweitert.

6.2.2 NAO/NXT Model

Die Modelle der beiden Robotertypen (im Speziellen gibt es für jeden einzelnen Roboter eine eigene Instanz) halten zum einen die aktuelle Position innerhalb der Karte, sowie im Falle des NXT-Modells noch Informationen über die letzte Kalibrierung, sowie eine Spur der gefahrenen Strecke. Diese wird verwendet um nach einer Kalibrierung die Daten der Karte gegebenenfalls zu korrigieren (s. dazu Auswertung der Kalibrierung).

6.2.3 Controller / View

Die Aufgabe des Controllers ist es, Änderungen der Daten an das Model zu propagieren, damit es zu jedem Zeitpunkt aktuell ist. Der View verwendet die Daten des Modells, um den Anforderungen entsprechende Darstellungen zu erzeugen.

6.3 Verfahren zur Problemlösung

Die Aufgabe des MCC besteht nicht nur darin eine geregelte Kommunikation zwischen den Robotergruppen und eine Visualisierung des Missionsablaufs zu ermöglichen, sondern auch gewisse Problemstellungen während der Mission zu lösen. Die einzelnen Problemstellungen werden im Weiteren detailliert erläutert.

6.3.1 Kommunikation

Die Kommunikation zwischen den beiden Robotergruppen und dem MCC ist zentral im MCC implementiert. Der generelle Aufbau entspricht grob einer Client-Server Architektur, wobei das MCC als Server

funktioniert und jeder einzelne Roboter als Client angemeldet wird. Nach einer erfolgreichen Anmeldung steht den einzelnen Instanzen der Roboter ein Satz von Funktionen zur Verfügung (im Falle der NXT wird tatsächlich lediglich eine Funktion aufgerufen um Daten zu übermitteln - in diesen ist der Zweck genau definiert), bzw. das MCC hat die Möglichkeit Funktionen in den Roboterinstanzen aufzurufen. Diese Funktionen implementieren die durch die Anforderungserhebung formulierten Aktionen, die während der Mission durchgeführt werden. Für den Aufbau einer Verbindung sendet jede Roboter-Instanz ein "Connection-Request" an das MCC - dieses wird durch die Vergabe eines Handles an den Roboter quittiert. Mit diesem Handle kann der Roboter nun das MCC ansprechen um Daten zu übermitteln. Die Funktionen für NAO'S und NXT's müssen nicht durch die Angabe des spezifischen Modells beschrieben werden, das übergebene Handle ist für die Erkennung ausreichend. Lediglich bei der initialen Registrierung wird der Typ und im Falle des NXT die Farbe, die für die Erkennung durch den NAO wichtig ist, übergeben. Das Handle selbst ist ein Integer, der mit jeder Registrierung inkrementiert wird.

6.3.2 Auswertung der Kalibrierung

Bei der Arbeit mit Robotern gehört die Erkennung, Verminderung oder sogar Behebung von Fehlern zu den grundlegenden Aufgaben bei der Lösung von Problemstellungen, die mit der Feststellung der Position des Roboters verbunden sind. Da es eine wesentliche Teilaufgabe dieses Problems ist, eine simple, aber doch möglichst genaue Karte der Umgebung zu erstellen, ist die Bestimmung der Position, bzw. Abweichungen während der Bewegung, von essentieller Wichtigkeit. Als Ausgangspunkt kann festgehalten werden, dass jeder der Roboter mit jeder Bewegung einen gewissen Fehler, im Bezug auf seine tatsächliche Position und die Position an der er sich selbst vermutet, erzeugt. Es kann weder davon ausgegangen werden, dass der Fehler unerheblich ist, noch dass er sich über einen gewissen Zeitraum oder gewisse Distanzen im Mittel selbst auslöscht. Um diesen Fehler zu verringern, werden die einzelnen NXT's zu gewissen Zeitpunkten durch den NAO kalibriert. Die Kalibrierungen finden nach der Phase Autonome Erkundung und Geführte Erkundung statt. Der NXT fährt dazu in den Sichtbereich des NAO's und der NAO berechnet die aktuelle Position des NXT's (s. dazu NAO: Kalibrierungsphase). Anhand der Position die der NXT für seine aktuelle hält und der vom NAO berechneten Position, kann eine relative Abweichung berechnet werden. Zum einen ist dies wichtig, damit ausgehend von dieser Position der NXT wieder bei einem Fehler von annähernd 0 startet (natürlich ist auch die Berechnung des NAO's in der Regel fehlerbehaftet), des Weiteren wird diese Abweichung verwendet um die gesamte, seit der letzten Kalibrierung oder dem Start abgefahrene Strecke, zu korrigieren. Der Algorithmus unterteilt dazu die seit der letzten Kalibrierung gefahrene Strecke in äquidistante Abschnitte und verschiebt die einzelnen Abschnitte in Richtung des bei der Kalibrierung ermittelten Wertes. Hierbei wird die Strecke in sofern gewichtet, als das Abschnitte nahe dem Startpunkt nur sehr wenig korrigiert werden und Abschnitte nahe dem Endpunkt sehr stark korrigiert werden. Diese Gewichtung erfolgt gleichmäßig über die gesamte Strecke, wobei am Startpunkt keine Verschiebung statt findet und der Endpunkt vollständig zu dem bei der Kalibrierung ermittelten Wert geschoben wird.

6.3.3 Geführte Erkundung

Die größte Schwierigkeit bei der geführten Erkundung ist, festzustellen, welche Bereiche des Gebiets für den weiteren Verlauf der Mission relevant sind, insbesondere da nicht bekannt ist wo die Grenzen des Gebietes liegen und wo das Ziel ist - falls es bisher noch nicht gefunden wurde. Davon ausgehend, dass die NXT's sich während der autonomen Erkundung zumindest ungefähr im Bereich des Ziels befinden (diese Annahme ist ausgehend von der Anforderungserhebung im Bezug auf die Ausmaße des Gebiets und die Geschwindigkeit der NXT's gerechtfertigt) wird bei der geführten Erkundung versucht das bereits gefahrene Gebiet so zu erweitern, dass eine möglichst zusammenhängende erkundete Fläche entsteht. Das Prinzip veranschaulicht 6.1.

Der Algorithmus verwendet die Daten vom Map-Modell und berechnet den Mittelpunkt aller bisher befahrenen Punkte. Von diesem Punkt ausgehend wird im Rotationsprinzip jeder Punkt der noch nicht befahren wurde ermittelt, wobei solange Punkte gesucht werden bis ein gewisser Schwellwert erreicht wird. Dieser Schwellwert kann abhängig von den Ausgangsbedingungen angepasst werden. Anschließend gibt es eine Menge von Punkten die Grundlage für die geführte Erkundung sind. Im nächsten Schritt wird die Menge aller Punkte in so viele Gebiete unterteilt wie es NXT's gibt. Ausgehend von der aktuellen Position der NXT's werden diese Gebiete zugeordnet und die NXT's fahren einzelne Punkte ab, um einen möglichst großen Teil der Ihnen zugeordneten Gebiete zu erkunden. Dabei kann nicht gefordert werden, dass die NXT's jeweils das ganze ihnen zugeordnete Gebiet erkunden, da nicht bekannt ist ob sich in den Bereichen Hindernisse befinden. Die Maßgabe für das erfolgreiche Beenden der Phase ist daher, dass das Ziel so wie ein Weg dorthin gefunden wurden.

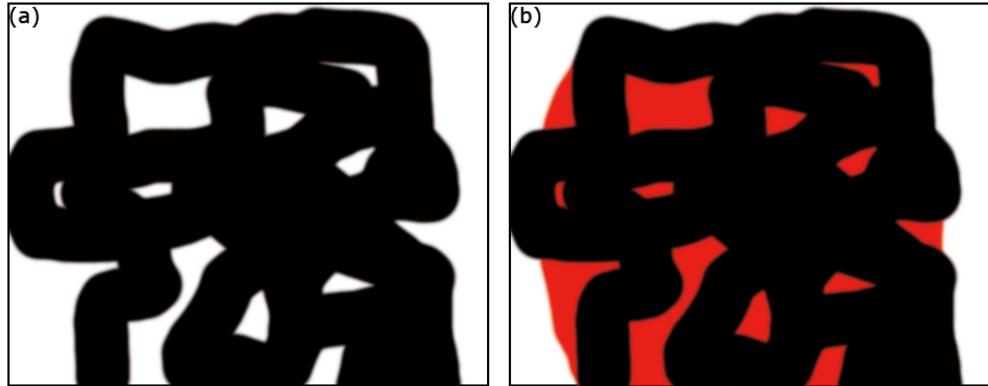


Bild 6.1: (a) Karte nach der autonomen Erkundung (b) Der rote Bereich markiert das Gebiet, dass in der geführten Erkundung hinzukommen soll

6.3.4 Pfad zum Ziel

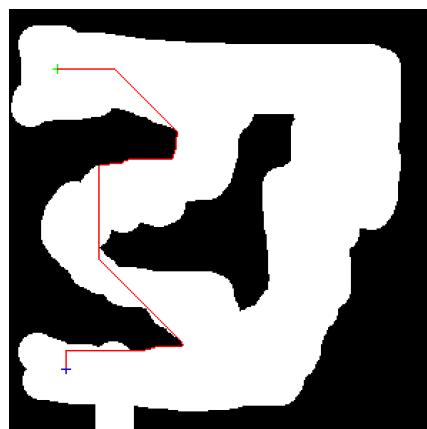
Überblick

Im folgenden wird der Pathfinding Algorithmus näher beschrieben bei welchem aus einer erkundeten Karte und einem Start und Zielpunkt ein optimaler Pfad berechnet wird. Bedingungen hierfür sind jedoch eine erkundete Topologie. Übergeben wird diese als binärer Zweidimensionaler Array, sowie einem Startpunkt des Nao's und dem gefundenen Endpunkt. Der Algorithmus teilt sich in 3 Phasen:

1. Die Breitensuche, die dafür sorgt ein gültigen Pfad vom Start, zum Endpunkt zu finden.
2. Die Repositionierung des Pfades um ein möglichst großen Abstand zu den Hindernissen zu erreichen.
3. Die Pfadoptimierung, die eine Minimierung der Punkte zum Ziel hat. Das Resultat ist ein optimierter Pfad vom Start, zum Zielpunkt. Repräsentiert wird dieser durch eine Liste von Punkten.

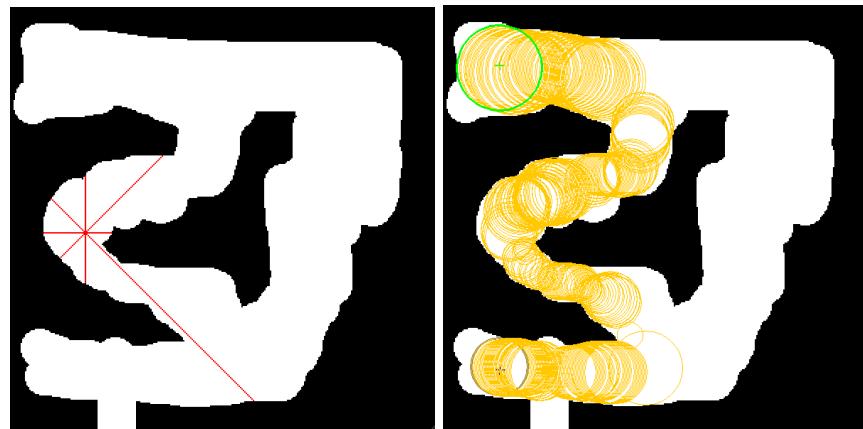
Breitensuche

Bei der Representation der Karte handelt es sich um ein 2-Demisionalnen Binären Array, wobei 1 für ein Hindernis stehen und 0 für einen Freifläche. Die Breitensuche geht nun folgendermaßen vor: Sie weist iterativ jedem Punkt die Entfernung zum Startpunkt zu. Dies geschieht indem jeder Punkt am Horizont, also der noch nicht betrachtet wurde und an einen bereits betrachteten Punkt grenzt, den Abstand des bisherigen Punktes + 1 annimmt. Dies geschieht so lange der Endpunkt noch nicht gefunden wurde. Ist ein Endpunkt gefunden Konstruiert man den Pfad vom Endpunkt zum Startpunkt aus, in dem jeweils immer der kleinste Folgepunkt dem Pfad hinzugefügt wird, bis man bei dem Startpunkt angekommen ist. Das Resultat ist ein Pfad an Punkten.



