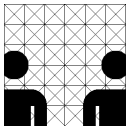
Diplomarbeit
im Studiengang Informatik

Dynamische Navigation und Bewegungssteuerung eines mobilen Robotersystems in Innenräumen

am Arbeitsbereich für
Technische Aspekte Multimodaler Systeme,
Universität Hamburg

vorgelegt von
Benjamin Adler

Mai 2008



betreut von
Prof. Dr. Jianwei Zhang
Dr. Werner Hansmann

Zusammenfassung

Thema dieser Diplomarbeit ist die Neuentwicklung einer Bewegungssteuerung für den Serviceroboter TASER des Arbeitsbereichs TAMS der Universität Hamburg. Die bisher verwendete Software wird im Zuge dieser Arbeit umstrukturiert und um neue Möglichkeiten erweitert werden. Zu diesen zählen im Wesentlichen neue Bewegungsprofile, wie die Fahrt von Freiformkurven mit Hilfe von Splines, eine dynamische Kollisionserkennung und -vermeidung durch rechtzeitige Aktualisierung der geplanten Route sowie eine deutlich verbesserte Integration in das genRob®-System, welches in einem Großteil der am Arbeitsbereich eingesetzten Verfahren Verwendung findet. Durch eine weitgehende Modularisierung der Programmteile wird es möglich, Kernteile der Anwendung auf entfernte Systeme zu migrieren, um dem Hostrechner des Roboters die Bewältigung anderer zeitkritischer Aufgaben zu ermöglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In den Gebieten Robotik und autonome Systeme wird seit Jahren intensiv geforscht. Dies liegt zu großen Teilen darin begründet, dass der erfolgreiche Einsatz selbstständig agierender Maschinen sowohl im Berufs- wie im Privatleben große Vorteile in Bezug auf Wirtschaftlichkeit, Sicherheit und Effizienz mit sich bringt.

Die andauernden Entwicklungen hin zu immer kleinerer und schnellerer Hardware, sowie die fortschreitende Verbesserung der zugehörigen Software ermöglichen schon heute ein Eindringen der Robotik in immer mehr Bereiche unseres Lebens. Dazu zählen naheliegende Beispiele wie Medizin und die Unterhaltungsbranche (GSN00), aber auch weniger bekannte Einsatzgebiete wie die Raumfahrt (HK06) oder das Militär. Wiederkehrende, dem Menschen gefährliche (BS07) oder schlicht langweilende Aufgaben werden mehr und mehr von Maschinen übernommen.

Im Rahmen der Forschung am Arbeitsbereich TAMS des Departments Informatik der Universität Hamburg werden solche Entwicklungen an Robotern erprobt. Die Abkürzung TAMS steht für „Technische Aspekte multimodaler Systeme“. Die eigentliche Herausforderung verbirgt sich hinter dem Wort „multimodal“: stets gilt es, die Daten mehrerer Sensoren auszuwerten, zusammenzuführen und daraus ein schlüssiges Bild zu konstruieren. Erst auf Basis dieses Bildes der eigenen Umwelt können die jeweiligen Systeme überhaupt sinnvoll handeln.

Die am Arbeitsbereich eingesetzten Roboter unterscheiden sich hinsichtlich ihrer Einsatzzwecke und Antriebskonzepte grundlegend. So verfügt der Arbeitsbereich sowohl über humanoide Roboter vom Typ HOAP-2[1], als auch über pneumatische Roboter[2], die mit speziellen Saugnäpfen in der Lage sind Wände hochzuklettern, um beispielsweise Fenster zu reinigen.

Eine weitere dieser Maschinen ist der Serviceroboter TASER, zu sehen in Abbildung 1.1, dessen Bewegungssteuerung Thema dieser praktischen Arbeit ist.

1.1 Motivation

Die Bewegungssteuerung ist eine der grundlegenden Voraussetzungen eines jeden mobilen Robotersystems. Während Arbeitsmaschinen mit feststehender Basis bereits seit vielen Jahren (KC02) existieren und besonders Produktionsprozesse erleichtern, sind Systeme mit der Möglichkeit, die eigene Lage frei im Raum zu verändern noch vergleichsweise selten.

Herausforderungen, die der Einsatz mobiler Systeme mit sich bringt, sind beispielsweise die kabellose Energieversorgung und Kommunikation sowie Gewicht und Größe des Systems - diese Eigenschaften müssen auf den jeweiligen Einsatzzweck abgestimmt sein.

[1] http://jp.fujitsu.com/group/automation/en/services/humanoid-robot/hoap2/, letzter Aufruf 2008-04-16

[2] http://tams-www.informatik.uni-hamburg.de/research/robotics/skycleaner/, letzter Aufruf 2008-04-16

Dadurch, dass das Kriterium der Mobilität in der Praxis meist eine Vorbedingung für autonom agierende Systeme ist, verdient eine verlässliche Steuerung der Antriebssysteme des entsprechenden Roboters besondere Aufmerksamkeit. Gelingt es einem mobilen System nicht, sich an den Ort der vorgesehenen Tätigkeit zu bewegen, büßt es dadurch automatisch Großteile seiner Nützlichkeit ein.

Akzeptanz und wirklichen Nutzen im Alltag kann mobilen Robotersystemen erst dann zugesprochen werden, wenn solch grundlegende Funktionen wie die eigene Positionierung so gut funktionieren, dass sie in den Hintergrund treten.

Somit ist Ziel dieser Arbeit, aktuelle Forschungsergebnisse auf den mobilen Serviceroboter TASER zu übertragen. Im Laufe dieser Arbeit sollen also bekannte Verfahren Anwendung finden und in einigen Bereichen Anpassungen und Erweiterungen erfahren. Die Hoffnung ist, nach Abschluss dieses Projekts durch eine ausgereifte Basis weitere Forschung an diesem Robotersystem zu vereinfachen.

1.2 Der mobile Serviceroboter

Der mobile Serviceroboter *TASER* ist eine Anschaffung des Arbeitsbereichs TAMS vom Dezember 2003. Es handelt sich um eine modifizierte Version der MP-L655[3] Basisplattform aus dem Hause Neobotix GmbH[4].

Der Roboter hat eine Grundfläche von 70*70cm, ist mit allen Aufbauten circa zwei Meter hoch und hat - bedingt durch seine solide Bauweise mit bis zu einem Zentimeter starken Aluminiumplatten und die acht verwendeten 12V-Bleiakkumulatoren mit jeweils 40 Amperestunden - ein Gewicht von ca. 200 Kilogramm.

Direkt über diesen Akkumulatoren ist ein Industrie-PC mit Intel Pentium 4-Prozessor und 1 GB RAM verbaut, dessen Netzteil an die Gegebenheiten der mobilen Energieversorgung angepasst ist. Statt einer normalen Festplatte wurde aufgrund der im Betrieb zu erwartenden Erschütterungen eine Notebookfestplatte mit einer Kapazität von 120 Gigabyte verbaut. Dieser Rechner ist nun für die Verarbeitung aller Sensordaten und die Steuerung sämtlicher Aktuatoren verantwortlich, so dass er ausreichend Rechenleistung für die grundlegenden Anwendungen wie die Arm- und Bewegungssteuerung bieten muss.

Angetrieben wird der Roboter durch zwei wie bei einem Rollstuhl angeordnete Räder, an denen jeweils ein eigener Antriebsmotor montiert ist. Beide werden mit einer Spannung von 48 Volt betrieben und leisten einzeln bis zu 400 Watt. Um ein Kippen des Roboters zu vermeiden, sind hinten eines und vorne zwei passiv mitrollende Stützräder angebracht. Die genauen Winkelstellungen der Motoren (und somit auch jene der Räder) können mit Hilfe zweier Inkrementalgeber[5] ausgelesen werden. Diese Enkoder, die Motorsteuerung sowie jeweils ein Sensor zur Messung der Motortemperatur sind via CAN[6] an den Hostrechner angeschlossen. Über diesen Bus sind auch ein Gyroskop und ein Empfänger für eine Funkfernbedienung angebunden.

[3] http://www.neobotix.de/de/products/Platforms.html, letzter Aufruf 2008-03-21

[4] http://www.neobotix.de/, letzter Aufruf 2008-03-21

[5] Inkrementalgeber, auch Enkoder genannt, sind an Motoren montierte Sensoren, die das Auslesen von absoluten oder relativen Rotationspositionen ermöglichen.

[6] CAN-Bus ist ein serielles und asynchrones Bussystem aus der Automobilindustrie. Als besondere Eigenschaften gelten seine Einfachheit und Ausfallsicherheit.

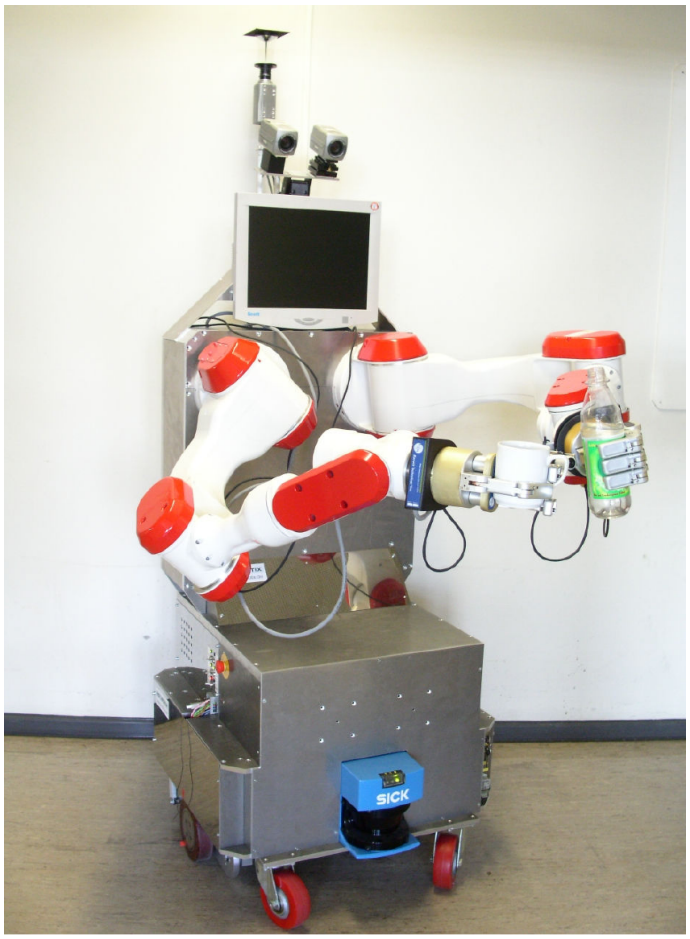


Abbildung 1.1: Der Serviceroboter TASER

Aus Sicherheitsgründen befindet sich in beiden Motoren je ein elektromagnetisches Bremssystem, das vor jeder Fahrt explizit gelöst werden muss und in diesem deaktivierten Zustand ungefähr ein Ampere Strom verbraucht.

Auf Schulterhöhe sind links und rechts zwei PA10-6C[7]-Manipulatoren der Mitsubishi Heavy Industries Ltd. verbaut. Sie sind über ARCNet[8] mit dem Hostrechner verbunden. An den Armenden ist jeweils eine BarrettHand BH-262 montiert, die zum Greifen und zur Manipulation unterschiedlichster Objekte verwendet werden können. Jede dieser Hände hat drei Finger und ist über eine serielle Schnittstelle (RS-232) steuerbar.

An der Vorder- und Rückseite des Roboters befindet sich jeweils ein SICK LMS200 Lasermesssystem. Diese sind allerdings nicht direkt mit ihrer RS-422 Schnittstelle an den Hostrechner angeschlossen, weil dies in der Vergangenheit zu hoher CPU-Last und häufigen Fehlern in der Da-

[7] http://www.mhi.co.jp/kobe/mhikobe/products/mechatronic/index.html, letzter Aufruf 2008-02-24

[8] ARCNet ist eine Vernetzungstechnologie ähnlich Ethernet. Es wurde 1976 erfunden und weitgehend durch Ethernet ersetzt, findet aber aufgrund seiner Vorteile in Echtzeitanwendungen besonders in industriellen Bereichen weiterhin Verwendung.