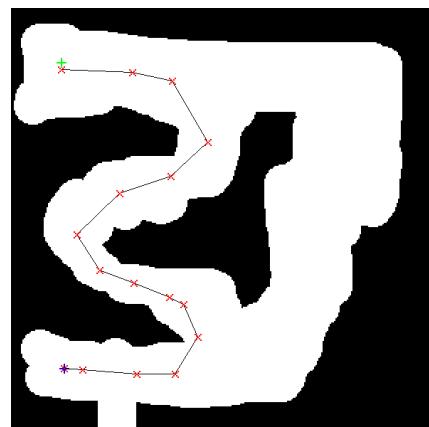
Repositionierung

Die Breitensuche sollte nun einen Pfad zurückliefern. Dieser Pfad ist jedoch auf die minimale Distanz optimiert. Einen Roboter entlang dieses Pfades zu schicken ist nicht möglich, da der Pfad oft an möglichen Hindernissen langführt und die Breite der Roboter nicht berücksichtigt. Da die Position der Hindernisse auf der Karte auch einem Fehler unterliegen ist es sicherer den Roboter in der “Mitte” eines “Ganges” zu schicken. Die Repositionierung sträbt nun an aus dem Pfad der Breitensuche einen solchen optimalen Pfad an, der möglichst weit weg von möglichen Hindernissen ist. Dazu wird durch jeden Punkt des Pfades 4 Linien geführt: Die Horizontale, Vertikale und beide Diagonalen, die jeweils dort enden, wo diese an ein mögliches Hindernis stoßen. Um eine Abschätzung des Optimalen Punktes zu bekommen, wird nun der Mittelpunkt der kürzesten Linie genommen und aus ihr wird ein Kreis mit der Diagonale der Linie konstruiert. Das Ergebnis ist eine Menge M von Kreisen auf der Karte.



Optimierung

In der Optimierungsphase werden alle Kreise aus M , die sich überschneiden, miteinander verbunden. Das Ergebnis ist ein Graph. Auf dem resultierendem Graphen wird nun noch einmal eine Breitensuche vorgenommen, wobei die Länge der Verbindung mit eingerechnet wird. Der resultierende Pfad aus dieser Breitensuche ist der gesuchte Pfad und wird zurückgegeben.



Kapitel 7

NXT

7.1 Entwurf

7.1.1 Software

Beim Entwurf der Software für unseren Teil der Aufgabe hatten wir zwei Dinge zu beachten, die eingeschränkte Rechenleistung¹ des LEGO[©] Mindstorms[©] NXT Brick (im folgenden nur noch Brick genannt) und die durch LEGO[©] begrenzte Anzahl von Robotern auf maximal 4.

7.1.1.1 nxc – Not eXactly C

nxc ist die für LEGO[©] NXT native Programmiersprache mit einem Compiler, welche für diversen Plattformen erhältlich ist. Obwohl die Syntax von nxc der Programmiersprache C ähnlich ist, ist sie in ihrem Umfang erheblich eingeschränkt. Aus dem Fehlen des Pointerkonzept resultiert unter anderem der Verlust auf die Speicherverwaltung direkt einwirken zu können.

7.1.1.2 nxt-python-framework

Das nxt-python-framework² agiert als Interface, welches die von LEGO[©] in dem „Bluetooth Development Kit“ veröffentlichten direkten Kommandos, nutzt um mit der Hardware zu interagieren.

Wie wird dies erreicht?

Mit LEGO[©] NXT ist es möglich, dass sich bei maximal vier NXTs einer zum Master erklärt. Der Master kann nun durch speziell kodierte Befehle die anderen drei NXTs fernsteuern. Dieses Verhalt macht sich das nxt-python-framework zu nutze und täuscht maximal 3 NXTs vor, dass es ein NXT-Master sei. Wenn dies geschehen ist können die NXTs von PC-Seite ferngesteuert werden.

Vorteil: Durch die Nutzung von nxt-python integriert sich die Komponente NXT-Erkunder nahtlos in das übrige System.

Nachteil: Der synchrone Start bzw. Stopp von zwei Motoren ist nur schwerlich realisierbar, da zwei Befehle benötigt würden, die nacheinander verschickt und auf NXT-Seite nacheinander ausgewertet werden würden.

Wegen dem immensen Vorteil der einfachen Integration in das Restsystem und dem Nachteil der asynchronen Ansteuerung von Motoren entstand die Idee die beiden Programmiersprachen (nxc und python) zu kombinieren.

7.1.1.3 hybrider Ansatz

Die Kombination von nxc und nxt-python wurde wie folgt realisiert. Es wurde ein einfaches Kommunikationsprotokoll (siehe Abbildung 7.5 auf Seite 24) konzipiert durch welches der Aufruf von in nxc implementierten Funktionen durch nxt-python ermöglicht wird.

¹8-Bit ARM mit 48 MHz Takt, 64KB RAM

²<http://code.google.com/p/nxt-python/>

7.1.2 Idee 1 → Modell 1

7.1.2.1 Idee

Unsere erste Idee bestand im Prinzip aus zwei unabhängigen Ideen. Zum Einen wollten wir ein Fahrgestell konzipieren, das auch bei unwegsamen Gelände eine kontrollierte Bewegung des Explorer ermöglichen würde und zum Anderen wollten wir einen Sensor der schon viele Informationen über die Umgebung sammelt ohne, dass der Explorer jeden Quadratzentimeter abfahren muss.

7.1.2.2 Konstruktion

Die Konstruktion bestand aus einem kettengetriebenen Fahrzeug, welches mit Hilfe eines Radars seine Umgebung wahrnahm (siehe Abbildung 7.1 auf Seite 19). Die Ketten waren dabei fest gespannt um mögliches schlüpfen der Kette über die Achse zu vermeiden und so eine Ungenauigkeit bei der Fahrt zu vermeiden. Des Weiteren bestanden sie aus Gummi und hatten eine große Auflagefläche zum Boden und somit eine möglichst hohe Reibung zum Boden. Der Radar bestand aus einem Motor der über einige Zahnräder und einer Schnecke einen Ultraschallsensor bewegte.

Verbaute Sensoren und Motoren:

- zwei Motoren für den Antrieb
- ein Motor für den Radar
- ein Kompass-Sensor zur Ermittlung der Ausrichtung des Fahrzeuges
- ein Lichtstärke-Sensor zur Zielfindung

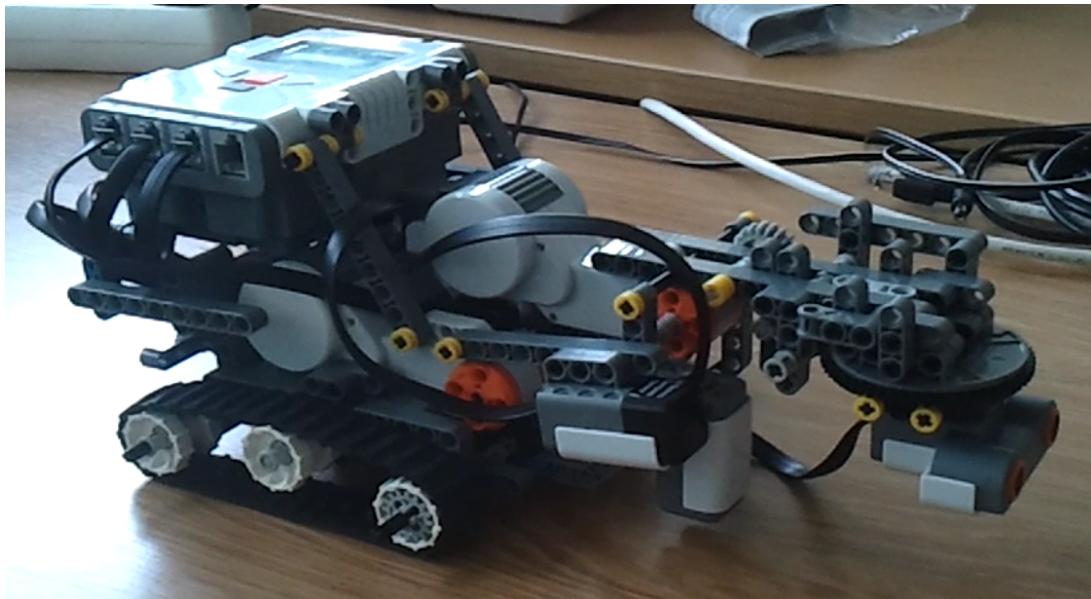


Bild 7.1: Modell 1

7.1.2.3 Test

Getestet haben wir sowohl das Fahrverhalten des Kettenfahrzeugs als auch die Möglichkeit mit dem Radar die Umgebung wahrzunehmen. Hierbei wurde besonders Wert darauf gelegt die Abweichung zum Ist-Wert zu ermitteln.

Beim Fahrzeug war es wichtig herauszufinden ob es geradeaus fahren kann und ja mit welcher Abweichung auf verschiedenen Distanzen (20cm, 50cm, 1m) zu rechnen war. Des Weiteren musste ermittelt werden ob das Fahrzeug in der Lage war sich auf der Stelle zu drehen und dies möglichst genau nach der Vorgabe eines vorher angegebenen Winkels. Hierfür wurden mehrere Test mit verschiedensten Winkeln durchgeführt bei denen das Fahrzeug sich so oft drehen musste bis es wieder auf seine Ausgangsposition angelangt ist z.B. vier mal eine Drehung um 90° im Uhrzeigersinn. Der Radar wurde auf seine Genauigkeit getestet, als auch auf sein Verhalten bei verschiedenen Materialien und Winkel zu den verschiedenen Objekten.

7.1.2.4 Pros & Cons

Pros:

- geringe Abweichung bei der Fahrt durch die hohe Reibung der Gummiketten
- hohe Geländetauglichkeit durch Kettenantrieb
- kennt das Gebiet vor sich und kann vorausschauend fahren

Cons:

- hohe Abweichung beim drehen
- sehr hohe Abweichung des Ultraschallsensors wenn nicht im 90° zum Hindernis
- hohe Ungenauigkeit beim drehen des Radarkopfes durch das Getriebe

7.1.2.5 Fazit

Die Idee ist alles in allem nicht schlecht aber die Umsetzung mittels LEGO® ist nicht praktikabel. Die hohen Ungenauigkeiten und die komplett Aussetzer des Ultraschallsensors lassen uns keine andere Wahl als ein neues Fahrzeug zu entwerfen. Hierbei muss sowohl der Antrieb als auch die Sensorkonstruktion überdacht werden. An einen Einsatz dieses Modells ist nicht zu denken.

7.1.3 Idee 2 → Modell 2

7.1.3.1 Idee

Da wir bei unserer ersten Idee feststellen mussten das ein kettengetriebenes Fahrzeug zu hohe Ungenauigkeiten verursachte, musste hier eine Alternative gefunden werden, welche aber keine Einschränkungen bei der Bewegungsfreiheit des Fahrzeuges macht d.h. möglichst genaues (vorwärts/rückwärts) Fahren und auf der Stelle wenden können.

Auch eine Alternative für den Radar musste her. Hierbei wurde in Kooperation mit dem MCC-Team vereinbart das nicht Hindernisse gefunden werden, sondern davon ausgegangen wird das die gefahren Strecke des Explores frei ist, sozusagen wurde das Bild der Karte invertiert. Dies ermöglichte uns nur auf Hindernisse reagieren zu müssen und nicht wie vorher Informationen über den Bereich des Einsatzgebietes zu sammeln.

7.1.3.2 Konstruktion

Unsere Konstruktion 2 (siehe Abbildung 7.2 auf Seite 21) erhielt nun ein 2-Achsen Antrieb, welcher es uns ermöglichte durch gleichzeitiges ansteuern der Motoren gerade Strecken zu fahren, als auch durch entgegengesetztes ansteuern sich auf der Stelle zu drehen. Als Zusatz wurde noch ein Omni-Wheel verbaut, welches dem Gefährt mehr Stabilität verleihen sollte. Um auf Hindernisse reagieren zu können, wurden an der Vorderseite des Fahrzeugs drei Touchsensoren befestigt. Diese sollten auslösen sobald das Fahrzeug vor ein Hindernis fuhr.

Verbaute Sensoren und Motoren:

- zwei Motoren für den Antrieb
- drei Touch-Sensoren um Hindernisse zu finden
- ein Lichtstärke-Sensor zur Zielfindung

7.1.3.3 Test

Auch beim zweiten Model wurden die oben schon beschrieben Tests für den Antrieb durchgeführt. Die Sensoren wurden in ihrer Anordnung getestet um eine möglichst optimale Anordnung zu finden.

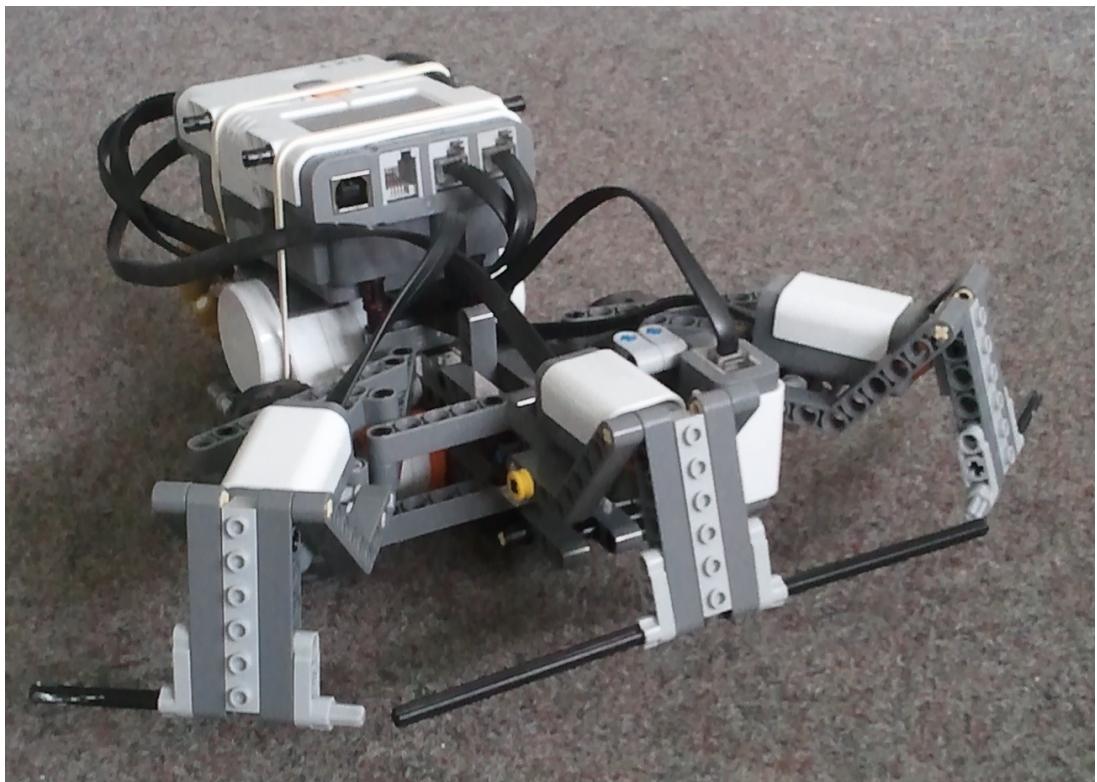


Bild 7.2: Modell 2

7.1.3.4 Pros & Cons

Pros:

- geringe Abweichung beim fahren
- geringe Abweichung beim drehen
- hohe Aussagekraft bzgl. Sensorevents da nur Touch-Sensoren verbaut wurden

Cons:

- keine Aussagen über das Gebiet vor dem Fahrzeug möglich
- Fahrzeug kann nur noch reagieren und kaum intelligent handeln
- Verhaken der Touch-Sensorkonstruktion

7.1.3.5 Fazit

Bei der zweiten Konstruktion überzeugt das Antriebsmodell durch seine geringen Abweichungen und seine Agilität. Die Sensoranordnung hingegen lässt noch einige Wünsche offen. Die fehlende Voraussicht ist dabei das größte Manko. An einen Einsatz dieses Model ist ebenfalls nicht zu denken. Eine Verbesserung des Sensorkonstrukts ist nötig.

7.1.4 Idee 3 → Modell 3

7.1.4.1 Idee

Das in Idee 2 entwickelte Antriebsmodell hat uns überzeugt. Im dritten Anlauf wird sollte nur noch die Sensoranordnung überdacht werden. Dem Bereich vor dem Fahrzeug sollte dabei mehr Beachtung geschenkt werden, um bessere und intelligenter Entscheidungen treffen zu können.

7.1.4.2 Konstruktion

Bei der dritten Konstruktion (siehe Abbildung 7.3 auf Seite 22) wurde das Antriebsmodell der zweiten Konstruktion übernommen. Bei den Sensoren wurde der mittlere Touch-Sensor durch ein Ultraschall-Sensor ersetzt. Dieser ermöglichte es die freie Strecke vor dem Fahrzeug zu ermitteln.

Verbaute Sensoren und Motoren:

- zwei Motoren für den Antrieb
- zwei Touch-Sensoren um Hindernisse zu finden
- ein Ultraschall-Sensor um den Bereich vor dem Fahrzeug zu überblicken.
- ein Lichtstärke-Sensor zur Zielfindung

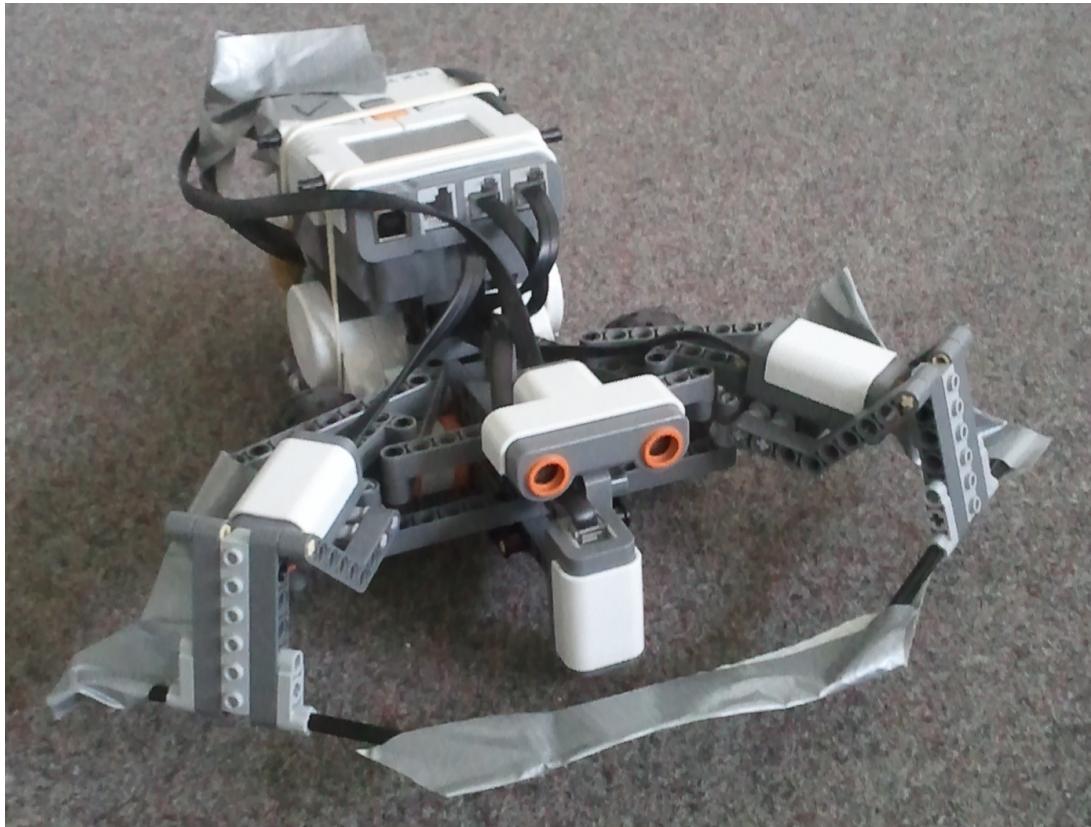


Bild 7.3: Modell 3

7.1.4.3 Test

Auch beim dritten Model wurden alle Antriebstests durchgeführt. Die neue Sensoranordnung wurde mit mehreren Testaufbauten auf ihre Einsatztauglichkeit geprüft. Hierbei wurden mögliche Anordnungen von Hindernissen konstruiert und die Sensorevents ausgewertet.

7.1.4.4 Pros & Cons

Pros:

- geringe Abweichung beim fahren
- geringe Abweichung beim drehen
- freie Strecke vorm Fahrzeug bekannt

Cons:

- Verhaken der Touch-Sensorkonstruktion

7.1.4.5 Fazit

Das dritte Model überzeugt durch seinen Antriebsmodel, sowie durch die verbesserte Sensoranordnung gegenüber des zweiten Models. Aber auch dieses Model ist mit Vorsicht zu benutzen. Die Möglichkeiten des Verhaken der Fahrzeuge ist ein großes Problem das es noch zu beseitigen gibt. An einen Einsatz ist nur bedingt zu denken.

7.1.5 Fazit und Entscheidung

Das erste Model ist für raues Gelände bestens geeignet. Durch seine hohen Abweichungen beim Fahren und des nicht zuverlässigen Ultraschall-Sensors im Radar aber kein Kandidat für die Lösung unseres Problems.

Das zweite Model überzeugt durch seine Genauigkeit beim Fahren und auch die Aussagekraft der Sensoranordnung. Allerdings fehlt uns hier das gewisse etwas um ein möglichst elegante Lösung für die autonome Erkundung eines Gebietes.

Von allen Model schneidet das dritte am besten ab. Es erbt die guten Fahrteigenschaften des zweiten Models und biete uns dazu noch die Möglichkeit durch seine überarbeitete Sensoranordnung möglichst intelligent das Problem zu lösen.

7.2 Kommunikation

7.2.1 Bluetooth

Bluetooth ist ein Funkstandard speziell für die Nahbereichskommunikation. Er wurde in den 1990er-Jahren von der Bluetooth Special Interest Group (SIG) entwickelt und in IEEE 802.15.1 festgeschrieben. Grundsätzlich sind verbindungslose und verbindungsorientierte Übertragungen möglich.

Bei der Konzeptionierung von LEGO[©] Mindstorms[©] NXT hat LEGO[©] die Verbindungsorientierung für ihre Geräte festgelegt.

7.2.2 Kommunikationsprotokoll PC ↔ NXT

Bei einem der Treffen im Rahmen des Semesterprojektes wurde durch den begleitenden Professor angeregt, dass durch das Team NXT sicherzustellen sei, dass alle Steuerkommandos und Antworten (über Bluetooth) auch von der Gegenstelle erhalten werden müssen. Als Denkanstoß verwies er auf den 3-way-handshake des TCP. Dieser Aspekt wurde von uns mit dem Protokoll aus Abbildung 7.4 implementiert, sodass auf jede Nachricht vom Typ „m“ eine Antwort vom Typ „r“ folgte und der Sende diese Typ „r“ Nachricht mit einer abschließenden Typ „a“ Nachricht quittierte. Dieses Protokoll hatte eine Datenflut zur Folge, die kaum zu beherrschen war, da bei überschreiten von Fristen Nachrichten wiederholt gesandt werden mussten.

Da bei LEGO[©] NXT Bluetooth verbindungsorientiert verwendet wird, war die oben beschriebene Herangehensweise nicht nötig, da bei Verbindungsorientiertheit die Zustellung von Nachrichten garantiert wird. Aus diesem Grund wurde das Protokoll auf das in Abbildung 7.5 reduziert.

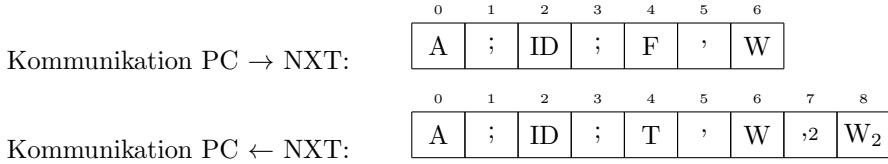
Definitions- und Wertebereich der im jüngsten Protokoll (Abbildung 7.5) verwendeten Platzhalter:

Kommunikation PC → NXT:

F	W	Beschreibung
0	beliebig	Stopp, der NXT in Hardware bricht alle seine Aktionen ab und wartet auf weitere Kommandos
1	0 – MAX_INT	Vorwärtsfahren: bei W = 0 bis Steuerkommando 0 oder Auslösen eines Sensors, bei W > 0 vorwärts fahren für W Grad in Motorumdrehungen
2	1 – MAX_INT	rückwärts fahren für W Grad in Motorumdrehungen
3	1 – MAX_INT	Fahrzeugachse um W Grad nach links drehen
4	1 – MAX_INT	Fahrzeugachse um W Grad nach rechts drehen
5	beliebig	Messung mit dem Ultraschallsensor ausführen

Kommunikation PC ← NXT:

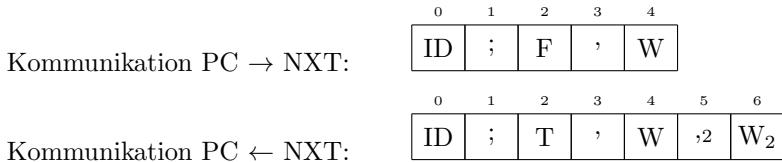
T	W	Beschreibung
1	0 – MAX_INT	ständige Standortmeldung: für W cm vorwärts gefahren (ohne Kollision)
2	1 – MAX_INT	Kollision: W cm vorwärts gefahren, Sensor W_2 hat ausgelöst
3	1 – MAX_INT	Fahrziel erreicht: für W cm vorwärts gefahren (ohne Kollision)
4	beliebig	Drehung/Rückwärtsfahren beendet
5	0 – 255	Ergebnis der Messung mit dem Ultraschallsensor
9	beliebig	Bodensensor hat Ziel gefunden



Legende:

- A = Typ der Nachricht (m = Nachricht, r = m erhalten, a = r erhalten)
- F = Funktion die aufgerufen werden soll
- W = Zahlwert
- T = Typ der Antwort
- ₂ = optional

Bild 7.4: Kommunikationsprotokoll 3-way-handshake



Legende:

- F = Funktion die aufgerufen werden soll
- W = Zahlwert
- T = Typ der Antwort
- ₂ = optional

Bild 7.5: Kommunikationsprotokoll

7.2.3 Kommunikation mit dem MCC

Für die Kommunikation mit dem MCC kam das Python Framework Twisted³ zum Einsatz. Twisted ist ein sogenanntes „event-driven“ framework, das heißt der Programmierer stellt für Aktionen Callback-Funktionen bereit. Twisted abstrahiert von der darunterliegenden Netzwerkarchitektur und dem verfügbaren Netzwerkprotokollstack (wenn auch nicht vollständig transparent), dies vereinfacht die Programmierung eines Verteilten Systems erheblich.