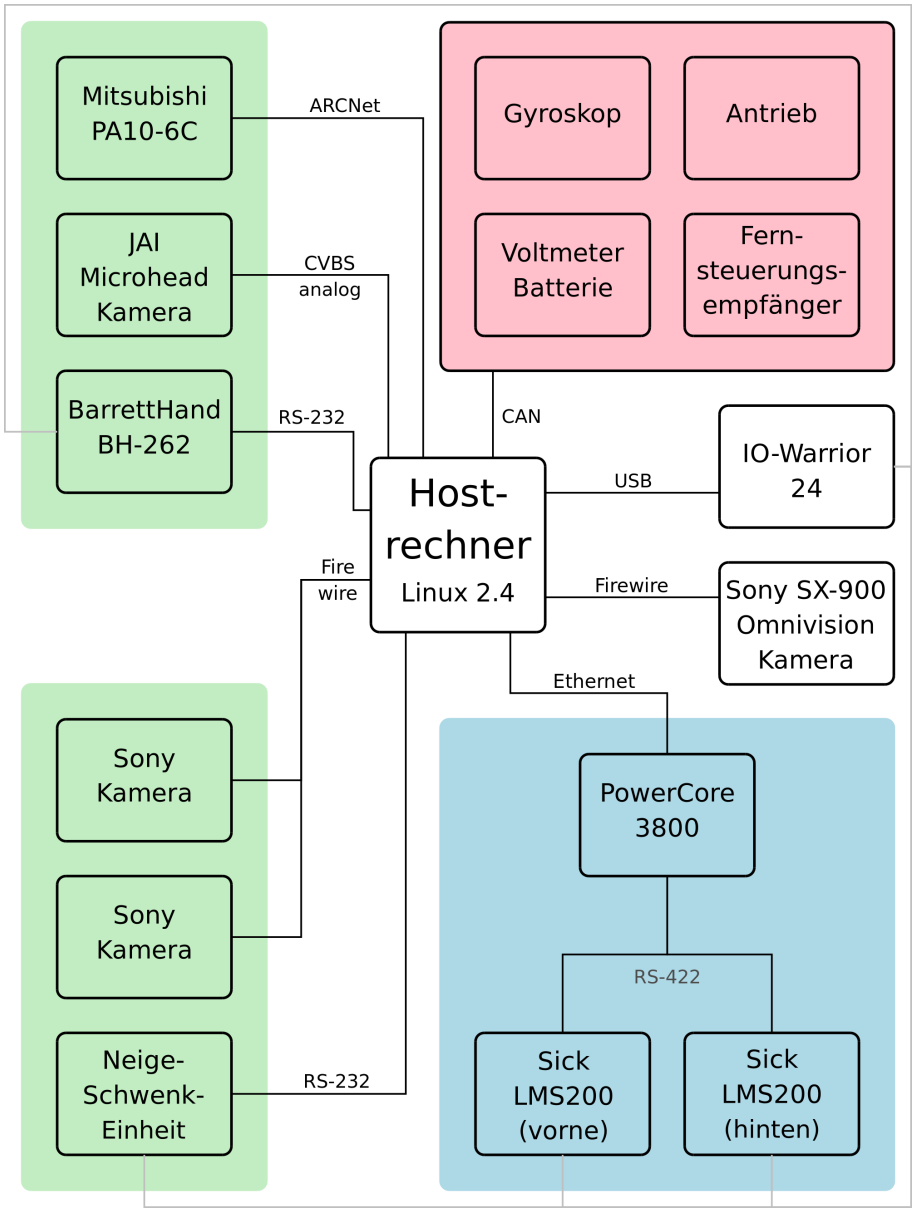


Abbildung 1.2: Hardwareausstattung des TASER

tenübertragung geführt hatte. Mit Hilfe eines von Hannes Bistry und Stefan Pöhlsen im Rahmen einer Diplomarbeit (BPWZ07) entwickelten Systems werden die Daten von einem eingebetteten System zuerst vorverarbeitet und dann über UDP-multicast-Pakete an den Hostrechner versandt. So wird es möglich, fast die gesamte Umgebung des Roboters in der waagerechten Ebene auf Höhe der beiden Scanner im Umkreis von ungefähr 8 Metern zu erfassen.

Insgesamt verfügt der Roboter über fünf Kameras:

- Eine Sony DFW-SX900 ist über Firewire an den Rechner angebunden. Sie ist am höchsten Punkt des Roboters mit der Blickrichtung nach oben angebracht und liefert über einen Spiegel in der Form eines halben zweischaligen Hyperboloids[9] Panorama-Bilder der aktuellen Umgebung.
- Zwei Sony Firewire-Kameras sind an einer Neige-/Schwenkeinheit PTU-D46 von Directed-Perception[10] fixiert. So ist es möglich, Stereo-Bilder eines beliebigen Ausschnitts der Umwelt zu erfassen, ohne den Roboter selbst dabei zu bewegen. In naher Zukunft sollen diese beiden Kameras durch zwei mit Gigabit-Ethernet verbundene Basler eXite-Kameras ersetzt werden. Da diese Kameras mit einer eingebetteten Recheneinheit versehen sind, wird ein Großteil der Bildverarbeitung vom Hauptrechner in die Kameras ausgelagert werden können.
- An den beiden Händen des Roboters befindet sich jeweils eine JAI Microhead-Kamera. Ihr analoger Ausgang ist mit einer Framegrabber-Karte des Hostrechners verbunden. Durch die Art der Montage können diese Kameras nicht nur in ihrer Orientierung, sondern auch in der Position manipuliert werden und bieten so besonders vielfältige Einsatzmöglichkeiten z.B. zur Unterstützung beim Greifen von Objekten verschiedenster Art.

1.3 Ziel der Arbeit

Das vor Beginn dieser Arbeit existierende Programm *mobiled* (abgeleitet von *mobile daemon*) zur Fahrtsteuerung des *TASER* kann am Roboter nur sehr grundlegende Bewegungen zur Ausführung bringen, die sich aus der Drehung auf der eigenen Achse (Rotation) und der Geradeausbewegung (Translation) zusammensetzen.

Obwohl dies eigentlich ausreichend ist, um jede erreichbare Pose[11] anzufahren, führt dies in der Praxis zu umständlichen, sowohl in Zeit wie Energieaufwand ineffizienten Bewegungsabläufen. Wenn man bedenkt, dass der Roboter bei jeder einzelnen Teilbewegung anfangs beschleunigen und letztlich bremsen muss, bevor er zur nächsten Teilbewegung übergehen kann, wird es unschwer möglich, sich die Langwierigkeit einer solchen Fahrt vor Augen zu führen.

Das Programm *mobiled* ist ein Resultat zahlreicher Forschungsarbeiten (WS02), (Sch04) in ähnlichen Themenbereichen, die hauptsächlich von Torsten Scherer, Hagen Stanek und Daniel Westhoff

[9] Panorama-Eye ist ein Produkt der Firma ACCOWLE Vision, http://www.accowle.com/english/index.html, letzter Aufruf 2008-04-12

[10] DirectedPerception, http://www.dperception.com/, letzter Aufruf 2008-04-12

[11] Eine Pose ist ein Tupel, durch das die Lage eines Objektes im Raum eindeutig bestimmt wird. Im Falle eines sich auf einer Ebene bewegenden Systems besteht eine Pose aus folgenden drei Parametern: jeweils einer X- und Y-Koordinate sowie einem Winkel, der die Orientierung des Systems angibt.

angefertigt wurden. In jeder dieser Arbeiten blieb *mobiled* jedoch stets nur Mittel zum Zweck. Dies führte dazu, dass sich der Quelltext des Programms zu einem lediglich sporadisch dokumentierten Code entwickelt hat, was wiederum Wartungsaufgaben und Weiterentwicklungen erschwerte. Noch schwerer wiegt die Tatsache, dass die Architektur der Software im Laufe dieser Arbeiten sichtlich gewachsen war: in vielen Bereichen des Quelltextes finden sich Codefragmente, die „ins Nichts“ führen, weil einige der damals gewünschten Funktionen wieder entfernt bzw. letztlich anders implementiert wurden.

Während der Entstehung und Entwicklung des *mobiled* entstand auch ein weiteres Softwaresystem namens genRob[12]. genRob ist ein verteiltes, Java-basiertes System, mit dessen Hilfe verschiedene Anwendungen durch Verwendung sogenannter *Roblets*[13] die Vorteile einer verteilten Rechnerumgebung nutzen können. Mit Hilfe dieses Rahmenwerks wurden robotikrelevante Anwendungen wie eine Kartenverwaltung (siehe Kapitel 3.2.6.2) oder eine Routenplanung (Kapitel 3.2.6.3) realisisiert.

Die meisten der auf dem Roboter aufsetzenden Anwendungen, wie beispielsweise die Verwendung der Arme, erfordern die genaue und aktuelle Pose des Roboters als Vorwissen. Diese Information ist aber nur durch Integration der Anwendung in das *genRob*-System abrufbar. Dies kann dann problematisch sein, wenn die Anwendung nicht in Java programmiert wurde und deswegen nur schwerlich mit dem *genRob*-Rahmenwerk kommunizieren kann.

Ziele dieser Arbeit sind deswegen die folgenden sechs Punkte:

- die Dokumentation des Programms, seiner Architektur und seines Quelltextes
- die Erweiterung der Fahrtsteuerung um die Möglichkeit, Freiformkurven abzufahren
- eine zuverlässige Kollisionserkennung und -vermeidung, bzw. deutliche Hinweise, wenn diese deaktiviert sind.
- eine beiderseitige Integration des *mobiled* ins *genRob*-System. Dies ist für eine reibungslose Fahrtplanung und -steuerung notwendig.
- eine Umstrukturierung der Programmarchitektur mit dem Ziel, klare und durch ihre Funktion voneinander abgetrennte Module zu erhalten.
- eine einfache Erweiterbarkeit des Bewegungsrepertoires, ermöglicht durch ein durchdachtes Interface zwischen *mobiled* und dem bewegungssteuernden Code.

1.4 Aufbau der Arbeit

Dieses Dokument umfasst insgesamt sechs Kapitel, deren Reihenfolge dem chronologischen Verlauf der Arbeit entspricht. Im nun folgenden zweiten Kapitel wird auf den aktuellen Stand der

[12] http://www.genrob.com/, letzter Aufruf 2008-05-02

[13] Ein Roblet ist ein Programm, welches z.B. zur Erhöhung von Leistung und Ausfallsicherheit zwischen verschiedenen Rechnern verschickt werden kann.

Forschung zu diesem Thema eingegangen. Hierbei werden die verschiedenen Lösungsansätze vorgestellt und gegeneinander abgegrenzt. Schließlich soll gezeigt werden, inwiefern die einzelnen Ansätze geeignet sind, den gestellten Aufgaben gerecht zu werden.

Der folgende dritte Abschnitt beschreibt die Architektur sowohl des Programms *mobiled* als auch die der es umgebenden Software. Es wird auf konkret verwendete Technologien und Einzelheiten eingegangen. Für Leser, deren Anliegen es ist, sich möglichst rasch ein Bild von der Robotersteuerung zu machen, ist dieser Abschnitt am ehesten zu empfehlen.

Kapitel vier dient der detaillierten Beschreibung der verwendeten Fahrtplanungs- und -steuerungsalgorithmen. Da die Beschäftigung mit den teilweise recht theoretischen Hintergründen der Regelungstechnik für eine natürliche und robuste Bewegungssteuerung unerlässlich ist, ist diesem Thema eine eigenes Kapitel gewidmet.

Im folgenden Kapitel fünf wird das entstandene Fahrverhalten des Roboters im Einzelnen getestet werden. Es werden Daten gesammelt und ausgewertet, um die Leistung des Steuerungsprogramms anschließend objektiv bewerten zu können.

Der letzte Abschnitt dieser Arbeit soll eine Zusammenfassung ihrer selbst sein und zugleich einen Ausblick auf die zahlreichen weiteren Möglichkeiten und Entwicklungsrichtungen bieten. Es wird deutlich, dass die Themen der Fahrtsteuerung und ihrer Voraussetzungen (z.B. die Selbstlokalisierung) keineswegs erschöpfend behandelt wurden.

2 Stand der Technik

Dieser zweite Abschnitt soll einen Überblick über die bisherigen Herangehensweisen an das Problem der Bewegungssteuerung mobiler Systeme geben - dieses ist verständlicherweise so alt wie die mobile Robotik selbst. Einen guten Überblick über das Feld gibt (MS07). Schon der „Vater“ eines der ersten mobilen Robotersysteme „Shakey“ (Nil84) erkannte, dass die Planung und Ausführung von dem aktuellen Umfeld angepassten Bewegungen eine Grundlage für die Nützlichkeit dieser Maschinen darstellt.

Dieses Forschungsfeld hat sich aufgrund der vielfältigen Anwendungsszenarien in den letzten 35 Jahren mit verschiedensten mobilen Systemen beschäftigt. Abhängig von gewünschtem Einsatzgebiet und verfügbaren Budget haben sich Forscher der Entwicklung autonomer Steuerungen für normale PKWs (Ror05), schwere Panzer (KPH88) und andere militärische Fahrzeuge (CMLL00), fussballspielende Miniroboter (Br9), ein-(Mar07) und dreirädrige (Hor94) mobile Plattformen oder gar in beliebige Richtungen steuerbare Systeme[1] ((KNDG02) und (PN95)) angenommen.

Dabei blieb die Problemstellung keineswegs auf einzelne Roboter beschränkt: mehrere Paper beschäftigen sich mit der Herausforderung, die Bewegung gleich mehrerer Roboter miteinander zu koordinieren (TBB03) - die sonst hilfreiche Annahme, dass der kürzeste Weg auch der schnellste ist, fällt spätestens hier der Natur solcher Szenarien zum Opfer. Weiterhin wurden Lösungswege für völlig gegensätzliche Umgebungen entworfen: Roboter bewegen sich heute schon in bergigem Terrain (BS02), in Wüsten (RHSH05), Wäldern und Minen (Kan04) oder auf dem Mars (HK07) - wie auch diese Arbeit befasst sich die große Mehrzahl an Beiträgen allerdings mit einer vergleichsweise einfachen Umgebung: Innenräumen.

Damit ist die Diversität des Forschungsfeldes aber noch immer nicht abgedeckt, denn auch die Art und Weise der Übermittlung von Fahraufträgen könnte kaum unterschiedlicher sein: Einige Systeme nehmen Bewegungsanforderungen per Touchscreen entgegen (HAK03), andere nehmen Posen ein, die ihnen durch das Setzen eines Punktes mit einem Laserpointer demonstriert werden (PA01). Es wurden auch Systeme entworfen, die eigentlich direkter menschlicher Steuerung über das Internet unterliegen und nur im Falle von Netzwerkproblemen autonom agieren (T1).

Um den Rahmen dieser Arbeit nicht zu sprengen, wurde versucht, die zahlreichen Ansätze anhand einiger Kernpunkte in verschiedene Klassen einzuordnen. Denn obwohl es beliebig viele verschiedene Möglichkeiten gibt, die Steuerung eines mobilen Systems zu implementieren, haben sich einige wenige Herangehensweisen etabliert. Eine gängige Dichotomie dieser Lösungswege orientiert sich an der Frage, ob die verwendeten Systeme vorbedacht planend oder - im Gegensatz dazu - reaktiv auf Sensordaten reagierend arbeiten.

Erstere Herangehensweisen finden häufig dort Verwendung, wo verlässliche und planbare Aktionen gut geeignet sind, vergleichsweise fest definierte Aufgaben zu erledigen. Wenn eine Maschine zeitlich aufeinander abfolgend zuerst mit Hilfe ihrer Sensoren die eigene Umwelt erfasst, darauf

[1] mobile Roboter mit sog. omnidirektionalem Antrieb

aufbauend ein Modell dieser erstellt, anschließend eigene Planungen in dieser Umwelt anstellt, um diese letztlich durch Steuerungssignale an den Aktuatoren zur Ausführung zu bringen, spricht man von einer sog. *sense, plan, act*-Architektur.

Abbildung 2.1: Die Sense-Plan-Act-Architektur

Die sequentielle und monolithische Herangehensweise der *SPA*-Architektur führt meist zu recht langsamen Anwendungen mit vergleichsweise hoher Reaktionszeit (Sim90) auf sich ändernde Bedingungen, da in jedem Iterationsschritt Fusion von Sensorwerten, Weltmodellierung und Planung ausgeführt werden müssen, bevor die Maschine reagieren kann. Änderungen in der Umwelt des mobilen Roboters können die Notwendigkeit ständiger Neuplanungen nach sich ziehen, was gerade bei komplexeren Systemen und entsprechend aufwändigen Planungsvorgängen häufig zu unrealistischen Anforderungen bezüglich der Leistungsreserven des Hostcomputers führt.

Dem gegenüber stehen rein reaktive Systeme, die aus den verfügbaren Sensordaten innerhalb kurzer Latenzzeiten die Ansteuerung der Aktoren und somit das Verhalten des Roboters bestimmen. Die Eigenschaft, verfügbare Sensordaten in hoher zeitlicher Dichte abzufragen und in Steuerungskommandos zu wandeln, hat ihnen in der englischen Literatur den Klassennamen „sampling-based motion planning“ eingebracht (LL04). Aufgrund ihrer Architektur sind sie eher geeignet, echtzeitfähige Robotersteuerungen zu implementieren. Reaktive Systeme zeichnen sich dadurch aus, dass sie keinerlei interne Welt-Modelle verwalten und somit auch nicht nach optimalen Pfaden zwischen Start- und Zielkonfiguration suchen. Vielmehr halten sie eine Zuordnungsmenge zwischen Bedingung und Verhalten bereit (Mat97, Ch.2). Dies bedeutet aber, dass sie nicht in der Lage sind, komplexere Aufgaben bzw. längerfristige Planungen zu bewältigen.

Die Mischung beider Lösungsansätze führt zu einer hybriden Lösung, in der beide Strategien ihren Platz gefunden haben: Die planende Komponente bedient sich des internen Welt-Modells, um sich um die langfristigen Ziele des Roboters (z.B. Routenplanung) zu kümmern, während sich der reaktive Teil des Systems mit aktuellen Problemen wie der Kollisionsvermeidung befasst. Die Herausforderung hierbei ist nun, beide Systeme sinnvoll miteinander zu koppeln.

2.1 Neuronale Netze

Die Konstruktion mathematischer Modelle, die eine korrekte Zuordnung zwischen Sensorwerten und Werten zur Ansteuerung der Aktuatoren modellieren, ist häufig schwierig, in manchen Fällen gar unmöglich.

Aus dieser unglücklichen Tatsache heraus entstand nun die Notwendigkeit, andere Möglichkeiten der Steuerung künstlicher, nichtlinearer Systeme zu finden. Eine dieser Forschungsrichtungen ist die der künstlichen neuronalen Netze, welche von Donald O. Hebb mit seiner Arbeit (Heb49, The Organization of Behavior) - aus der auch die Hebb'sche Lernregel stammt - begründet wurde. Aufgrund dieser Veröffentlichung gilt Hebb bis heute als Entdecker der synaptischen Plastizität,

welche die Basis für sowohl Lernen als auch Erinnerung und damit die Voraussetzung für die in den nächsten Abschnitten behandelten Verfahren bildet.

Für alle der auf KNNs[2] basierenden Ansätze gilt, dass sie in der Lage sind, das jeweils gesuchte Modell aus Beispielen im Rahmen eines Trainings zu erlernen. Dadurch entfällt die Voraussetzung, zur Konstruktion einer Steuerung zuerst aus komplexem Verhalten konkrete Regeln zu abstrahieren. Diese Herangehensweise ist in vielen Fällen (Oub06, Ch.8) bereits erfolgreich eingesetzt worden.

In den nun folgenden Unterkapiteln sollen die grundlegenden, an künstliche neuronale Netze gebundenen Eigenschaften und Verfahren vorgestellt werden.

2.1.1 Lernverfahren

Wie eben bereits erwähnt, müssen neuronale Netze vor ihrem Einsatz trainiert werden. Während des Trainings werden über die Eingangsneuronen des Netzes bestimmte Reize präsentiert, die zu Erregungen der Ausgangsneuronen führen. Durch den Vergleich des jeweiligen Ausgangssignals mit dem gewünschten Ausgangssignal werden Informationen gewonnen, die nach jedem Schritt zu einer Veränderung des Netzes genutzt werden. Je nach Typ des neuronalen Netzes bedeutet die Veränderung des Netzes das Festlegen von Parametern wie die Erregungsschwellen einzelner Neuronen oder die Gewichtung neuronaler Verknüpfungen.

Hauptsächlich existieren drei unterschiedliche Methoden, ein künstliches neuronales Netz zu trainieren und jede ist für die Lösung eines jeweils konkreten Problems unterschiedlich gut geeignet. Diese Methoden sollen im Folgenden vorgestellt werden.

2.1.1.1 Überwachtes Lernen

Das künstliche neuronale Netz wird anhand von Beispielen mit vorgegebener Eingabe und den korrekten zugehörigen Ausgabewerten trainiert. Im vorliegenden Fall würde also ein Lehrer den Roboter fahren, um ihn zu trainieren.

Das Verfahren ist im Grunde simpel und einfach zu implementieren, bringt aber auch gravierende Nachteile mit sich. Nach (OS97, Ch.9) gelingt es so trainierten Netzen häufig nicht, das Verhalten ihres Lehrers in ausreichender Annäherung abzubilden. Dies droht insbesondere dann, wenn der Lehrer zur Steuerung des Roboters andere Daten als die Eingangswerte des neuronalen Netzes zur Steuerung verwendet. Als weiteres Beispiel führen die Autoren Ben Kröse und Joris van Dam auf, dass das Umfahren von Hindernissen entlang unterschiedlicher Seiten zu Mehrdeutigkeiten in der Lernmenge des Netzes führt.

Da besonders in Backpropagation-Netzen auch die chronologische Reihenfolge der zugeführten Trainingsbeispiele von Bedeutung ist, wird somit nicht nur die Auswahl der Lernbeispiele, sondern auch deren Abfolge zu einem Problem. Hohen Aufwand erfordert die Auswahl von Lehrbeispielen auch deswegen, weil unbedingt vermieden werden muss, dass das neuronale Netz Muster in den Eingabedaten zu erkennen erlernt, die zwar in den Beispielen mit den gewünschten Ausgabedaten

[2] künstliche neuronale Netze

in Korrelation stehen, nicht aber im späteren Einsatz. Beispielsweise könnte ein KNN sich ausschließlich auf die Helligkeit verwendeter Trainingsbilder konzentrieren, wenn dies innerhalb der Trainingsmenge zu adäquatem Verhalten führt.

Die Methode des überwachten Lernens hat weiterhin den Nachteil, dass sich ein einmal eingelerntes Netz im Betrieb nicht weiter anpassen kann. Ändert sich die Umwelt des Systems, muss es neu trainiert werden, was eine erneute Auswahl von Lernbeispielen notwendig macht.

Leider muss das KNN auch dann angepasst werden, wenn sich intrinsische Eigenschaften des mobilen Systems verändern. Eine Gewichtsveränderung des Roboters durch Aufnahme einer Transportlast oder das langsame Absinken der Batteriespannung könnten durch ein mit dieser Methode trainiertes Netz nicht kompensiert werden.

2.1.1.2 Unüberwachtes Lernen

Um ein KNN durch unüberwachtes Lernen zu trainieren ist es erforderlich, stets einen Referenzzustand des Systems zum Vergleich mit dem aktuellen Zustand zur Verfügung zu haben. Dieser Zustandsvergleich wird zur Ermittlung des Fehlers im aktuellen Verhalten des Netzes genutzt und führt so - je nach Größe des Fehlers - zu entsprechend großen Korrekturen des Netzes selbst.

Übertragen auf das Thema dieser Arbeit wäre der Referenzzustand die zum aktuellen Zeitpunkt gewünschte Pose des mobilen Roboters, während der aktuelle Zustand die mit Hilfe der Lokalisierung ermittelte Pose ist. Somit besteht der Fehler aus der Distanz und Orientierungsdifferenz dieser beiden Posen.

Problematisch bei diesem Ansatz ist, dass der Referenzzustand des Systems im Vorwege bekannt sein muss; dies ist aber im Falle eines Fahrauftrages von einer Start- zu einer Zielkonfiguration nicht der Fall. Die Ermittlung der Referenzzustände wird erst dadurch möglich, dass ein vorgegebener Pfad zeitlich abgeschritten und so die momentan gewünschte Pose aus ihm abgeleitet werden kann.

Letztlich würde dies bedeuten, dass der Ansatz des unüberwachten Lernens hier eher zur Korrektur der Steuerungswerte beider Antriebsmotoren als zur vollständigen Bewegungssteuerung des Roboters genutzt werden könnte. Hinsichtlich des nun relativ beschränkten Einsatzgebietes ist es fraglich, ob eine Implementierung auf Basis dieses recht aufwendigen Verfahrens lohnend ist.

2.1.1.3 Bestärkendes Lernen

Wird nun gewünscht, dass das mobile System auch den Fahrtweg zwischen Start- und Zielkonfiguration selbst wählt, ist die Berechnung von Referenzkonfigurationen nicht mehr möglich - gerade dies soll das System ja selber leisten.

Bestärkendes Lernen findet nach (OS97, Ch.9.2.3) meist dort Verwendung, wo zum Training verwendbare Reize sowohl weniger informativer (häufig sogar eindimensionaler) Natur als auch zeitlich stark verzögert sind. Gäbe man dem mobilen System einen Fahrauftrag, ohne dabei eine Route vorzugeben, ist eine Bewertung des Verhaltens des Netzes nicht zu jedem Zeitpunkt möglich. Rückmeldung bekäme das System somit erst, wenn es entweder zu einem klaren Fehler (wie

beispielweise einer Kollision oder das Fahren in eine Sackgasse) käme, oder wenn die Zielkonfiguration erreicht würde.

Häufig ist es allerdings auch aufgrund fehlender Sensordaten nicht möglich, nötige Rückmeldungen zu geben. Somit werden Szenarien denkbar, in denen sich das System durch entweder verzögerte oder schlicht nicht vorhandene Lerndaten selbst in Gefahr bringt: Der Roboter könnte in Betracht ziehen, sein Ziel durch das Herunterfahren einer Treppe zu erreichen.