7.3 Logik

Für die logische Programmierung des „Explorers“ bestand vorwiegend das Problem, dass die Hardware (der Brick) Zeit benötigt um die Aktionen auszuführen, also blockiert, jedoch die Softwarekomponente asynchrones Messaging nutzt. Um diese Hürde zu überwinden, wurde bei der Konzeptionierung des Systems darauf geachtet, dass auf jedes Steuerkommando, welches eine Handlung der Hardware hervorruft, nach Abschluss der Handlung eine Antwort des Brick gesendet werden muss. Dass versetzte uns in die Lage mit einer synchronisierten boole'schen Variablen (blockiert = True) die Software solange zu blockieren, bis die Antwort vom Brick eingegangen ist und damit die boole'sche Variable wieder auf False gesetzt wird.

³<http://twistedmatrix.com>

7.3.1 Explorationsalgorithmen

In der Welt der autonomen Rasenmäh- und Staubsaugroboter haben sich vier Algorithmen durchgesetzt⁴:

1. Touch and Go⁵
2. circle
3. radar
4. Wandverfolgung

1 – 3 werden im Folgenden näher besprochen. 4 konnte aufgrund der begrenzten Anzahl an Sensoren nicht implementiert werden und bleibt deshalb außen vor.

7.3.1.1 Exploration – simple

Der erste und auch gleichzeitig einfachste Algorithmus. Der Roboter fährt solange geradeaus bis er auf ein Hindernis trifft. Nach dem Auslösen eines Sensors wechselt der Roboter beliebig seine Fahrtrichtung und fährt wieder geradeaus bis zum nächsten Ereignis. Der Explorer wechselt die Richtung nicht vollständig beliebig, er wählt die Drehrichtung in Abhängigkeit von dem auslösenden Sensor, soll heißen wenn der linke Sensor angeschlagen wurde, dreht sich der Explorer um einen beliebigen Winkel zwischen 30° und 160° nach rechts.

7.3.1.2 Exploration – circle

Wie in Abbildung 7.6 zu sehen ist macht der Explorer eine Bewegungsabfolge in Form einer eckigen Nautilussschnecke. Durch diese Bewegungsfolge wird der Bereich in dem sich der Explorer befindet schnell erkundet.

In Abbildung 7.7 wird an Hand eines Zustandsautomaten gezeigt, wie der Explorationsalgorithmus umgesetzt ist. Es gibt keinen ausgezeichneten Endzustand, was damit begründet ist, dass der Algorithmus von einem externen Zähler abgebrochen wird. Der Abbruch kann in jedem der Zustände geschehen. Man kann erkennen, dass zusätzlich zur reinen Bewegung weitere Zustände und Transitionen eingefügt wurden um Messungen mit dem Ultraschallsensor durchzuführen. Die Messungen dienen als Grundlage zur Berechnung eines Korrekturfaktors für alle Bewegungen.

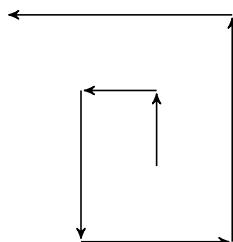


Bild 7.6: Bewegungsmuster Exploration – circle

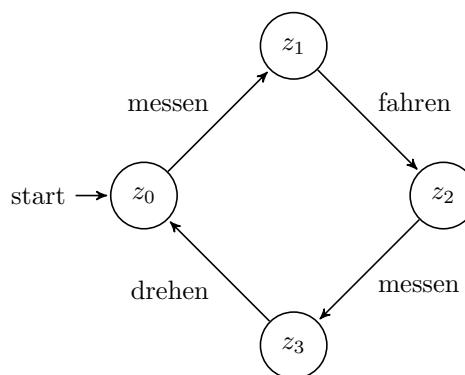


Bild 7.7: state-machine Exploration – circle

⁴Quelle: Roboter mit Mikrocontrollern, Heinz Schmid, Franzis Verlag, 2011

⁵hier simple genannt

7.3.1.3 Exploration – radar

Wie in Abbildung 7.8 zu sehen ist, macht der Explorer eine Bewegungsabfolge in Form einer eckigen Wellenausbreitung. Durch diese Bewegungsfolge wird der Bereich vor dem Explorer schnell erkundet und der Explorer verlässt langsam seinen bisherigen Bereich.

Wie im Abschnitt Exploration – circle wird hier auch ein Zustandsautomat (Abbildung 7.9) zur Darstellung des Algorithmus verwendet. Er unterliegt den gleichen Kriterien wie oben.

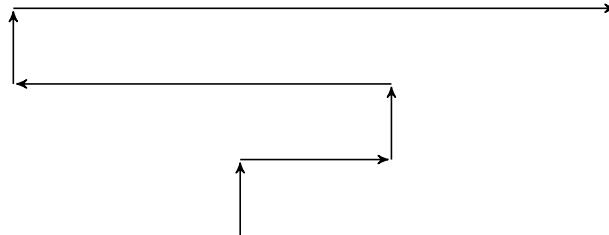


Bild 7.8: Bewegungsmuster Exploration – radar

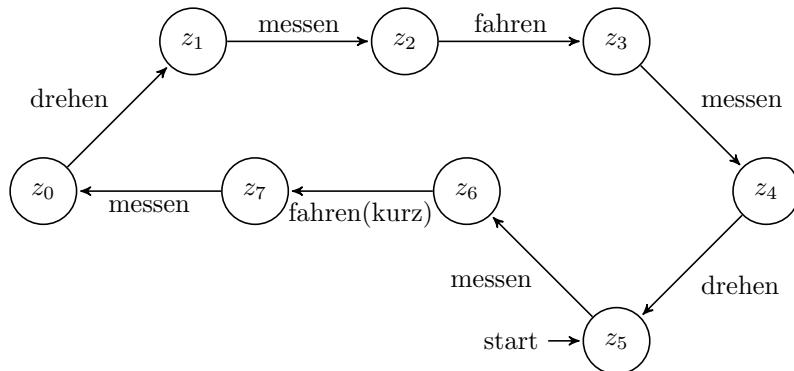


Bild 7.9: state-machine Exploration – radar

7.3.2 GoToPoint

Für die GoToPoint – Funktionalität wurden die zwei Funktionen aus Listing 7.1 und Listing 7.2 benötigt. Mit der Funktion berechnePunkt() kann aus der eigenen Ausrichtung, der gefahrenen Entfernung und dem ehemaligen Standort der neu Standort berechnet werden. Die Funktion berechneVektor() aus Listing 7.2 berechnet zu zwei gegebenen Punkten den verbindenden Vektor, dazu wird erst ein relatives Koordinatensystem geschaffen, dann wird über den Satz von Pythagoras die Entfernung, d.h. der Betrag des Vektors berechnet und danach Winkel berechnet.

Listing 7.1: berechnePunkt()

```

1 def berechnePunkt(ausrichtung, entfernung, standort = {'x':0.0, 'y':0.0}):
2     return {'x': standort['x'] + entfernung * GRAD2CM *
3             math.cos(ausrichtung * (math.pi / 180.0)),
4             'y': standort['y'] + entfernung * GRAD2CM *
5                 math.sin(ausrichtung * (math.pi / 180.0))}
  
```

Listing 7.2: berechneVektor()

```
1 def berechneVektor(standort = {'x':0.0, 'y':0.0}, ziel = {'x': 0.0, 'y': 0.0}):
2     relativ_ziel = {'x': abs(ziel['x'] - standort['x']),
3                     'y': abs(ziel['y'] - standort['y'])}
4     dbg_print("St0: (%d,%d), Z: (%d,%d), rZ: (%d,%d)"%(standort['x'],
5                                                       standort['y'],
6                                                       ziel['x'],
7                                                       ziel['y'],
8                                                       relativ_ziel['
9                                         x'],
10                                         relativ_ziel['
11                                         y']), 1, 0)
10    entfernung = math.sqrt(relativ_ziel['x'] ** 2 + relativ_ziel['y']
11                           ** 2)
11    if relativ_ziel['x'] == 0:
12        winkel = 90
13    elif relativ_ziel['x'] < 0:
14        winkel = math.atan(float(relativ_ziel['y']) / float(
15            relativ_ziel['x'])) *
16            (180.0 / math.pi) + 180
16    else:
17        winkel = math.atan(float(relativ_ziel['y']) / float(
18            relativ_ziel['x'])) *
19            (180.0 / math.pi) + 360
20    return {'winkel': winkel % 360,
21            'entfernung': entfernung,
22            'rel_x': relativ_ziel['x'],
22            'rel_y': relativ_ziel['y']}
```

Kapitel 8

NAO

8.1 Vorüberlegungen

Aus dem Szenario des Gesamtprojekts gingen bei unseren Vorüberlegungen zwei Hauptaufgaben für das NAO Team hervor. Zum einen ist das die Kalibrierung der NXT-Roboter während der Erkundungsphase auf unbekanntem Gebiet. Und zum anderen muss die Zielfindung des humanoiden NAO-Roboters, nachdem das Gebiet erkundet und eine konkrete Route zum Hotspot gefunden wurde, eingeleitet werden. Unter der Kalibrierung der NXT Roboters soll verstanden sein, dass der NAO Roboter mit Hilfe seiner Sensoren den NXT lokalisieren und diese berechnete Entfernung dem MCC mitteilen soll. Eine Kalibrierungsanfrage erfolgt immer nur aus Richtung des MCC an den NAO und zwar dann, wenn ein NXT “verloren gegangen ist”, d.h. dieser keine genauen Daten mehr liefert. Zunächst einmal bietet der Hersteller Aldebaran auf seiner Website¹ eine recht umfangreiche Dokumentation. Diese ist, wie wir nach längerer Zeit aber feststellen mussten teilweise sehr schlecht. Es ist oft nicht klar, was die Daten, die man z.B. über die Sensoren abgreifen kann tatsächlich bedeuten. Dieses Problem trat beispielsweise bei den NAOMarks oder bei dem Sonar auf. In dem Abschnitt Probleme und Lösungen wird darauf noch eingegangen werden. Gemeinsam mit den anderen Teams überlegten wir uns als einheitliche Programmiersprache Python zu verwenden, da sie sowohl von dem NXT-Mindstorm Robotern, als auch vom NAO unterstützt werden. Es hat sich allerdings herausgestellt das für die Programmiersprache C++ deutlich mehr Beispiele auf der Aldebaran Webseite vorliegen und die Dokumentation dafür besser ausgeführt ist. Bei Aldebaran gibt es eine API, die es erlaubt vordefinierte Schnittstellen z.B. der Sensoren des NAOs zu benutzen.

8.2 Software NAO

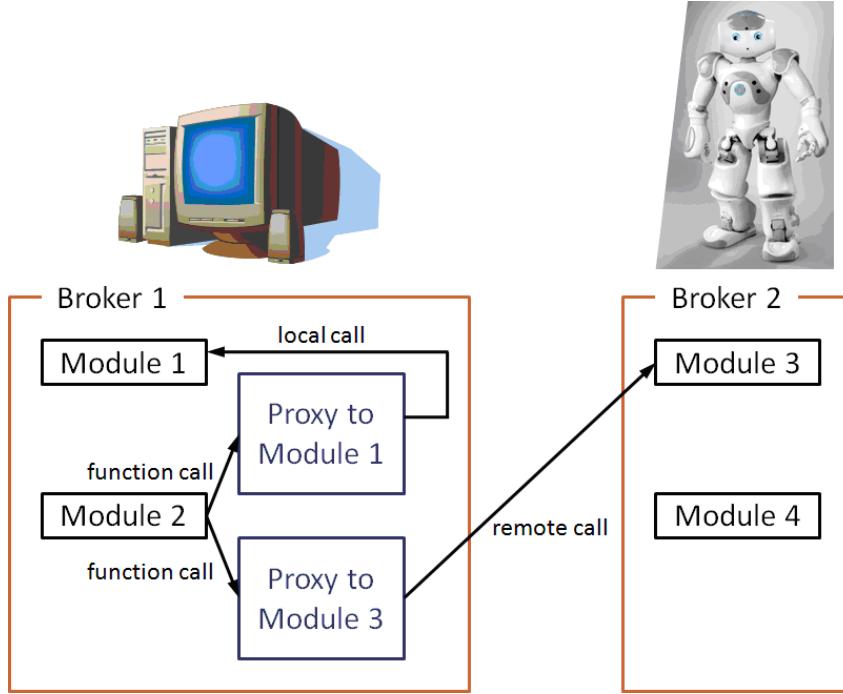
8.2.1 NAOqi

NAOqi ist ein Framework von Aldebaran, was auf dem NAO läuft und den Roboter steuert. Es ist hardware- und plattformunabhängig. Für verschiedene Programmiersprachen wie z.B. C++ oder Python stellt Aldebaran ein Software Development Kit (SDK for NAOqi) bereit, um das Implementieren von Programmen auf dem NAO in verschiedenen Sprachen möglich zu machen. Jede Maschine, auf der die NAOqi läuft, erschafft einen Broker. Jeder Broker verwaltet die Module, die innerhalb von ihm ausgeführt werden. Wenn ein Modul aufgebaut ist, bekommt es einen Zeiger auf den Broker übergeben. Module können Funktionen der anderen Module durch einen Proxy, der durch den Broker erzeugt wird, aufrufen. Der Broker abstrahiert die Netzwerk-Schnittstelle auf den Maschinen, so dass, wenn eine Funktion über einen Proxy aufgerufen wird, diese Funktion möglicherweise auf einer anderen Maschine im Netzwerk ausgeführt werden kann.