2.1.2 Verfahrensübergreifende Probleme

Künstliche neuronale Netze tendieren im Allgemeinen dazu, die während des Trainings gelernten Beispiele einfach auswendig zu lernen. Dieses „Überanpassung“ genannte Phänomen verhindert im späteren Betrieb die Möglichkeit, sich auf neue Situationen einzustellen bzw. neuen Daten zu abstrahieren, um passende Steuerungsbefehle zu generieren.

Für mobile Robotersysteme, die auf Grund ihres Gewichts oder ihrer Bewegungsgeschwindigkeit für ihre Umwelt gefährlich werden können, ist es besonders wichtig, verlässliche und von Menschen antizipierbare Bewegungsabläufe zu generieren. Diese Eigenschaft kann bei Verwendung künstlicher neuronaler Netze nicht garantiert werden: Eine Vorhersage des Verhaltens eines Roboters ist schon bei in ihrer Struktur verhältnismäßig einfachen Feed-Forward-Netzen aufwändig. Kommen nun noch komplizierte rekurrente[3] neuronale Netze zum Einsatz, wird es nahezu unmöglich, das Verhalten des Roboters im Vorwege einzuschätzen. Dazu trägt auch die Tatsache bei, dass sich das Netz - je nach verwendetem Lernverfahren - aufgrund seiner synaptischen Plastizität noch während des Betriebs verändern kann.

Obwohl neuronale Netze sehr flexibel eingesetzt werden können, um Verhalten zur Bewältigung verschiedenster Aufgabenstellungen zu erlernen und später anzuwenden, müssen die vorher verwendeten Trainingsdaten sehr genau auf das Problem zugeschnitten sein. Die Kodierung dieser Daten und die Art ihrer Eingabe in das Netz haben einen großen Einfluss darauf, wie gut es das jeweilige Problem erlernen kann. So ist es wenig erfolgversprechend, ein KNN auf die Erkennung von Türen in Kamerabildern zu trainieren, indem man deren Pixeldaten direkt in die Netzeingänge führt. Die gewonnenen Daten z.B. durch eine Kantenerkennung vorzubearbeiten und somit die Problemstellung zu präzisieren und dem Netz Abstraktionsarbeit abzunehmen würde das Erlernen der Aufgabe sehr erleichtern bzw. überhaupt erst ermöglichen.

2.2 Verhaltensbasierte Steuerung

Zu den ältesten in der Praxis eingesetzten Verfahren zur Steuerung mobiler Roboter zählt der verhaltensbasierte Ansatz. Nach Vorstellung einer wegweisenden Arbeit von Rodney Brooks (Bro85) im Jahre 1985 gestaltete sich dieses Thema für einige Jahre zu einem Forschungsschwerpunkt in der Robotik (TBF06, Ch.1.6). Dieses Kapitel soll die hinter diesem Ansatz stehende Idee verdeutlichen und ihre Vor- und Nachteile erläutern.

[3] Rekurrente KNNs zeichnen sich durch geschlossene Wege innerhalb ihres Konnektivitätsgraphen, d.h. durch Verbindungen von Neuronen zu anderen Neuronen der selben oder einer vorausgegangenen Schicht aus.

Verhaltensbasiert bedeutet, dass auf dem Roboter gleichzeitig mehrere Verhalten „ablaufen“, deren Aktionen dann gemeinsam auf die Aktoren des Roboters angewendet werden. Man spricht bei einer solchen Dekomposition von komplexem Verhalten in einzelne und einfache Verhaltensmodule von der sog. Subsumptions-Architektur (Bro87), dargestellt in Abbildung 2.2.

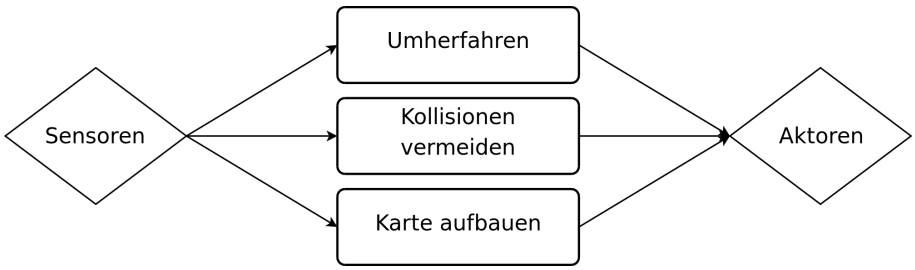


Abbildung 2.2: Ein Beispiel einer Subsumptions-Architektur - mehrere Verhaltensprogramme laufen gleichzeitig ab und konkurrieren um die Steuerung des Systems

Jedes Verhalten hat das Erreichen oder Erhalten eines Zustands zum Ziel; so könnten beispielsweise die folgenden Programme gleichzeitig auf einem autonom agierenden, mobilen Staubsauger ablaufen:

- Herumfahren und Staub saugen
- Kollisionen vermeiden
- Bei niedriger Akkuspannung an die Ladestation andocken
- Treppenhäuser meiden
- Menschen meiden um nicht zu stören
- die Wohnung nicht verlassen
- Bei gefüllter Staubkammer nach draußen fahren und diese entleeren

All diese Verhaltensmuster sind für sich genommen eigene reaktive Systeme und - abhängig von den zur Verfügung stehenden Sensordaten - relativ einfach zu implementieren. Auf den ersten Blick ergibt sich so ein für den Einsatzzweck schlüssiges Gesamtverhalten. Das ist der Grund, warum mobile Systeme mit überschaubaren Aufgabenstellungen häufig auf diese Weise realisiert sind[4].

Auf der anderen Seite lässt sich beobachten, dass verhaltensbasierte Systeme häufig zu Fehlern neigen, Aktionen wiederholen und im Beobachter einen Eindruck der Verwirrung hinterlassen. Ursache dieser Probleme ist die Tatsache, dass die einzelnen Verhalten sich in verschiedenen Stufen ihrer Ausführung widersprechen können. In solchen Fällen muss entschieden werden, welches Verhalten in diesem Moment wichtiger ist und somit zur Ausführung kommen soll.

Der Staubsauger wird bei gefüllter Staubkammer auf dem Weg nach draußen also einem Konflikt begegnen, da er die Wohnung nicht verlassen darf. Denkbar wäre auch ein Konflikt, wenn er bei

[4] der iRobot® Roomba® ist ein Beispiel eines solchen verhaltensbasierten Staubsaugers

niedriger Akkuspannung zur Ladestation fährt, die sich allerdings in einem Treppenhaus befindet. Die Lösung dieser Probleme scheint eine Bewertung der Wichtigkeit dieser einzelnen Handlungsmodule zu sein. Die Erfahrungen zeigen aber, dass im praktischen Einsatz so viele Ausnahmen und Priorisierungen benötigt werden, dass sich schnell ein hoch komplexes System entwickelt.

Im Falle der Subsumptionsarchitektur wissen die einzelnen Verhalten nichts von einander. Selbst mit korrekter Priorisierung könnte also ein Verhalten zur Kollisionsvermeidung den Roboter aufgrund eines Hindernisses immer wieder in eine Richtung steuern, die der Richtung des vorher und nachher aktiven Ziel-anfahren-Verhaltens entgegengesetzt ist.

Problematisch gestaltet sich der Entwurf der einzelnen Verhaltensmodule und ihrer Prioritäten auch deshalb, weil es schon während der Entwicklung schwierig wird, die Ursache für ein fehlerhaftes Gesamtverhalten zu ermitteln.

Zu noch größeren Hürden führt die Notwendigkeit, in einer höheren Ebene der Kontrollsoftware analysieren zu können, ob die momentan agierenden Einzelverhalten tatsächlich zu zielführenden Aktionen führen. Wäre dem nicht so, müssten diese ja geändert oder in ihrer Gewichtung angepasst werden. Ein Verfahren, das ausgereift und praxisnah genug wäre, um es tatsächlich anwenden zu können, hat der Autor nicht finden können.

2.3 Fuzzy-Logik

Fuzzy-Logik wurde in den späten 60ern und frühen 70ern entwickelt, angestoßen durch einen Beitrag von Lotfi A. Zadeh (Zad65) über „Fuzzy Sets“. Die grundlegende Idee ist, anstelle von „scharfen“ Zahlen mit Werten zu arbeiten, die ein gewisses Maß an Unsicherheit bzw. Unschärfe enthalten. Dieses Maß wird durch einen Wahrheits-, Wahrscheinlichkeits- oder Zutrefflichkeitswert zwischen 0 und 1 beschrieben. Somit entspricht die Fuzzy-Logik eher einem menschlichen Denken, welches zwischen „wahr“ und „falsch“ noch weitere Abstufungen zulässt. Durch Kombination dieser Fuzzy-Werte in neue Fuzzy-Werte durch Fuzzy-Regeln und abschließende Rückwandlung in scharfe Zahlen lassen sich Probleme lösen, die mit der booleschen Logik allein nicht handhabbar wären.

Die der Klasse Fuzzy-Logik zugeordneten Ansätze gehören meist zur Gruppe der reaktiven Systeme. Sie sind daher gut geeignet, um schnelle Reaktionen auf äußere Umstände zu erhalten. Dies führt automatisch zu großen Vorteilen wenn es darum geht, Maschinen in dynamischen bzw. schlecht antizipierbaren Umgebungen zu betreiben. Daher scheint ein Fuzzy-Ansatz für die Zwecke dieser Arbeit auf den ersten Blick durchaus geeignet zu sein.

Fuzzy-Systeme verhalten sich laut (SRK97) generell sehr robust gegenüber dem unausweichlichen Rauschen der Sensordaten, großer Umgebungsvariabilität und Unsicherheit in den verwendeten Fuzzy-Parametern. Ferner sind effiziente Implementierungen in Soft- und Hardware vergleichsweise einfach zu erreichen. Dies ist in der mobilen Robotik ein großer Vorteil, weil eingebettete Systeme mit vergleichsweise geringer Rechenleistung hier häufig Verwendung finden.

Letztlich führt eine auf Fuzzy-Logik basierende Herangehensweise in den meisten Fällen zu einer Umsetzung innerhalb eines verhaltensbasierten Systems, so zum Beispiel auch in (TA01). In diesem verhaltensbasierten Konzept finden Fuzzy-Systeme ihren Platz überwiegend dort, wo es darum geht, die Ausgaben der einzelnen gleichzeitig agierenden Verhalten zu optimieren.

Möglich ist aber auch, das wohl größte Problem verhaltensbasierter Systeme mit Fuzzy-Logik zu lindern: (YP95) hatten 1995 die Idee, die Koordination mehrerer gleichzeitig ablaufender Handlungsstränge und ihrer teils widersprüchlichen Ausgaben ebenso mit Hilfe von Fuzzy-Regeln zu einer klaren Ansteuerung der Aktoren zu fusionieren. Eine Umsetzung dieser Methode zeigen (LTF97) mit Hilfe eines sehr einfaches Satzes von Fuzzy-Regeln an einem simulierten Roboter.

In einer anderen Arbeit (SRK97) bieten die Autoren eine konkrete Umsetzung auf einem realen Roboter, der mit lediglich vier unabhängigen Verhaltensregeln durch Innenräume navigieren kann. Das Konkurrenzproblem der einzelnen Programme lösen sie durch kontextabhängige Aktivierung der einzelnen Verhalten. So wird z.B. das „fahre durch Tür 5“-Verhalten nur aktiviert, wenn aus dem allen Einzelverhalten zur Verfügung stehendem Gesamtkontext deutlich wird, dass der Roboter sich in der Nähe eben jener Tür befindet.

Allerdings wird hier auch deutlich, welche Hindernisse bei Verfolgung dieser Idee entstehen: Es ist höchst schwierig, aus einem übergeordneten Ziel (z.B. Fahre von Raum F-334 in den Raum F-312) konkrete Verhaltensweisen abzuleiten. Selbst die in dem o.g. Beitrag generierten Programme („folge Korridor 1“) wären nicht in der Lage, die eigentliche Bewegungsrichtung des mobilen Systems zu erkennen und laufen so Gefahr, in die falsche Richtung zu steuern. Die Aufgabe, überhaupt die eigene Position zu erkennen und diese einer bekannten Gegend („Korridor 1“) zuzuordnen, klammern die Autoren völlig aus.

Auch die Umsetzung der in der Theorie so einfach zu implementierenden Einzelverhalten scheint in der Praxis mit hohem Aufwand verbunden zu sein. Fuzzy-Regeln zu finden, die dem Roboter in seiner Umgebung tatsächlich weiche und situationsangepasste Bewegungen ermöglichen, sind schwer und nur durch Experimente zu ermitteln. Implizit bestätigt dies auch die Danksagung in (TA01), in welcher speziell den Helfern für ihre „harte Arbeit“ beim Feintuning der Fuzzy-Controller gedankt wird.

Wohl aus diesem Grunde entwickelten sich einige der als Implementierungen der Fuzzy-Logik-Theorie gestarteten Arbeiten (ZRH94) konsequenterweise derart fort, die schwierig zu erstellenden Fuzzy-Regeln mit Hilfe neuronaler Lernverfahren zu generieren (ZS98). Dies gilt auch für die Arbeit von (ZTA).

Auch wenn sich Fuzzy-Systeme systemimmanent robust gegenüber Sensorrauschen verhalten, bleibt die Tatsache bestehen, dass reaktive Systeme den Schwerpunkt ihrer Handlungsgrundlage weniger auf Planung, sondern mehr auf den Eingang von Sensordaten legen. Dadurch entsteht wiederum ein Unsicherheitsfaktor in den Steuerdaten der Aktoren, der sich bei einem ca. 200 Kilogramm schweren Roboter direkt in einen Unsicherheitsfaktor für die ihn umgebenden Menschen übersetzen lässt. Angesichts dieser Überlegungen wurde davon Abstand davon genommen, *mobiled* in ein auf Verhalten und Fuzzy-Logik basierendes System umzuwandeln.

2.4 Lokalisierung

Wie der Titel dieser Arbeit schon zeigt, beschäftigt sie sich mit *mobilen* Robotersystemen. Diese zeichnen sich dadurch aus, dass sie in der Lage sind, ihre eigene Position im Raum zu ändern. Motiviert ist dies durch die Hoffnung, dass sich Roboter durch Entwicklung menschenähnlicher Fähigkeiten ihrer Umwelt angepasster verhalten und somit ihnen übertragene Aufgaben besser bearbeiten können.

Die Evolution hat den Menschen nach einem langen Zeitraum mit hochentwickelten Sinnesorganen versehen, die durch afferente Bahnen mit dem zentralen Nervensystem verbunden sind und so zur Fähigkeit der Propriozeption[5] führten. Um sich an einen gegebenen Ort zu bewegen, muss ein Roboter zuerst wissen, wo er sich momentan befindet. Was nun bei Menschen Propriozeption genannt wird, heißt bei Robotern Selbstlokalisierung und soll genau dieses Wissen liefern.

Der Mensch nutzt zur Lokalisierung seiner selbst hauptsächlich sein visuelles System. Einige Arbeiten haben demnach versucht, Kameras einzusetzen, um den eigenen Standpunkt zu ermitteln (MKS97).

Ein prinzipiell gangbarer Lösungsweg ist, eine ganze Reihe von Referenzbildern aufzunehmen, mit denen das in der aktuellen Roboterpose aufgenommene Bild verglichen werden kann. Dies bedeutet einen hohen zeitlichen Aufwand und auch einen großen Bedarf an Speicherplatz. Die Genauigkeit ist hier abhängig nicht nur von der Qualität der erzeugten Kamerabilder, sondern auch davon, dass sich die Umgebung zwischen den Aufnahmen optisch wenig verändert. Das Umräumen von Gegenständen oder auch eine Änderung der Lichtverhältnisse können das Verfahren zum Scheitern bringen. Nicht zuletzt zeigen Tests in (Gaw05), dass die Standortbestimmung mit Kamerabildern zu Lokalisierungsfehlern jenseits des für Serviceroboter tolerablen Bereichs führt. Für Außeneinsätze in höchst komplexen Umgebungen mit zeitgleich weniger ausgeprägten Ansprüchen in Bezug auf Genauigkeit der ermittelten Pose und Laufzeit des Verfahrens könnten solche sog. „visual servo“ Algorithmen laut (CMPV06) dennoch geeignet sein.

Eine anderes Verfahren ist die Stereo-Odometrie (SKLL05). Hierbei werden mit zwei Kameras Stereo-Bilder aufgenommen und die Position markanter Bildpunkte im Raum berechnet. Wird dieser Vorgang in kurzen Abständen wiederholt, kann die Verschiebung der immer gleichen Punkte im Raum Aufschluss über die zwischenzeitliche Bewegung des Roboters geben.

Schon vor Beginn dieser Arbeit war der Serviceroboter TASER in der Lage, sich selbst zu lokalisieren. Diese Fähigkeit war das Resultat einer Fusion mehrerer verschiedener Sensordaten. Einerseits wurde mit jeder Übermittlung von Sollgeschwindigkeiten an die Antriebsmotoren die von ihnen durch Drehung im gleichen Zeitintervall zurückgelegte Strecke erfragt. Durch die Verwendung dieser Odometriedaten lässt sich die Pose des Roboters relativ zum Beginn seiner Fahrt ermitteln. Leider schleichen sich bei wiederholten Messungen im Laufe der Zeit Fehler verschiedenster Art ein: durch Schlupf, Unebenheiten im Boden oder sogar kleinste Rechenfehler bei Verwendung von Gleitkommaarithmetik weicht die ermittelte Pose immer stärker von der tatsächlichen Pose des Roboters ab. Odometrie allein ist also für eine präzise Selbstlokalisierung nicht ausreichend.

Aus diesem Grund nutzt *mobiled* die Messwerte beider Laserscanner zur Verbessung der geschätzten Pose. Im Flur des Arbeitsbereichs sind zum jetzigen Zeitpunkt insgesamt 54 Lasermarken verteilt. Diese bestehen lediglich aus 15cm langen und ca. 2cm breiten reflektierendem Streifen, die mit ihrer Rückseite an Wände und Ecke geklebt werden. Auf der gleichen Höhe wie die Laserscanner der mobilen Plattform montiert, sorgen sie für besonders starke Reflektionen des Laserstrahls und heben sich so von normalen Hindernissen ab. Mit dem Wissen der relativen Position mehrerer Lasermarken zur mobilen Plattform kann *mobiled* nun auf die Position des Roboters schließen.

Da diese Lokalisierung äußerst zuverlässig sehr genaue Ergebnisse liefert, wurde sie im Zuge dieser Arbeit nur an das neue System angepasst, nicht jedoch grundlegend verändert. In Kapitel

[5] Propriozeption ist die menschliche Fähigkeit, die Lage des eigenen Körpers im Raum einzuschätzen

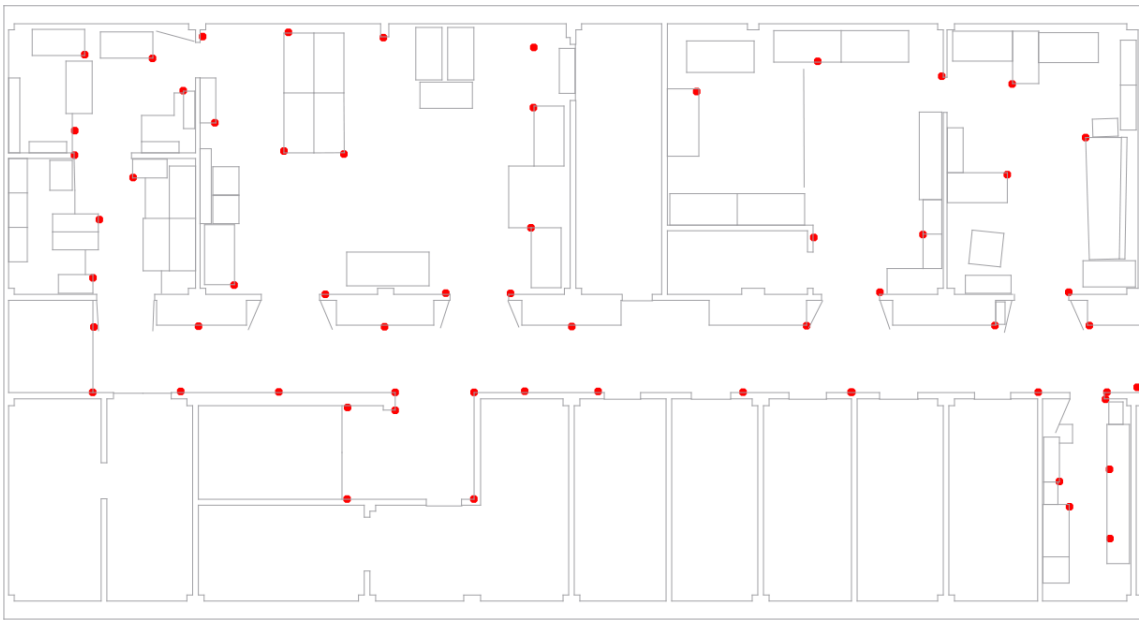


Abbildung 2.3: Abbildung der im TAMS-Flur montierten und in der Karte entsprechend eingetragenen Lasermarken. Insgesamt 59 Lasermarken ermöglichen eine präzise Selbstlokalisierung der mobilen Plattform.

3.2.7.3 wird auf die technischen Details dieser Lösung eingegangen, auf Seite 70 werden mögliche Erweiterungsszenarien diskutiert.

2.5 Der klassische Ansatz

Eine andere Vorgehensweise ist die Verwendung eines eher klassischen Ansatzes, welcher sich eng an der weiter oben beschriebenen SPA-Architektur orientiert. Eigenständige Programmteile übernehmen jeweils feste Aufgaben wie Lokalisierung, Fahrtplanung, Bewegungssteuerung und Kollisionsvermeidung. So nutzt das System seine Sensoren, um ein Abbild seiner Umgebung (und seine Position in ihr) zu erstellen. Darauf aufbauend plant es eine Fahrt an einen gewünschten Zielpunkt und führt diese mit seiner Bewegungssteuerung aus.

Um dem Nachteil hoher Latenzen zwischen den einzelnen Schritten zu begegnen, sind die einzelnen Abläufe möglichst unabhängig voneinander und weitestgehend nebenläufig implementiert. Die Sensorik bleibt während der gesamten Fahrt aktiv, aktualisiert ständig die Position der mobilen Plattform und bremst ab, sobald sie ein Hinderniss im Weg erkennt.

Diese Methode hat sich in den bisherigen Versionen von *mobiled* als praktikabel und vor allem als sehr sicher erwiesen. Ferner sind Fehler leicht aufzuspüren und zu beseitigen - es ist kein langwieriges und fehlerträchtiges Optimieren von Parametern, Netzgewichtungen oder Fuzzy-Regeln nötig. Somit fiel die Entscheidung leicht, das bestehende System zu verbessern und in seiner Funktionalität zu erweitern. Der Wunsch, neben grundlegenden Bewegungen auch weiche Kurven fahren zu können, führte zur Ausschreibung dieser Diplomarbeit und soll nun mit diesem Verfahren umgesetzt werden.

Ein weiteres Argument die bestehende Basis weiterzuverwenden ist, dass eine große Menge im Fachbereich verwendeter Anwendungen auf dem Programm, seinem Kommunikationsprotokoll und somit in Teilen auch seiner internen Umsetzung basieren. Einen komplett anderen Ansatz zu verfolgen würde bedeuten, dass viele dieser Anwendungen angepasst werden müssten.

Nach der Festlegung auf das verwendete Verfahren galt es, die bestehende Hard- und Softwarearchitektur zu analysieren und letztere in einem nächsten Schritt an die neuen Forderungen anzupassen. Dies ist das Thema des foglenden Kapitels 3.

3 Architektur

3.1 Hardware

3.1.1 USB-IO-Warrior

Am Roboter sind mehrere Geräte montiert, die während des Betriebs nur zeitweise Verwendung finden. Aus Gründen der Energieeffizienz sind diese über softwareseitig aktivierbare Relais mit der Energieversorgung verbunden. Konkret handelt es sich hierbei um den USB-IO-Controller *IO-Warrior*[1], der mit fünf Relais bestückt ist.

An diesen Relais sind jeweils ein Laserscanner, Lüfter zur Abfuhr von Wärme der Antriebsmotoren, die Neige- und Schwenkeinheit der Stereo-Kameras und die Stromversorgung der BarretHand angeschlossen.

Auch für dieses Gerät liefert der Hersteller Treiber für linux 2.4 und 2.6 mit, so dass eine Shared-Library *iowarrior.so* zur Verfügung steht, um den USB-Controller anzusprechen.

3.1.2 CAN-Bus

Wie in Kapitel 1.2 bereits erwähnt, sind die Antriebsmotoren, der Empfänger für Fernsteuerungen und die Akkumulatoren über CAN-Bus an den Hostrechner angeschlossen. Im Detail handelt es sich um einen Bus mit einer Datenrate von 1 MBit/s, der über den PCAN-Dongle der Firma PEAK-System[2] mit dem Parallelport des PCs verbunden ist. Siehe dazu Abbildung 3.1. Mit Strom wird der Adapter aus dem PS/2-Port des Rechners versorgt, der Anschluss für die Tastatur wird hierbei durchgeschleift.