Innerhalb des NAOqi Prozesses laufen verschiedene Module. Wir verwenden diese Module:

1. ALLandMarkDetection: wird verwendet, um NAOMarks zu finden und zu erkennen
2. ALMemory: zum ergebnisbasierten Speichern von Daten
3. ALMotion: ansprechen der Motoren zum Bewegen des NAO
4. ALVideoDevice: erlaubt Zugriff auf die beiden Kameras des NAO

¹<http://www.aldebaran-robotics.com/documentation/index.html>



8.2.2 Choreographe

Choreographe ist ein graphisches Programmierinterface. Durch das Auswählen und Verbinden von vorgefertigten Funktionsboxen kann man schnell und intuitiv erste Programme erzeugen und direkt auf dem NAO oder mit der Software NAOsim testen. Alle Hauptfunktionen des NAOs wie beispielsweise die Steuerung aller Gelenke können mit verschiedensten Konfigurationen getestet werden. Dadurch ist dem Benutzer schnell ermöglicht erste Erfahrung mit dem Umgang des Roboters zu gesammelt. Wenn eine Verbindung zu einem NAO vorhanden ist, kann über ein dreidimensionales Robotermodell jeder Motor einzeln angesprochen werden. Darüber hinaus kann in einem weiteren Fenster das aktuelle Kamerabild angezeigt werden.

8.2.3 Weitere Software

NAOsim

NAOsim ist ein Simulationsprogramm, womit ein vollständiger NAO simuliert werden kann. Es ist möglich Testumgebungen selbst zu erschaffen, indem man sich beispielsweise einen Raum mit Hindernissen einrichtet. Dadurch sind auch Tests an einen realen NAO möglich. Auf einen realen Test sollte man allerdings keinesfalls verzichten, da NAOsim immer vollständig exakte Werte und Bewegungen des NAOs simuliert.

Monitor

Dies ist ein Programm, dass das aktuelle Kamerabild oder alternativ die Sensordaten des NAOs auslesen und aufzeichnen kann. Das Programm ist vor allem dann hilfreich, wenn mit der Detektion von NAO-Markern gearbeitet wird. Es ist möglich über Monitor einzelne Einstellungen der Kamera wie beispielsweise Helligkeit, Kontrast, Gain, Auflösung, Sättigung oder Farbton vorzunehmen.

8.3 Hardware NAO

In unseren Tests und für das Szenario selbst benutzen wir einen NAO H21 Body. Dieser zeichnet sich durch verschiedene Sensoren aus. Es sind zwei Infrarotsensoren, drei Trägheitsmesser (2x Schwingungs- und 1x Beschleunigungssensoren), KontaktSENSoren für Brust, Füße und Kopf, Sonar (2 Empfänger und 2 Sender), 32 Positionssensoren, ein Force Sensing Resistor und vier Mikrofone, zwei Kameras, sowie zwei Lautsprecher eingebaut. Das vollständige Datenblatt kann man online bei Aldebaran anschauen².

²http://www.technik-lpe.eu/fileadmin/bilder/produkte/aldebaran/NAO_H21_Next_Gen.pdf

8.4 Messungen

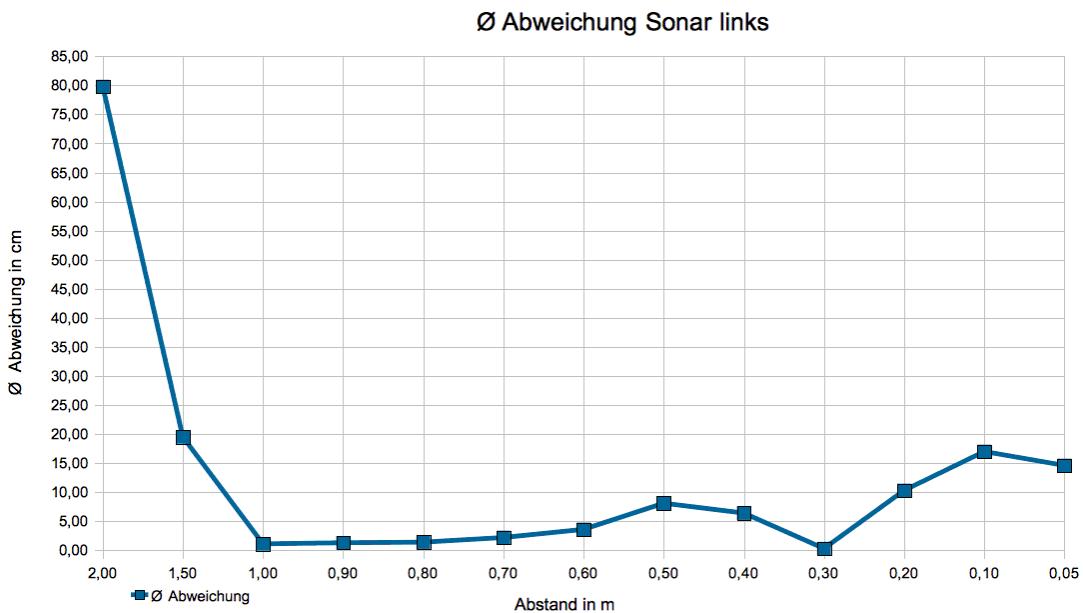
Wie in dem Abschnitt Sensoren bereits beschrieben wurde, besitzt der NAO Roboter einige Sensoren, die wir vor der Benutzung auf die Genauigkeit verifiziert haben. Wir haben Messreihen angelegt für die Vorwärtsbewegung des NAOs, also dem einfachen Laufen für das in dem Roboter integrierte Sonar und haben die Kamera auf Schärfe und Brauchbarkeit untersucht.

8.4.1 Kamera

Eine Herausforderung bei dem NAO stellt die integrierte Kamera dar. Da die Kommunikation drahtlos über ein WLAN-Netzwerk sehr viel besser funktioniert, als mit Kabel, ist nur eine geringe Auflösung des Kamerabildes möglich (VGA). Alternativ kann auch eine höhere Auflösung (1280x960 Pixel) genutzt werden, dazu muss man aber gleichzeitig eine teilweise stark reduzierte Framerate (1 fps) in Kauf nehmen. Zusammen mit der Kamera werden wichtige integrierte Funktionalitäten, wie die Erkennung von NAO-Markern und einem Roten Ball (mit genau definierter Größe), mitgeliefert. Diese werden in den nächsten Abschnitten genauer erklärt.

8.4.2 Sonar

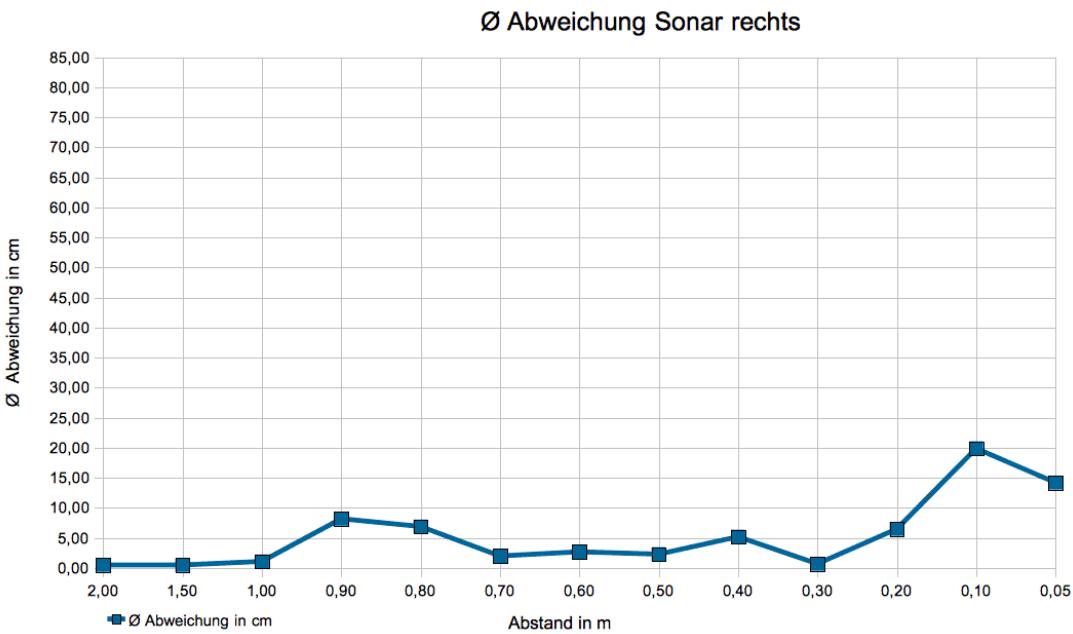
Das Sonar des Roboters lässt sich relativ einfach über das bereits vorhandene Beispiel auf der Aldebaran Website³ auslesen. Dabei gibt es zwei Sonarsensoren im NAO je einen für links und rechts. Die Ergebnisse können in dem hier vorliegenden Diagramm abgelesen werden. Wir habe für jeden Abstand zwischen 2.0 m und 0.1 m, die in 0.1 m Schritten weiter unterteilt sind, jeweils 10 Messungen entnommen und gemittelt. Es ist auffällig, aber nicht weiter verwunderlich, dass beim linken Sonar unseres NAOs die Genauigkeit abnimmt, je weiter er sich vom Hindernis entfernt. Beim rechten Sonar dagegen ist die Genauigkeit überraschenderweise auch bei einer Entfernung von 1,5 m und größer weiterhin sehr gering (Abweichung ± 1 cm). Sowohl bei dem linken, als auch dem rechten Sonar wird die Genauigkeit bei einer geringen Entfernung (± 20 cm) gleichermaßen schlechter. Bei unseren Messungen haben wir keine Differenzierung in der Oberflächenbeschaffenheit unternommen, sodass unsere Messungen ausschließlich mit einer weißen glatten Wand als Hindernis durchgeführt wurden. Aussagen über andere Oberflächen können daher nicht gemacht werden.



8.4.3 NAO Walk

Für das Szenario unseres Semesterprojektes benötigen wir einen Roboter, der möglichst ohne Abweichungen laufen kann, um sein Ziel zu erreichen. Man könnte auch annehmen, dass der NAO Roboter sich auf etwas größere Entfernungen trotzdem noch geradlinig bewegt. Dies ist tatsächlich aber nicht der Fall, da unser NAO schon bei kleineren Entfernungen einen deutlichen Drift nach rechts hatte. Um eine

³http://www.aldebaran-robotics.com/documentation/_downloads/sensors.sonar.py



konkrete Aussage über die Genauigkeit des Laufens des NAOs treffen zu können und der Abweichung ggf. entgegenzuwirken, haben wir verschiedene Messreihen angelegt. Der NAO bietet über die bereits beschriebene mitgelieferte Software Choreographie die Möglichkeit verschiedenste Einstellungen der Laufbewegung einzustellen, was man natürlich auch über das NAOqi selbst implementieren kann. Einstellbare Parameter für den NAO Walk sind intuitiverweise die X- und Y-Bewegung, in der sich die Laufbewegung vorwärts (X positiv), rückwärts (X negativ), links (Y positiv) und rechts (Y negativ) wiederfinden lassen. Des Weiteren gibt es den Parameter Theta, mit der man die Rotation des NAO definieren kann, wobei der Bereich [-1.0, 1.0] eingehalten werden muss und das Minimum (-1.0) die maximale Rotation im Uhrzeigersinn und das Maximum (1.0) die maximale Rotation gegen den Uhrzeigersinn widerspiegelt. Eine weiterere wichtige Einstellung, die man für das Laufen konfigurieren kann, ist die Schrittgröße sowohl für die X- (angegeben als "MaxStepX"), als auch für die Y-Laufrichtung ("MaxStepY"). Dieser Wert liegt im Bereich von [0.1 cm; 6 cm]. Um zu erkennen welchen Einfluss die Schrittweite auf die Genauigkeit des Laufens an sich hat, haben wir jeweils 6 verschiedene Schrittweiten von 1 cm bis 6 cm in jeweils Zentimeterschritten genommen: Schrittweite_i = 1 + i [cm] mit i = 0, 1, 2, 3, 4, 5. Für jede Schrittweite wurden 5 Messungen vorgenommen und die Gesamtstrecke belief sich einmal auf 1 m und zum anderen auf 0.5 m, sodass insgesamt 60 Messungen vorgenommen wurden. Die Ergebnisse sind hier einmal in Tabellenform und einmal als Diagramm deutlich gemacht. Zum Verständnis der Messergebnisse ist hier auch der Versuchsaufbau schematisch dargestellt.

8.4.4 Auswertung

Die Visualisierung des Diagramms macht bei beiden Entfernungen (0.5 m und 1 m) deutlich, dass je kleiner die Schrittweite des NAOs ist, die Abweichungen vom Ziel drastisch erhöhen. Daraus lässt sich schließen, dass beim Laufen eine möglichst hohe Schrittweite (Maximum von 6 cm) genommen werden sollte. Außerdem wird deutlich, dass der NAO fast immer über sein Ziel hinausläuft (dy ist größer 0).

8.4.5 Fazit

Die Messwerte für den NAO Walk sind scheinbar stark abhängig von Roboter selbst. Allerdings kann man sagen, dass die aufgezeigten Abweichungen insgesamt trotzdem relativ klassisch für NAO sind, da andere NAO Body auch immer ein Abdriften verzeichnet haben. Um diese Abweichung beim Laufen zu verringern, könnte man die Messwerte mitteln und auf den Lauf anpassen. Allerdings kann man dadurch das Abdriften auch nicht verhindern, da man davon ausgehen muss das verschiedenste Faktoren, was bei uns nicht in die Messungen selbst mit eingeflossen ist, das Laufverhalten beeinflussen. So ist z.B. nicht komplett vorhersehbar, wie sich der Roboter auf einer völlig anderen Oberfläche als einem Teppich vortbewegt. Auch die Betriebsdauer des NAOs spielt hier eine entscheidende Rolle, da ein Roboter mit heißen oder sogar überhitzten Gelenken deutlich ungenauer in seiner Bewegung ist, als ein kurzzeitig eingeschalteter. Auch wenn das Abdriften, wie bereits erwähnt, scheinbar klassisch für einen NAO ist, sollte man sich in einem Anwendungsszenario für Krisensituationen, wie wir es betrachten, sich nicht aus-

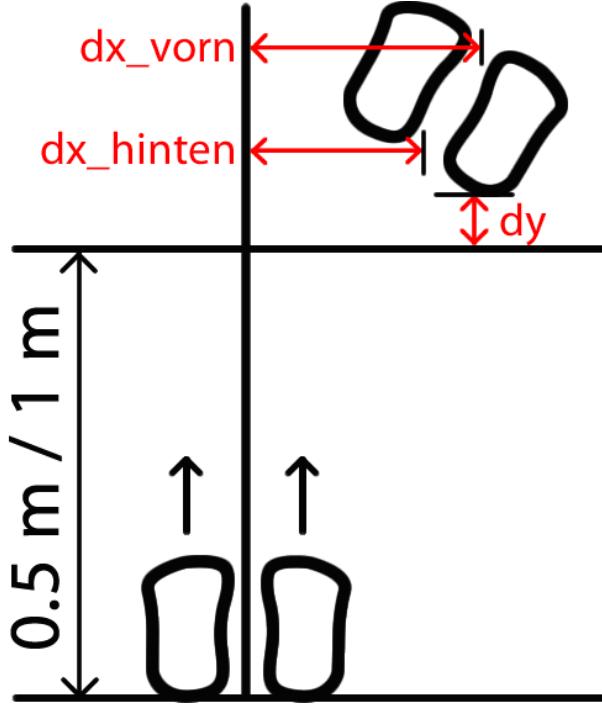


Bild 8.1: Der NAO läuft 0.5 bzw. 1 m; dx_vorn und dx_hinten sind die Abweichungen ausgehend von der Mittellinie zwischen den vorderen bzw. hinteren Füßen (ist der Wert positiv, dann gibt es ein Abdriften nach rechts, andernfalls links); dy gibt an, wie weit der NAO über das Ziel hinausgelaufen ist (ist der Wert positiv, dann ist er zu weit gelaufen, andernfalls zu wenig)

schließlich auf die Messungen stützen und den NAO kalibrieren. Diese Werte sind aber zu NAO-spezifisch und nicht allgemein genug, sodass hier klar geworden ist, dass man sich zur exakten Fortbewegung auf relative Bewegungen zurückgreifen sollte. D.h. der NAO soll sich möglichst selbstständig anhand seiner Umgebung neu orientieren und bei Abweichungen neu ausrichten. Dies könnte man beispielsweise durch Landmarken realisieren, was wir im nächsten Kapitel beschreiben werden.

8.5 Kalibrierung

Die Kalibrierung der NXT erfolgt mit Hilfe der NAO integrierten Kamera. Die Idee besteht darin, den NAO an eine feste Position am Rand des Feldes zu positionieren. Von diesem Punkt aus misst der NAO die Entfernung zum NXT, indem er diese mit seiner integrierten oberen Kamera erkennt und anschließend den Abstand berechnet.

8.5.1 NAOMarker

Um die NXTs überhaupt visuell zu erkennen werden sogenannte NAOMarker benutzt, die von dem NAO bereits standardmäßig erkannt werden. Auf der Aldebaran DVD gibt es genau 10 verschiedene NAOMarker, mit jeweils einer unterschiedlichen ID. Sollten diese Marker nicht ausreichen, werden online noch zusätzliche angeboten, was wir allerdings nicht bestätigen können, da weder über den freien Zugang noch über den persönlichen Login weitere Marker zur Verfügung standen.

8.5.2 Marker Konstruktion für NXT

Da, wie bereits beschrieben, die NAOMarker mit der Erkennung von NXT-Robotern einher geht, haben wir uns Gedanken gemacht, wie ein oder mehrere Marker auf den "Erkundungsroboter" platziert werden können, um diese an möglichst jedem Punkt auf dem unbekannten Feld zu lokalisieren. Wir haben uns dafür entschieden sechs verschiedene Marker pro NXT zu verwenden, was von oben betrachtet einem gleichseitigen Hexagon entspricht. Auf den Bildern X und Y kann man diese noch einmal genauer sehen. Dadurch, dass wir von jeder Seite des NXTs eine andere ID erkennen, die einen NAOMarker codiert, können wir auf 60° genau die Drehung des NXTs unterscheiden. Dies wird unterteilt in 0° bzw. 360° (front), 60° (rechts vorn), 120° (rechts hinten), 180° (hinten), 240° (links hinten), 300° (links vorn). Daraus ergibt sich ein Vorteil: Bei unseren Tests haben wir festgestellt, dass sobald sich die Konstruktion

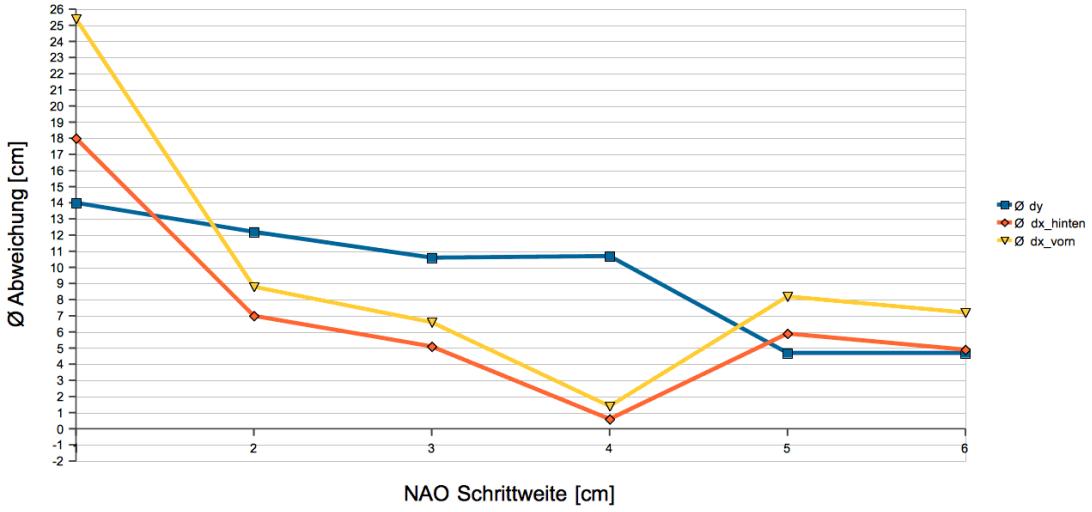


Bild 8.2: Abweichungen bei 0.5 m Entfernung

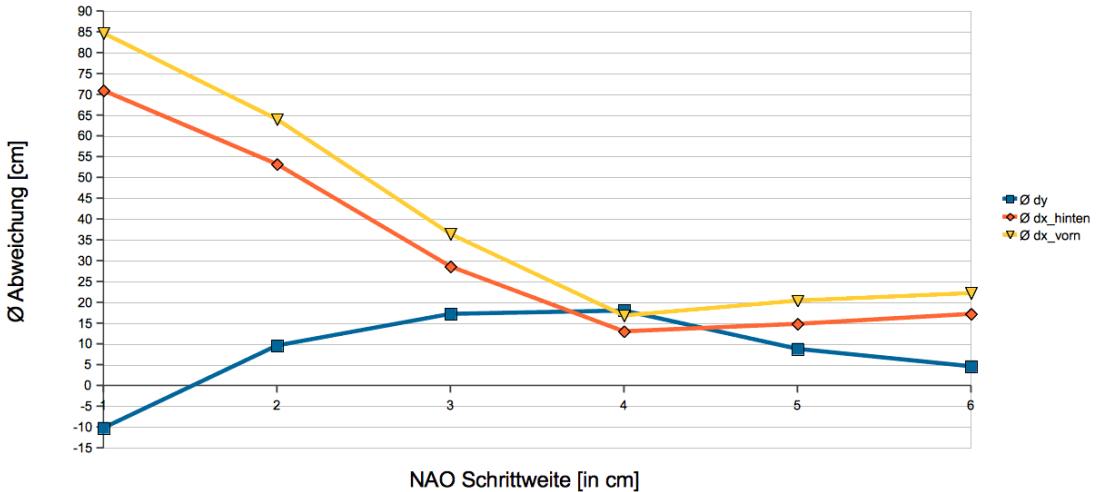


Bild 8.3: Abweichung bei 1 m Entfernung

in erkennbarer Reichweite für den NAO befindet, die Marker immer erkannt werden können, egal in welchem Winkel die Konstruktion zum NAO steht. D.h. ein Marker wird auch dann erkannt, wenn er im schlechtesten Fall in einem Winkel von 30° zur Kamera steht.

Da wir unser Konstrukt auf einen relativ kleinen NXT platzieren und Einsparungen in der Größe der Marker machen wollten, haben wir den Kreis mit Struktur ausgeschnitten und auf den NXT platziert. Bei späteren Tests mit den ausgeschnittenen Markern haben wir allerdings herausgefunden, dass eine richtige Erkennung nur dann möglich ist, wenn er die auf dem Bild X die erkennbare Umrandung beibehält. Die Marker, die wir benutzen, sind wegen Platzgründen nur halb so groß und im Durchmesser ca. 6 cm. Trotz kleinerer NAOMarker werden diese auch noch bei bis zu 160 cm erkannt. Man sollte außerdem wissen, dass die Erkennung der NAOMarker nur dann funktioniert, wenn die Marker wie in den Abbildungen im Hochformat vorliegt. Die Farbe ist allerdings egal nur die Struktur und der beschriebene Rand sind dagegen essentiell. Da wir in unserem Szenario von drei NXTs ausgehen, die das Gelände erkunden und kalibriert werden müssen, bräuchten wir 18 Marker. Weil allerdings, wie bereits erwähnt, nur die 10 Marker vorhanden waren, haben wir uns dazu entschieden farbige Marker für die Unterscheidung zwischen den NXTs zu benutzen. Dies wird im Abschnitt X Farberkennung beschrieben.

8.5.3 Auslesen der übergebenen Werte

Die Markerinformationen können m.H. dieses Python Programmcodes abgegriffen werden: `markerInfo = ALProxy("ALMemory", self.IP, self.PORT).getData("LandmarkDetected")`

Die Daten in Form eines Arrays bei der Marker Detektion können wie folgt interpretiert werden: `markerInfo = [[Zeitstempel_in_s, Zeitstempel_in_ms], [Markerarray_0, Markerarray_1, ..., Markerarray_N], [Mar-`

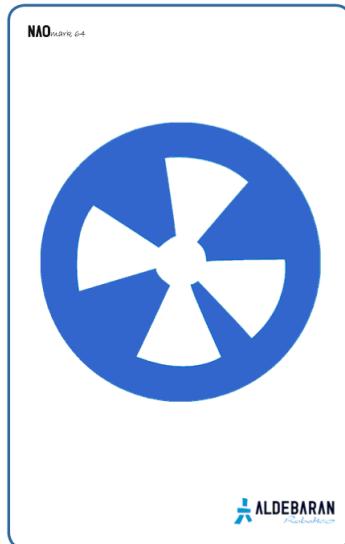


Bild 8.4: Beispiel für einen NAOMarker mit ID 64.