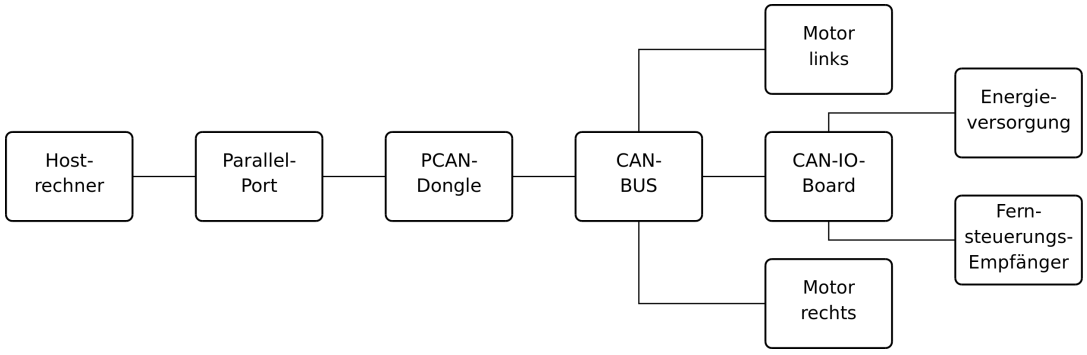


Abbildung 3.1: Der CANBus mit angeschlossenen Geräten

[1] http://www.codemercs.com/IOWarriorD.html, letzter Aufruf 2008-04-18
[2] http://www.peak-system.com/db/de/pcandongle.html, letzter Aufruf 2008-04-20

Im Lieferumfang des Adapters befinden sich Treiber für linux in den Versionen 2.4 und 2.6, die bei erfolgreicher Initialisierung die Gerätedateien */dev/pcan{0-16}* bereit stellen. Durch Aufrufe der `ioctl()`-Methode auf einem Datei-Deskriptor, der auf eine dieser Geräte-Dateien verweist, können so CAN-Pakete gesendet und empfangen werden. Als Argument im Aufruf dieser Methode wird hierbei entweder `PCAN_READ_MSG` oder `PCAN_WRITE_MSG` verwendet.

Im Gegensatz zu den meisten anderen Bussystemen verwendet CAN-Bus keine Adressen, um einzelne Busteilnehmer anzusprechen. Statt dessen beinhaltet jede über den Bus versandte Nachricht einen Objektidentifier, durch den auf den Inhalt des Pakets (z.B. Werte der Drehgeschwindigkeiten oder Batteriespannung) geschlossen werden kann. Dies hat dann Vorteile, wenn mehrere Geräte den gleichen Informationstyp verarbeiten. Während in anderen Bussystemen die gleiche Information mehrfach gesendet werden müsste, genügt bei CAN ein einzelnes Paket.

Zu sendende CAN-Pakete tragen den o.g. Objektidentifier an erster Stelle. Danach folgt ein Byte, welches das Paket-Kommando (bspw. `CMD_MOTCTRL_SETCMDVAL` zum Setzen einer Motorgeschwindigkeit) repräsentiert. Weitere 7 Byte stehen für beliebige Daten zur Verfügung, in unserem Beispiel würden Byte 6 und 7 die gewünschte Drehgeschwindigkeit angeben.

Empfangene CAN-Pakete beinhalten zuerst 6 Datenbytes. Das siebte Byte entspricht dem Objektidentifier. Das achte und letzte Byte ist aufgeteilt: Die ersten 6 Bits geben das Kommando an, welches zum Versand dieses Antwortpakets führte, während die letzten zwei Bits zur Übermittlung von Fehlercodes reserviert sind.

Diese Interna sind durch die Klasse `CanMessage` gekapselt. Für nähere Informationen siehe die Datei `canmessage.h`.

3.1.3 CAN-IO-Karte

Der Fernsteuerungsempfänger und die Energieversorgung haben selbst keine CAN-Schnittstelle. Ersterer stammt aus dem Modellbaubereich und legt das Signal am Ausgang pulsmoduliert an, bei letzterer wurde einfach ein mit A/D-Wandler ausgestatteter Eingang der CAN-IO-Karte mit den Akkumulatoren verbunden.

Die IO-Karte ist so programmiert, dass es nur Pakete mit einem Objetkidentifier von `0x0101` verarbeitet, ihre Antworten tragen den Code `0x0100`. Auf diese beiden Werte sind `CAN_ID_IO_CMD` und `CAN_ID_IO_REPLY` in `cmd_ioboard.h` gesetzt. An gleicher Stelle werden auch die für dieses CAN-Gerät möglichen Kommandos definiert.

Um das IO-Board zu nutzen, muss es zuerst durch Senden eines Pakets wie in Tabelle 3.1 initialisiert werden.

Sender	**Empfänger**	**Kommando**	**Wert**
mobiled	CAN_ID_IO_CMD	CMD_IOBOARD_CONNECT	-
CAN_ID_IO_CMD	mobiled	CMD_IOBOARD_CONNECT	-

Tabelle 3.1: CAN-Pakete zur Initialisierung der CAN-IO-Karte

Nun ist es möglich, die Werte des Fernsteuerungsempfängers durch die Pakete in Tabelle 3.2 abzufragen. Im Antwortpaket repräsentiert das erste Byte den Wert der Seitensteuerung, das zweite Byte den Wert von Gas bzw. Bremse. Beide Werte werden in Ganzzahlen (*int*) umgewandelt.

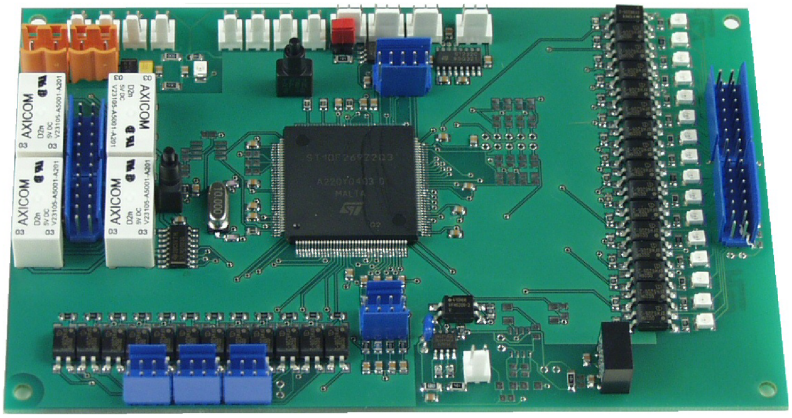


Abbildung 3.2: Die CAN-IO-Karte bietet 16 digitale Eingänge (rechts), 12 digitale Ausgänge (links unten), 4 Relais (links), 8 analoge Eingänge unter und drei Enkodereingänge rechts neben der CPU.

Sender	Empfänger	Kommando	Wert
mobiled	CAN_ID_IO_CMD	CMD_IOBOARD_GETJOYVAL	-
CAN_ID_IO_CMD	mobiled	CMD_IOBOARD_GETJOYVAL	Steuerung

Tabelle 3.2: CAN-Pakete zur Initialisierung der IO-Karte

Die Spannung der Energieversorgung kann ganz analog erfragt werden. Das Anfrage-Paket muss als Kommando `CMD_IOBOARD_GETVBATT` enthalten, das Antwort-Paket enthält den gewünschten Wert wieder in den ersten zwei Bytes. Wie aus `battery.cpp` ersichtlich, bedarf dieser Wert vor Rückgabe noch einer etwas komplizierteren Umrechnung. Weitere technische Details zur IO-Karte finden sich auch in (Neo04).

3.1.4 Antriebsmotoren

Nach jeder Unterbrechung ihrer Stromversorgung müssen die Antriebsmotoren in einer genau definierten Prozedur initialisiert werden, bevor sie Rotationskommandos entgegennehmen können. Dafür muss von Seiten der Software zuerst eine Verbindung mit den E/A-Einheiten beider Motoren aufgenommen werden, was durch das Senden von CAN-Paketen wie in Tabelle 3.3 erfolgt.

Die Motoren verarbeiten CAN-Pakete mit den in `drive.h` deklarierten Identifiern `CAN_ID_LEFT_CMD` bzw. `CAN_ID_RIGHT_CMD` - Antwortpakete tragen die Identifier `CAN_ID_LEFT_REPLY` und `CAN_ID_RIGHT_REPLY`. Mögliche Kommandos sind in `cmd_motctrl.h` definiert.

Auf jedes Kommando bzw. Paket signalisieren die Motoren (wie alle an den CAN-Bus angeschlossenen Geräte) in einem Antwortpaket, ob der jeweilige Befehl erfolgreich war und der gewünschte Zustand erreicht wurde.

Wie aus Abbildung 3.3 ersichtlich, befindet sich der Motor anfangs im Zustand *BrakeClosed* und wird durch das Senden von CAN-Paketen wie in Tabelle 3.4 in den fahrbereiten Zustand gebracht. Diese Initialisierungsschritte übernehmen die Methoden

Sender	Empfänger	Kommando	Wert
mobiled	MOTOR LINKS	CMD_MOTCTRL_CONNECT	-
MOTOR LINKS	mobiled	CMD_MOTCTRL_CONNECT	-
mobiled	MOTOR RECHTS	CMD_MOTCTRL_CONNECT	-
MOTOR RECHTS	mobiled	CMD_MOTCTRL_CONNECT	-

Tabelle 3.3: CAN-Pakete zur Initialisierung der Motorkommunikation

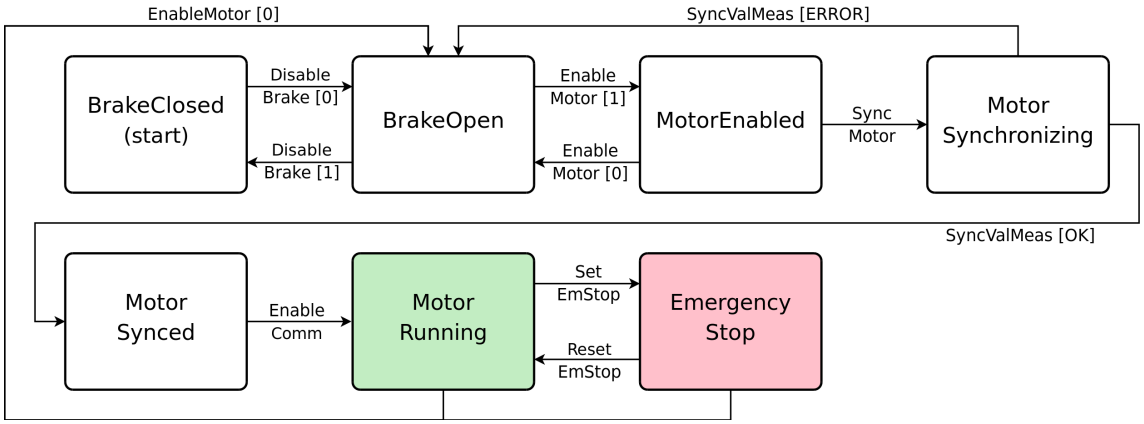


Abbildung 3.3: Zustandsdiagramm eines Antriebsmotors - Die Initialisierungssequenz muss die Motorsteuerung ausgehend vom Zustand „BrakeClosed“ in den Zustand „MotorRunning“ überführen.

```
void Drive::setBrakeEnabledLeft(bool enable);
void Drive::setMotorEnabledLeft(bool enable, int torque);
void Drive::syncMotorLeft(int torque);
void Drive::enableCommutationLeft(void);
void Drive::initializeAngleLeft(void);
```

der Klasse „Drive“ für den linken und in entsprechend abgewandelter Form für den rechten Antriebsmotor. Der `int torque`-Parameter der zweiten Methode entpuppte sich nach einigen Versuchen und anschließender Nachfrage beim Hersteller als funktionslos. Da nicht ausgeschlossen ist, dass er zu einem späteren Zeitpunkt wieder Verwendung finden könnte, wurde er in der Methodensignatur belassen.

All diese Methoden sind allerdings nicht von außerhalb der Klasse aufrufbar - die Initialisierung der Motorkommunikation erfolgt im Konstruktor der Klasse Drive, die Initialisierung der Motoren

Sender	Empfänger	Kommando	Wert
mobiled	MOTOR	CMD_MOTCTRL_DISABLEBRAKE	1
mobiled	MOTOR	CMD_MOTCTRL_ENABLEMOTOR	1
mobiled	MOTOR	CMD_MOTCTRL_SYNCHMOTOR	
MOTOR	mobiled	CMD_MOTCTRL_SYNCHMOTOR	aktueller Winkel
mobiled	MOTOR	CMD_MOTCTRL_ENABLECOMM	-

Tabelle 3.4: CAN-Pakete zur Initialisierung eines Antriebsmotors

Sender	Empfänger	Kommando	Wert
mobiled	MOTOR	CMD_MOTCTRL_SETCMDVAL	Geschwindigkeit
MOTOR	mobiled	CMD_MOTCTRL_GETSTATUS	Status
MOTOR	mobiled	CMD_MOTCTRL_GETPOSVEL	Position

Tabelle 3.5: CAN-Pakete zum Setzen einer Geschwindigkeit

selbst durch Aufruf von `void Drive::startMotors(void);`

Um die Motoren nun tatsächlich in Bewegung zu setzen, müssen ihnen CAN-Pakete mit Befehlen zum Setzen von Sollgeschwindigkeiten gesendet werden. Dies übernehmen die Methoden `void Drive::setMotorSpeedLeft(int speed)` bzw. `void Drive::setMotorSpeedRight(int speed)` . Diese rechnen die übergebene Translationsgeschwindigkeit in Mikrometern pro Sekunde in Enkoderinkrementierungen pro Sekunde um. Letzterer Wert beschreibt dem Motor, wie häufig pro Sekunde die internen Enkoder inkrementieren sollten, damit die gewünschte Geschwindigkeit erreicht ist. Er verhält sich linear zur Drehgeschwindigkeit.