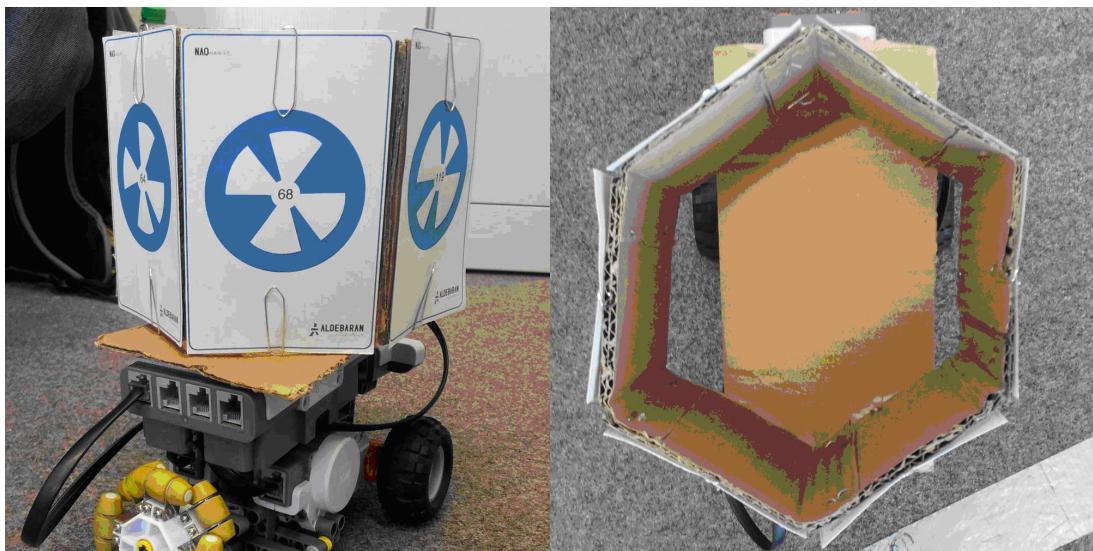


Bild 8.5: Markerkonstruktion für NXT Roboter

kerInfoArray_a], [MarkerInfoArray_b]], Kamerainfo] N ist die Anzahl der erkannten Marker. Je mehr der NAO an Marker erkennt, desto größer wird das Array mit den bereitgestellten Informationen. Markerarray_i, in der sich die spezifischen Daten für die Größe und Winkel der einzelnen Marker befindet, ist wie folgt aufgebaut: Markerarray_i = [[Form, Alpha, Beta, GroesseX, GroesseY, Titel], [MarkerID]]. MarkerInfoArray_a und b sind jeweils sechs unbekannte Werte und auch bei Aldebaran nicht weiter kommentiert oder erklärt. Was die Angaben für Form (shape) und Titel (heading) bedeuten, ist gänzlich unbekannt und die Dokumentation von Aldebaran gibt hier auch keine Informationen. Die Angabe zur Größe des erkannten Markers für X und Y sind zueinander immer gleich (abhängig von der Entfernung des Markers). Die interessanten und für uns wichtigen Werte sind in Alpha und Beta gespeichert. Alpha gibt die Ausrichtung der Kamera nach rechts und links an, also den yaw-Winkel und Beta ist die Ausrichtung für oben und unten, also der pitch-Winkel. Alpha liegt in [-0.3, 0.3], wobei negative Werte rechts und positive Werte links vom Mittelpunkt des Kamerabildes liegen. Beta liegt in [-0.25, 0.1], wobei negative Werte für oben und positive Werte für unten zu interpretieren sind.

8.5.4 Farberkennung

Da es von Aldebaran nur 10 frei verfügbare NAOMarker gibt, aber wir insgesamt 18 Marker brauchen, haben wir uns dafür entschieden jedem NXT die gleichen Marker mit unterschiedlichen Farben zuzuordnen. Dadurch ist es uns möglich über die Farbe den NXT zuzuordnen und über den Marker selbst die Ausrichtung des NXTs herauszufinden. Die Triangulation ruft die Farberkennung auf, sobald ein Marker

gefunden und auf ihn zentriert worden ist, der einem einem NXT zugeordnet werden soll. Die Farberkennung gibt dann die Farbe des zentrierten Markers zurück. Zur Bildverarbeitung greifen wir auf OpenCV zu, das schon auf dem NAO installiert ist und viele vorgefertigte Methoden zum Verarbeiten bereitstellt. Zuerst nimmt die NAO Kamera ein Foto auf und speichert dies auf dem NAO. Zusätzlich wird der Pixelarray von dem Foto gespeichert, aus dem mit der Python Image Library ein Bild erzeugt wird. Danach wird anhand der gemessenen Markergröße ein Rechteck aus dem Bild ausgeschnitten, in dem sich der Marker befindet. Das Bild wird dann zu HSV(Hue, Saturation, Value) konvertiert, um leichter die Farbe bestimmen zu können. Nun wird das Bild für jede Farbe nacheinander auf einen bestimmten HSV Bereich begrenzt bzw die Pixel in dem restlichen Bereich ausgeblendet. Um das Rauschen in den Fotos nicht zu berücksichtigen, wird ein Mindestbereich an Pixeln gewählt. Zusätzlich wird mit dem Pythagoras der Abstand des Bereichs zum Bildmittelpunkt berechnet. Im letzten Schritt werden die Bereiche von jeder Farbe miteinander verglichen und die Farbe mit dem Bereich, welche den kürzesten Abstand zum Mittelpunkt hat, als die wahrscheinlichste Farbe zurückgegeben. Falls keine der vorhandenen Farben gefunden wurde, dann wird dies zurückgegeben.

8.5.5 Triangulation

Die Idee zur Abstandsberechnung durch Triangulation kam uns durch die Arbeit von [Stepan Krivanec et. al. 2009]⁴. In der Quelle wird beschrieben, dass eine Abstandsberechnung mit Hilfe der NAOMarker nur mit zwei NAOs möglich ist oder bei Bewegen des NAOs an eine andere Position, da der Roboter keine Stereo Kamera besitzen. Tatsächlich aber kann man auch über nur einen NAO triangulieren. Die Höhe der NAO-Kamera (obere Kamera) lässt sich einfach ausmessen und ist in der Initialposition, wenn der NAO steht, genau 50 cm vom Boden entfernt. Außerdem werden, wenn ein NAOMarker erkannt wird, u.a. zwei Winkel zurückgegeben, die die horizontalen und vertikalen Winkel angeben, in dem sich die NAO Kamera zum Marker befindet.

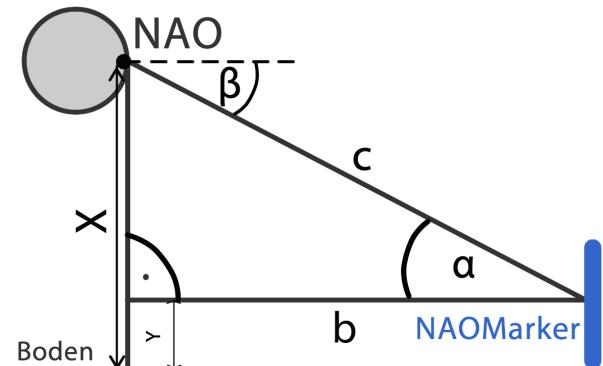


Bild 8.6: Darstellung des Aufbaus zur Markerdetektion

Mit Hilfe der Höhe und des Winkels kann nun Gleichung die Entfernungs berechnung erfolgen. Dazu verwenden wir die einfache trigonometrische Funktion . Die angepasste trigonometrische Funktion lautet: . Der Winkel kann aus dem bereits beschriebenen Markerarray_i als Alpha extrahiert werden. Durch Umstellung erhalten wir den gesuchten Abstand mit . Man beachte, dass in die Berechnung auch die Höhe des Markermittelpunktes Y eingeht und vorher genau ausgemessen werden sollte.

8.6 NAOWalk mit RedBall Tracking

Wie schon im Punkt “4.3 NAOWalk” beschrieben, ist der NAO nicht im Stande selbst kurze Strecken exakt zu laufen. Daher wäre es nicht möglich den NAO sich selbst über die Karte navigieren zu lassen. Um den NAO dennoch zum Ziel zu navigieren, ist es nötig Fixpunkte im Gelände zu haben, an denen er sich orientieren und seine Laufbahn korrigieren kann. Realisiert werden diese Fixpunkte über einen roten Ball, den der NAO visuell erkennen und verarbeiten kann. Dieser ist auf einem der NXT’s befestigt. Der Vorgang der Wegfindung basiert nun auf einem Pfad, den das MCC aus der Karte berechnet. Für diesen Vorgang verweise ich auf den Punkt Pfadberechnung in der MCC-Dokumentation. Letztendlich entsteht dabei ein Pfad, der in Teilabschnitte unterteilt ist, die jeweils nur durch gerade Strecken miteinander verbunden sind.

⁴[ftp://cmp.felk.cvut.cz/pub/cvl/articles/prusa/Krivanec-TR-2010-21.pdf](http://cmp.felk.cvut.cz/pub/cvl/articles/prusa/Krivanec-TR-2010-21.pdf)

Entlang dieses Pfades wird der NXT mit dem roten Ball auf die bestimmten Positionen geschickt und der NAO folgt ihm. Um zu verhindern dass der NAO und NXT sich gegenseitig in die Quere kommen werden die einzelnen Abschnitte des Pfades noch wie folgt in Phasen aufgeteilt, wobei die Ausgangsbedingung ist, dass sich der NXT eine Position weiter auf dem Pfad befindet als der NAO:

P0: Sofern kein NXT mit rotem Ball in Sicht ist, versuche einen zu finden
P1: NAO läuft gerade bis auf einige cm Abstand auf den NXT zu
P2: NAO speichert die Position des NXT
P3: NAO meldet dem MCC vor dem NXT angekommen zu sein
P4: MCC gibt NXT den Befehl, auf die nächste Position zu fahren
P5: NAO läuft auf die gespeicherte Position und richtet sich neu nach dem NXT aus

Nach diesem Ablauf werden alle Teilabschnitte des Pfades abgearbeitet, bis der NAO sich im Zielgebiet befindet.

Im Folgenden werden die einzelnen Fähigkeiten erklärt, die der NAO mitbringen muss, um oben genannte Phasen erfolgreich abschließen zu können.

P0:

Der NAO muss entscheiden können, ob er einen roten Ball sieht oder nicht. Hierbei wird das NAOQi API Modul ALRedBallTracker benutzt, welche an das Event redBallDetected() der ALRedBallDetection des ALVision-Moduls gekoppelt ist. ALRedBallTracker bietet die Funktion isNewData() an, welche true zurückgibt falls ein neuer roter Ball seit der letzten Abfrage erkannt worden ist. Um zu entscheiden, ob ein Ball erkannt wird, benutzt die Methode hasBall() der NAOWalk-Klasse diese Funktionalität.

Liefert hasBall() false, so scannt er sein Blickfeld, indem er methodisch seinen Kopf derart bewegt, dass das maximal mögliche Sichtfeld abgedeckt wurde. Hat er danach noch immer nichts gefunden, dreht er sich um seine senkrechte Körperachse um 90 Grad und wiederholt den Vorgang bis er sich vollständig um die eigene Achse gedreht hat. Findet er letztendlich einen Ball, so richtet er sich nach diesem mit der privaten Methode _turnToBall() aus. Diese Funktionalität bietet die retrieveBall()-Methode aus der NAOWalk-Klasse.