Als Antwort auf jedes CAN-Paket zum Setzen der Drehgeschwindigkeit antwortet der Motor mit zwei CAN-Paketen. Das erste Paket enthält eine Statusangabe, das zweite den zuletzt ausgelesenen Enkoder-Wert.

Aus Sicherheitsgründen stoppen die Motoren ihre Drehbewegungen, wenn sie länger als eine Zehntelsekunde keine Bewegungskommandos erhalten. So wird ausgeschlossen, dass ein Absturz des Steuerungsprogramms zu einem unkontrolliert umherfahrenden Roboter führt. Da eine präzise Fahrtsteuerung des Roboters ohnehin mehr als zehn Geschwindigkeitsänderungen pro Sekunde erfordert, stellt diese Beschränkung kein Problem dar.

3.1.5 Bremsen

Die Bremsen der Motoren basieren auf einem Metallstift, der durch eine Feder automatisch in eine Position geschoben wird, in der dieser die Rotation der Antriebsräder verhindert. Soll die Bremse gelöst werden, wird der Stift durch einen Elektromagneten aus dieser Sperrstellung gezogen und dort so lange gehalten, bis die Bremse wieder angezogen wird. Der Begriff „Bremse“ ist insofern weniger treffend, als es sich hier eher um eine Blockierung der Räder handelt. Daraus ergibt sich, dass diese Vorrichtung nicht zum Abbremsen des Roboters, sondern zur Sicherung seines Stillstands verwendet werden sollte.

Dies ist eine sehr sichere Lösung, weil diese Feststellbremsen konstruktionsbedingt kaum verschleißen. Außerdem folgt zwangsläufig, dass der Roboter bei Abfall der Betriebsspannung stehen bleibt, weil das nun fehlende Magnetfeld den Metallstift in die Blockierungsposition fallen lässt. Natürlich folgt daraus, dass *mobiled* stets darauf achten muss, die Bremsen nicht während der Fahrt zu aktivieren. Ein weiterer Nachteil dieses Prinzips ist, dass der o.g. Elektromagnet recht viel Energie verbraucht, während die Bremsen gelöst sind. Somit ist es sowohl aus Gründen der Sicherheit wie der Energieeffizienz sinnvoll, die Bremsen vor der Fahrt möglichst lange angezogen zu lassen und nach der Fahrt schnell wieder zu aktivieren.

3.2 Software

3.2.1 Module

Bislang bestand mobiled aus genau einem Programm, dass zum Absetzen von Befehlen über seine TCP/IP-Schnittstelle angesprochen wurde. Diese Architektur - dargestellt in Abbildung 3.4 - erlaubte *mobiled* allerdings nicht, sich z.B. im Falle eines versperrten Weges erneut direkt mit dem externen Routenplaner in Verbindung zu setzen, um eine alternative Strecke zu planen. Auch war es nicht möglich, dem Programm zur Verwaltung der Umgebungskarte vom Roboter erkannte Hindernisse mitzuteilen.

Bisher gestaltete sich der typische Ablauf derart, dass durch die Benutzeroberfläche *genView* ein Fahrauftrag erstellt wurde. genView fragte daraufhin genPath nach einer Route von der Start- zur Zielkonfiguration. genPath wiederum benötigte zur Erstellung einer Route alle Elemente der Karte von genMap. Die Antwort von genPath an genView bestand aus einer Liste von Wegpunkten, welche genView stückweise an das mobilerobot-Modul weitergab, welches wiederum mobiled steuerte.

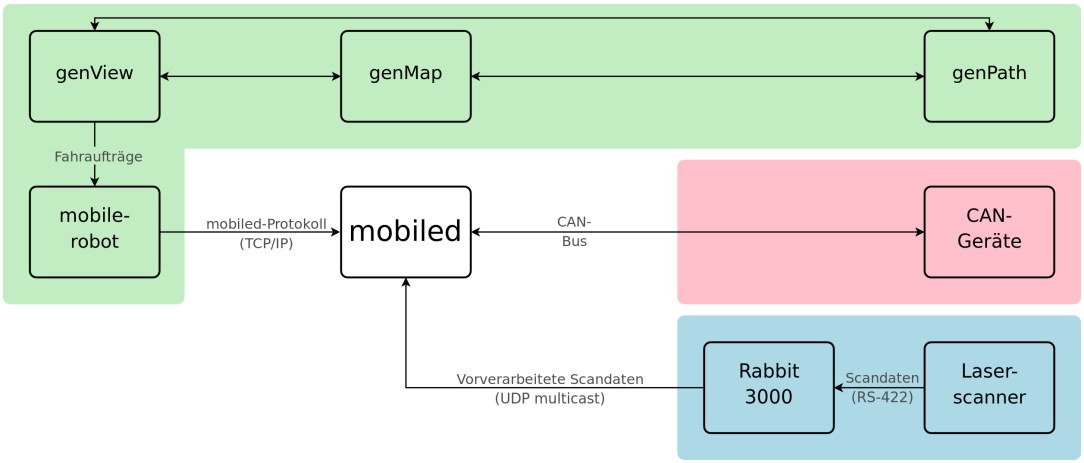


Abbildung 3.4: Die bisherige Software-Architektur

Im Rahmen dieser Arbeit wurde diese Architektur deswegen grundlegend verändert: Aus *mobiled* wurde das Programmmodul zur Ansteuerung des CAN-Busses ausgegliedert: Es existiert nun ein Programm *CanServer*, vorgestellt in Kapitel 3.2.3, das die am CAN-Bus angeschlossenen Geräte über eine TCP/IP-Schnittstelle bereitstellt.

Ein Klient zur direkten Ansteuerung von *mobiled* übers Netzwerk wurde erstellt, um letzteres Programm einfach testen zu können. Es findet allerdings im produktiven Einsatz keine Verwendung.

Ferner wurden die Programme genPath und genMap mit *mobiled* gekoppelt, um der Plattform Kollisionsvermeidung und Routenplanung zu ermöglichen. Diese Programme werden in Kapitel 3.2.6.3 und 3.2.6.2 vorgestellt.

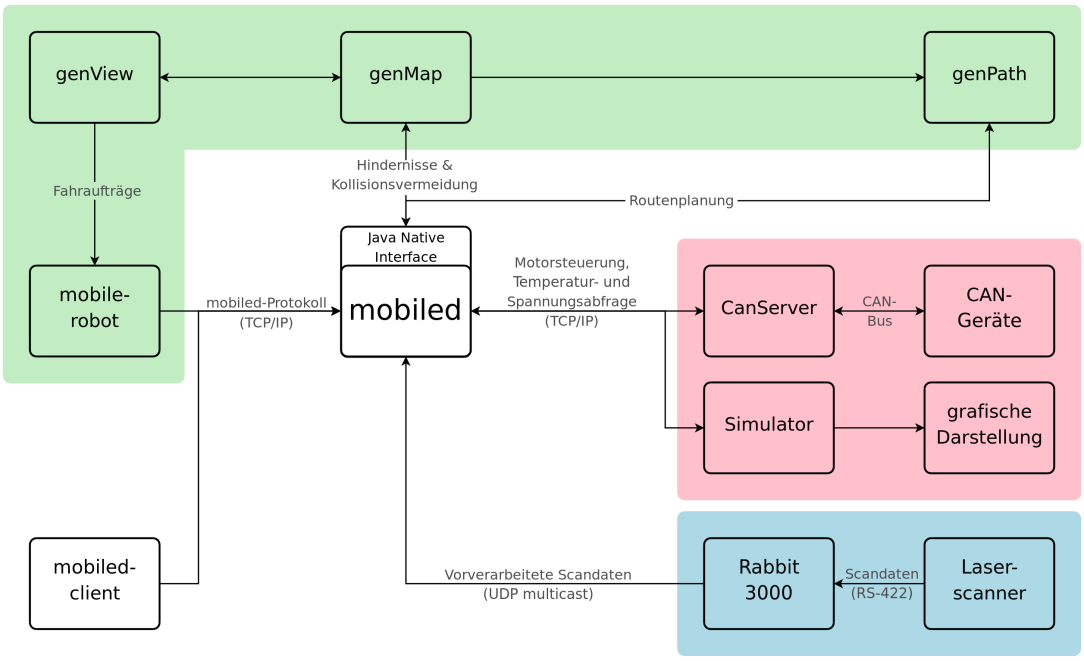


Abbildung 3.5: Die neue Software-Architektur

3.2.2 Simulator

Da der Roboter auch von anderen Studenten verwendet wurde, stand er während der Entwicklung von *mobiled* leider nicht immer für Testfahrten zur Verfügung. Aus diesem Grunde wurde ein Simulator programmiert. Dieses Programm stellt den Roboter und seine Umgebung mit einer grafischen Oberfläche auf einer Landkarte dar. Sowohl die zur Umwelt gehörenden Linien (Wände, Schränke etc.) als auch die Position der Lasermarken werden aus jeweils einer Datei geladen.

Entweder mit der Maus oder durch direkte Eingabe einer Pose kann der Roboter in seiner Umwelt verschoben werden. Radgeschwindigkeiten, Motortemperaturen, Werte des Fernsteuerungsempfängers und die Batteriespannung können über Bedienelemente eingestellt werden. So können grundlegende Bewegungsverhalten des Roboters getestet werden, ohne dass ein weiteres Programm (wie *mobiled*) benötigt wird.

Während der Entwicklung dieses Simulators fiel auf, dass die bisher innerhalb *mobiled* für Rotationsbewegungen verwendete und in (LaV06, Ch.13) aufgeführte Formel zur Rotation des Roboters $u_w = u_r - u_l$ - wobei u_l die Rotationsgeschwindigkeit des linken und u_r die Rotationsgeschwindigkeit des rechten Rades ist - das Verhalten des Roboters bei komplexeren Bewegungen nicht ausreichend genau abbildet: Wird nur ein Rad des Roboters bewegt, so würde er sich nach dieser Formel auf der eigenen Höhenachse drehen; dies entspricht aber nicht seinem tatsächlichen Verhalten (siehe Abbildung 3.6a). Steht ein Rad still während sich das andere bewegt, so dreht sich die Plattform um das stillstehende Rad. Bislang war diese Modellierung ausreichend und nie als Problem aufgefallen, weil die früheren Versionen von *mobiled* die Räder nur entweder in genau gleicher (Vorwärts- und Rückwärtsfahrt) oder genau entgegengesetzter Geschwindigkeit (Rotation um die Mitte der Antriebsachse) drehen konnten.

Um diesem Problem zu entgegnen, wurde der Klasse *Pose* eine Methode `int Pose::advance(const int advanceL, const int advanceR)` hinzugefügt. Bekommt diese Methode die Fortschritte beider Räder in Mikrometer übergeben, so bestimmt sie daraus eine neue Pose und gibt die insgesamt vom Mittelpunkt der Plattform zurückgelegte Strecke (wieder in Mikrometern) zurück.

Hier findet der folgende Algorithmus Verwendung:

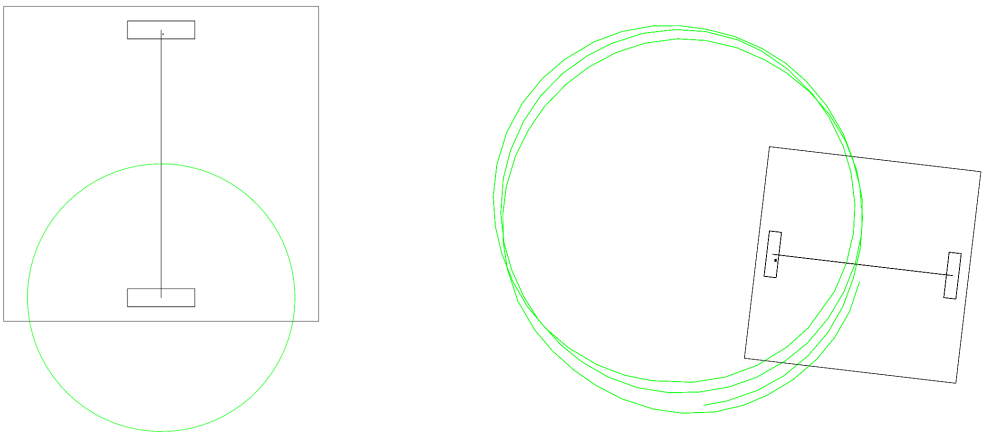
1. Die Räder werden einzeln nach ihrer Orientierung und ihrem Fortschritt bewegt, als wären sie nicht durch eine Achse verbunden.
2. Beide Räder werden nun durch eine Linie A verbunden.
3. Vom Mittelpunkt der Linie A wird eine Linie L zum linken Rad und eine Linie R zum rechten Rad gezogen.
4. Die Länge der Linien L und R wird mit der Länge der Roboterachse verglichen.
5. Die Längen der Linien L und R werden um die Differenz aus Schritt 4 im Verhältnis der zugehörigen Radfortschritte gekürzt.
6. Beide Räder werden an das Ende der Linien L und R verschoben.
7. Erneut wird eine Linie zwischen beiden Rädern erstellt, deren Winkel ist die neue Roboterorientierung.