P1:

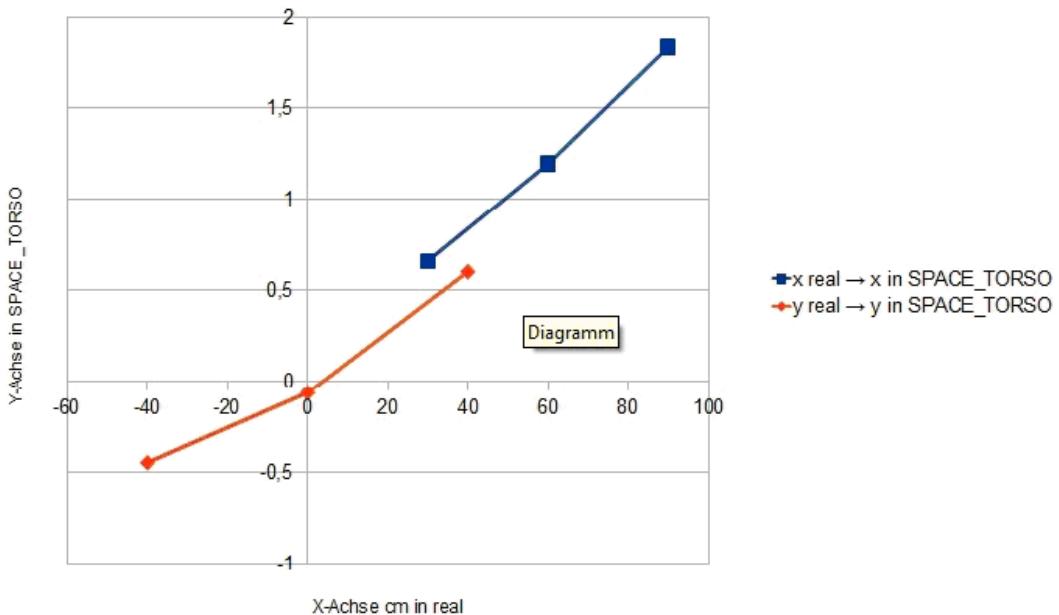
Zusätzlich zu der Grundfunktionalität des Laufens in eine gegeben Richtung, welche über das ALMotion-Modul mittels der Methode walkTo() gegeben wird, muss der NAO hierbei seine Laufbahn stetig korrigieren können. Da walkTo() dies per se nicht ermöglicht, muss eine Abweichung von der Laufbahn festgestellt werden können. Dies wird über eine Abfrage des Drehwinkels des Kopfgelenks realisiert. Die visuelle Erkennung des roten Balls ermöglicht es dem NAO, diesen permanent in der Mitte seines Sichtfeldes zu halten. Hierbei wird nur der Kopf bewegt. Bewegt sich der NAO relativ zum Ball, neigt bzw. dreht er den Kopf. Ab einer bestimmten Abweichung des Drehwinkels zum Nullpunkt ist davon auszugehen, dass der NAO seine gerade Bahn zum NXT verlassen hat und er wird gestoppt, richtet sich neu nach dem NXT mit turnToBall() aus und läuft erneut los.

Da der Sichtbereich der oberen Kamera des NAO's bei aufgerichtetem Körper zu weit von ihm entfernt endet, um nah genug an den NXT heran zu kommen ohne ihn aus dem Blick zu verlieren, muss während der Annäherung ab einer bestimmten Distanz auf die untere Kamera umgeschaltet werden. Ein Beugen des Oberkörpers zu diesem Zweck führt zu einer erheblichen Instabilität der gesamten Laufanimation, sodass ein Hinfallen des Roboters unausweichlich ist. Dies geschieht über die beiden privaten Methoden _setTopCamera() und _setBottomCamera() der NAOWalk-Klasse.

Zur Berechnung der Distanz wird die Methode getPosition() von ALRedBallTracker benutzt. Hierzu muss noch der SPACE_TORSO eingeführt werden: NAO kennt verschiedene karthesische Räume, wie zum Bsp. SPACE_TORSO, bei welchem ein 3D-Koordinatensystem in die Körpermitte des NAO gelegt wird. getPosition() liefert [x,y,z]-Koordinaten des Balls in SPACE_TORSO zurück, wobei x die Distanz nach vorne bzw hinten, y nach rechts bzw links und z die Höhe ist.

Um festzustellen, ob die Rückgabewerte von getPosition() für eine automatisierte Verarbeitung geeignet sind, wurde eine Testreihe angelegt, deren Ergebnis dem Diagramm unten zu entnehmen ist. Real bezeichne dabei die in der realen Welt gemessenen Werte.

Es ist deutlich zu sehen, dass der Werteverlauf bei x sowie y linear ist. z kann hierbei ignoriert werden, da es für die Berechnung des Abstandes irrelevant ist. Die geringen Abweichungen ergeben sich aus Messungenauigkeiten. Daraus folgt, dass die cm in SPACE_TORSO sich aus den Rückgabewerten mittels einer linearen Funktion in cm in real interpolieren lassen. Ergo ipso facto lassen sich exakte Werte zur Weiterverarbeitung gewinnen.



Da die Höhe des Balls konstant bleibt, lässt sich der Abstand aus der Auslenkung der x- und y-Achse interpolieren. Dies geschieht mit einem simplen Pythagoras und ist in die private Methode `_getDistance()` ausgelagert. Diese gesamte Funktionalität ist in der `walkUpToBall()`-Methode aus der NAOWalk-Klasse implementiert.

P2:

Um die Position eines NXT zu speichern legt der NAO ein Datenfeld mit den Koordinaten der `getPosition()`-Methode zum Zeitpunkt der Speicherung mittels `_safePosition()` an. Nun kann er diese Daten benutzen, wenn sich der NXT schon auf die nächste Position bewegt hat.

P5:

Aus der vorher gespeicherten Position relativ zum NAO in SPACE_TORSO, kann der NAO nun interpolieren, wie weit er sich bewegen muss um exakt auf dieser Position zu stehen. Dies geschieht in der `walkToPosition()`-Methode der NAOWalk-Klasse. Dieser Abstand ist sehr kurz und kann daher ohne Korrektur vom NAO gelaufen werden. Sollte sich der NAO noch nicht im Zielgebiet befinden, richtet er sich mit `retrieveBall()` erneut nach dem NXT mit dem roten Ball aus und der Phasendurchlauf beginnt von vorne.

8.7 Probleme & Lösungen

8.7.1 NAO

Am Anfang haben wir uns immer direkt mit einem LAN Kabel zu dem NAO verbunden, wobei diese Verbindung sehr störanfällig war. Die Konnektivität erfolgte später über WLAN. Dies war sehr viel stabiler und störunanfälliger, jedoch kam es häufig trotzdem zu Verbindungsproblemen, die meistens mit einem Neustart des NAO behoben werden konnten. Die Sensoren des NAO waren bei uns teilweise sehr anfällig und ungenau. Wenn der NAO längere Zeit mit Stiffness stand, dann haben sich die Motoren so stark erhitzt, dass der NAO erst abkühlen musste bevor er weiter verwendet werden konnte. Teilweise ist auch das NAOqi grundlos abgestürzt und der NAO ist daraufhin einfach hingefallen.

8.7.2 NAOMarker

Der NAO sollte eigentlich maximal 6 Marker gleichzeitig erkennen, jedoch hat er bei uns auch mehr Marker gleichzeitig erkannt. Die Markererkennung arbeit auf einem bestimmten Bereich vor dem NAO, ca. zwischen 20 cm und 140 cm, akkurat. Wenn der Marker jedoch näher oder weiter entfernt ist, dann erkennt der NAO den Marker entweder überhaupt nicht oder verwechselt ihn mit einem anderen Marker.

8.7.3 Triangulation

Die Triangulation schwankt durch die Ungenauigkeit der Markererkennung, wodurch die Abweichung bis zu 10 cm betragen kann.

8.7.4 Farberkennung

Die Bilder der Kamera sind durch die verschiedenen Beleuchtungsverhältnisse sehr unterschiedlich von der Helligkeit und dem Kontrast. Am Anfang haben wir Versuche mit der Farbe gelb und später schwarz durchgeführt, jedoch war gelb immer zu schwer von weiß und schwarz zu schwer vom Untergrund zu unterscheiden und deshalb sind diese weggefallen. Bei den anderen Farben haben wir es nach vielen Feinjustierungen geschafft, die unterschiedlichen Farben über verschiedene Distanzen zu erkennen. Durch die Zentrierung der Kamera NAO auf die einzelnen Marker wurde auch das Problem der verschiedenen Farbbereiche im Sichtfeld, wobei nicht nach der Größe der Farbbereiche beurteilt werden darf, gelöst. Die Farberkennung dauerte am Anfang auch sehr lange, aber konnte durch die Wahl des Bildtyps von JPG anstatt PNG stark beschleunigt werden.

III Fazit

9 Fazit	40
9.1 Aufgabenstellung	40
9.2 Projektplanung	40
9.3 Aussicht	41

Kapitel 9

Fazit

9.1 Aufgabenstellung

Die gegebene Aufgabenstellung umfasst viele Teilgebiete des Bachelor-Studiengangs Informatik an der Humboldt-Universität. Es werden sowohl gute Kenntnisse im Bereich Software-Engineering, Kommunikationssysteme und ein gutes Gefühl für Algorithmen benötigt als auch ein grundlegendes Wissen im Bereich mathematischer Problemstellungen, dass über die vermittelten Inhalte des Abiturs hinausgeht. Außerdem bietet sich die Möglichkeit einen Einblick in die Arbeit mit verschiedenen Robotern zu bekommen - wobei sich dieser Aspekt, auf Grund des kleinen Zeitfensters, nur auf sehr rudimentäre Anwendungen beschränkt. Die Problemstellung ermöglicht eine gute und sinnvolle Unterteilung in Kleingruppen, die unter Einhaltung vereinbarter Schnittstellen unabhängig voneinander arbeiten können. Diese Teilaufgaben bieten weiterhin die Möglichkeit, sich für ein gewisses Themengebiet zu entscheiden: Für die NXT's ist eine gewisse Kreativität und handwerkliches Geschick erforderlich, da die Modelle, bzw. das endgültige Modell, viele Anforderungen an seine Umgebung und seine Funktionalität erfüllen muss; Der NAO-Roboter bietet die Möglichkeit der Bilderkennung; die MCC-Gruppe hat hauptsächlich algorithmische Probleme zu lösen. Wie bereits in der Aufgabenstellung zitiert, ist das Problem sehr Anwendungsorientiert - auch wenn es keinen direkten Forschungsbezug gibt, so gewinnt man doch einen sehr guten Eindruck davon, wie zukünftig Roboter bei realen Problemen aushelfen können und vor welchen Schwierigkeiten die Entwickler und Ingenieure stehen.

9.2 Projektplanung

Die größte Hürde der meisten Projekte ist die Projektplanung. Nicht nur ein Vorgehensmodell für die Softwareentwicklung, sowie eine exakte Erfassung von Anforderung sind hier notwendig, sondern auch eine gute Planung auf Ebene der gesamten Gruppe. Das heißt, eine unabhängige Kleingruppenarbeit ist nur möglich wenn die Schnittstellen einvernehmlich und präzise definiert wurden. Ein Integrations- und Systemtest ist nur möglich wenn alle Teilaufgaben bis zu einem vereinbarten Zeitpunkt fertig sind. Ein erfolgreicher Abschluss ist nur möglich wenn von Anfang ein genauer Projektplan ausgearbeitet wird, der es ermöglicht, bei nicht haltbaren Fristen entsprechend zu reagieren. Auch in diesem Projekt hat sich die fehlende Erfahrung deutlich bemerkbar gemacht, insbesondere dadurch, dass letztendlich kein vollständig autonomer Ablauf bei der Abschlusspräsentation möglich war. Es ist offensichtlich, dass nicht nur Wissen in den oben genannten Fachgebieten von Nöten ist, sondern auch eine zumindest grundsätzliche Einführung in Projektmanagement wichtig wäre. Die Projektplanung umfasst häufig auch den Einsatz bestimmter Werkzeuge während des Projekts. Hier bestand das Problem, dass es, abgesehen vom Versionierungstool "GIT", keine einheitliche Verwendung von Tools zur Dokumentation, Verteilung von Aufgaben, Software-Entwicklung und Software-Entwurf gab. Auch die Verwendung der verschiedenen Plattformen (MAC, Windows, Linux) bereitete insbesondere im Bereich der Visualisierung erhebliche Probleme. Zuletzt scheint auch Entscheidung für Python als durchgängig zu verwende Programmiersprache keine gute gewesen zu sein. Allerdings muss an dieser Stelle gesagt werden, dass die damit verbundenen Probleme keineswegs vorhersehbar gewesen sind.

Abschließend lässt sich sagen, dass doch ein sehr erheblicher Teil der Arbeitszeit und des letztendlichen Projektergebnisses der ungenügenden Projektplanung zuzuschreiben sind - es wäre sicherlich sehr zielführend, wenn die Studierenden in zukünftigen Projekten explizit auf diese Herausforderung vorbereitet werden.

9.3 Aussicht

Aus unserer Sicht war das Semesterprojekt trotz der genannten Probleme nicht nur sehr lehrreich, sondern die meiste Zeit auch eine Bereicherung für die Teilnehmer. Insbesondere kann man aus den größeren aufgetretenen Problemen viel lernen, um es in zukünftigen Projekten besser zu machen. Das Themengebiet bietet im allgemeinen unzählige Möglichkeiten an weiteren Projekten teilzunehmen, Abschlussarbeiten zu schreiben oder Forschung zu betreiben. Durch das Semesterprojekt hat man einen weit gefächerten Einblick in die Thematik erhalten. Die Aufgabe hat gezeigt, wie weit der Schritt zu einem tatsächlich autonom agierenden Roboterteam in einem unwegsamen Gelände noch ist.