Bewegt sich ein Rad nicht, wird es weder in Schritt 1 noch in Schritt 5 verschoben, da es sich ja auch im Verhältnis zum anderen Rad nicht bewegt hat. Bewegt es sich weniger als das gegenüberliegende Rad, so wird seine Stellung in Schritt 5 entsprechend weniger korrigiert. So ist gegenüber der vorigen Formel aus (LaV06) sichergestellt, dass sich die Plattform in den Fällen, in denen die Räder nicht mit einer gleichen Betragsgeschwindigkeit drehen, korrekt verhält.

Der Roboter verhält sich im Simulator im Gegensatz zur Realität - abgesehen von minimalen Ungenauigkeiten in der Gleitkommaarithmetik des Prozessors - perfekt. Es gibt keinen Schlupf beim Anfahren oder Abbremsen, die Räder sind prinzipiell unendlich schmal und exakt gleich groß. Um auch im Simulator ein gewisses Maß an Realität zu erreichen, ohne all diese Gegebenheiten tatsächlich schätzen und berechnen zu müssen, wird eine simulierte Ungenauigkeit in Form zufälligen Rauschens hinzugefügt: In einem Eingabefeld lässt sich das maximale prozentuale Rauschen festlegen, was dazu führt dass die Plattform in der Simulation ähnlich einer echten Plattform ihre perfekte Bahn verlässt.

Wirklichen Nutzen bringt der Simulator natürlich erst, wenn er verwendet werden kann, um *mobiled* selbst zu testen. An dieser Stelle spielt die modulare Architektur ihre Stärken aus: Wie in Kapitel 3.2.1 erläutert, ist der CAN-Bus der mobilen Plattform normalerweise von einem eigenständigen Programm namens *CanServer* geöffnet. Dieses Programm stellt dann die am CAN-Bus angeschlossenen Geräte über eine TCP/IP-Schnittstelle zur Verfügung. Diese Schnittstelle wird nun von *mobiled* angesprochen, um die Antriebe des Roboters zu steuern.

So gestaltet es sich sehr einfach, den Simulator mit einer identischen Netzwerkschnittstelle auszustatten. Würde *mobiled* sich normalerweise mit dem *CanServer* verbinden, um den Roboter zu



(a) Ein Rad der Plattform steht

(b) Bahn der Plattform mit Rauschen

Abbildung 3.6: Roboter-Simulation

bewegen, so bewegt das Programm nach Austausch von *CanServer* durch den Simulator den virtuellen Roboter. Für *mobiled* geschieht dies völlig transparent, d.h. dieser Tausch ist für den Klienten nicht feststellbar.

3.2.3 CanServer

Der *CanServer* ist ein Programm, dass beim Start den CAN-Bus auf Seite des Hostrechners initialisiert und auf einem in der Konfiguration festgelegten Port (voreingestellt ist `4321`) auf TCP-Verbindungen wartet.

Sobald sich ein Client verbindet, wird der Server-Socket des Programms geschlossen. So wird sichergestellt, dass zu jeder Zeit maximal ein Klient mit *CanServer* verbunden sein kann. Das verbundene Programm kann nun auf alle an den CAN-Bus angeschlossenen Geräte zugreifen.

Wird die Verbindung zwischen Client und *CanServer* unterbrochen, zieht *CanServer* aus Sicherheitsgründen automatisch die Bremsen der Antriebsräder an und wartet anschließend auf eine neue Netzwerkverbindung.

Hierdurch wird es möglich, *mobiled* auf einem anderen Rechner auszuführen als *CanServer*, sodass das Programm zur Steuerung der mobilen Plattform nicht länger auf der Plattform selbst laufen muss.

3.2.4 Fernsteuerung

Um sich mit der Architektur der Antriebe, des CAN-Busses und der Software vertraut zu machen, wurde zuerst ein einfaches Projekt realisiert: das Fahren der mobilen Plattform mit einer Funkfernbedienung.

Im Zuge dieses ersten Projekts wurde der *CanServer* implementiert, um Empfänger und Motoren ansprechen zu können. Anschließend wurde eine Client-Anwendung (*Remotecontrol*) programmiert, die auf die TCP/IP-Schnittstelle des CanServers zugreift, die Empfängerwerte ausliest,

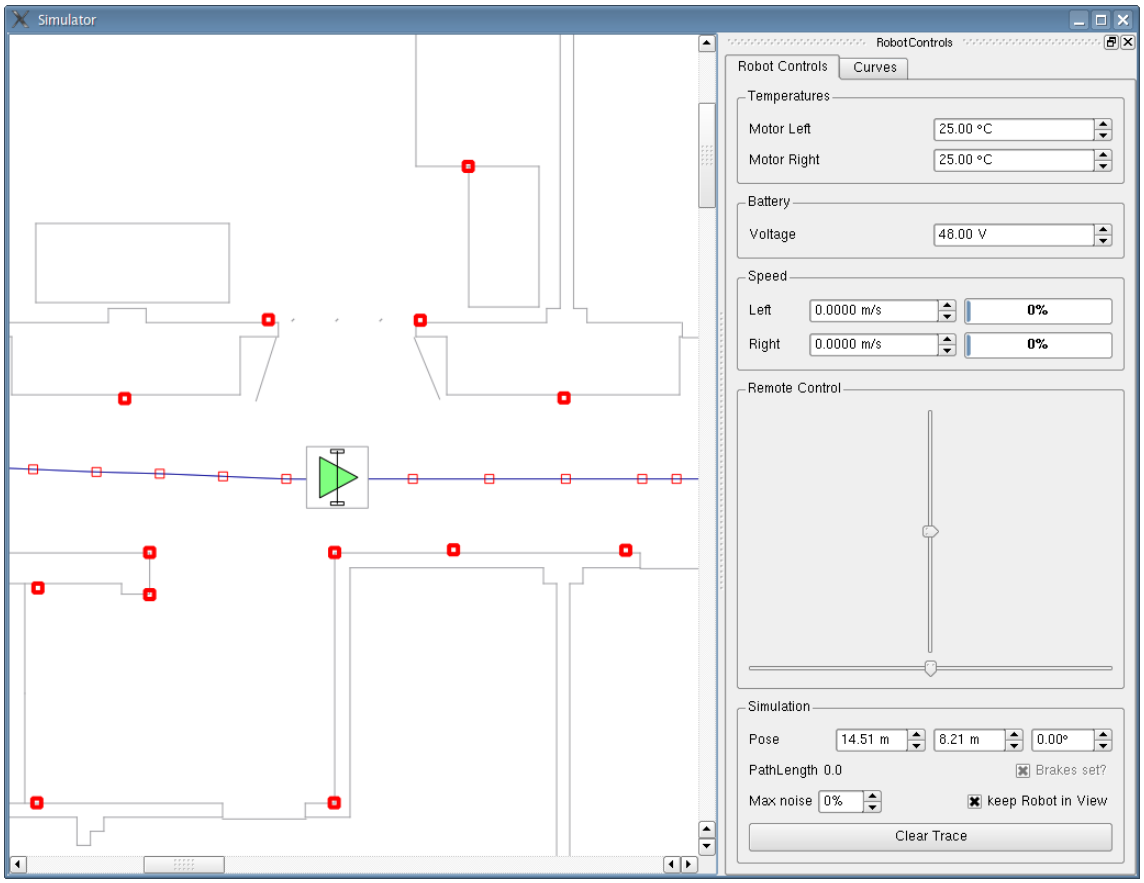


Abbildung 3.7: Der Service-Roboter im Simulator

Stellgrößen für die Motoren errechnet und diese dann - wieder über den *CanServer* - an die Motoren sendet. Um eine flüssige Bewegung zu ermöglichen, wird diese Kontrollschleife 15 Mal pro Sekunde ausgeführt.

Obwohl jeder Kanal der Fernsteuerung durch ein Byte repräsentiert wird und somit $2^8 = 256$ verschiedene Werte annehmen könnte, liefert die IO-Karte (siehe Kapitel 3.1.3) lediglich Werte zwischen -34 und 34. Das Rauschen in den Messungen der Funkfernsteuerungen führte dazu, dass sich die mobile Plattform auch bei Nullstellung beider Steuerhebel minimal bewegte. Um dies zu verhindern, werden in ihrem Betrag sehr kleine Steuerwerte ignoriert. Zur natürlicheren Handhabung der Beschleunigung wird dieser Wert vor Rückgabe an die aufrufende Funktion zu seiner dritten Potenz verrechnet.

3.2.5 Client

Um mobiled schnell und unkompliziert testen zu können, wurde eine eigene Benutzeroberfläche (siehe Abbildung 3.8) geschaffen. Dieses Programm verbindet sich statt des genRob-mobilerobot-Moduls direkt mit mobiled und kann die meisten der im mobiled-Protokoll enthaltenen Befehle absetzen. So können z.B. der aktuelle Bewegungsstatus der mobilen Plattform, aktuelle Motor-

temperaturen und die Batteriespannung abgefragt oder verschiedene Fahrten initiiert werden.

3.2.6 genRob

Einen großen Teil der Architektur stellen Programme der genRob GmbH dar. genRob steht neben dem Firmennamen auch für ein in Java programmiertes Rahmenwerk, auf dessen Basis verschiedene Programme implementiert wurden. Das genRob-Rahmenwerk zeichnet sich durch seine gute Unterstüzung von über mehrere vernetzte Rechner verteilte Datenverarbeitung aus. genRob-Programme (auch *Roblets* bzw. *Roblet-Server* genannt) lassen sich über ein Netzwerk an verschiedene Roblet-Server verschicken und können dort zur Ausführung gebracht werden.

Wie bereits erwähnt, sind das genRob Rahmenwerk und somit auch alle darin implementierten Anwendungen in Java programmiert. mobiled aber ist in C++ geschrieben. Der naheliegende Weg, die Programme über ein Netzwerkprotokoll miteinander kommunizieren zu lassen, widerspricht der Architektur des genRob-Rahmenwerks. Der effizienteste Weg ist somit, *mobiled* mit Hife des Java Native Interface (JNI) Roblets erstellen zu lassen. Diese werden dann zur Kommunikation mit den entsprechenden Roblet-Servern zu jenen verschickt und dort ausgeführt. Die konkrete Umsetzung von JNI wird in Kapitel 3.2.7.6 behandelt.

3.2.6.1 genMediator

Das Programm genMediator ist ein Verzeichnisdienst. Beim Start melden die verschiedenen genRob-Server über Jini[3] (also über UDP-Multicasts) die Verfügbarkeit ihrer Dienste bei genMediator an und bei ihrer Beendigung wieder ab. Wird nun ein sogenanntes Roblet gestartet, kann es sich beim Verzeichnisdienst nach einem verfügbaren Roblet-Server erkundigen und sich anschließend selbst zu diesem verschicken, um dort ausgeführt zu werden.

Wichtig ist somit, dass alle an einem genRob-Netzwerk teilnehmenden Rechner multicast-Netzwerke unterstützen. In einem Netzwerk dürfen beliebig viele Instanzen von genMediator aktiv sein, solange keine zwei Instanzen auf dem gleichen Rechner laufen.

3.2.6.2 genMap

Auch die Anwendung genMap ist als Roblet-Server implementiert. Sie dient der Verwaltung einer Umgebungskarte. Innerhalb dieser Arbeit findet genMap Verwendung, um die Umwelt des Roboters, also den Flur des Arbeitsbereichs TAMS zu verwalten. Konkret ist genMap in der Lage, eine beliebige Menge von Linien und Lasermarken auf einer Ebene zu speichern.

Um Elemente zu dieser Karte hinzuzufügen, wird zuerst ein Roblet erstellt. Diesem wird nun die Menge von Elementen übergeben. Anschließend nimmt das Roblet Kontakt mit genMediator auf und erfragt, ob ein genMap-Roblet-Server im Netzwerk verfügbar ist. Sobald ein solcher Server gefunden wird, verschickt es sich eigenständig dorthin und führt sich selber aus. Während dieser Ausführung werden die Elemente zur Karte auf dem (dann lokalen) Server hinzugefügt. Das Löschen von Elementen geschieht ganz analog.

[3] http://www.jini.org, letzter Aufruf 2008-04-24

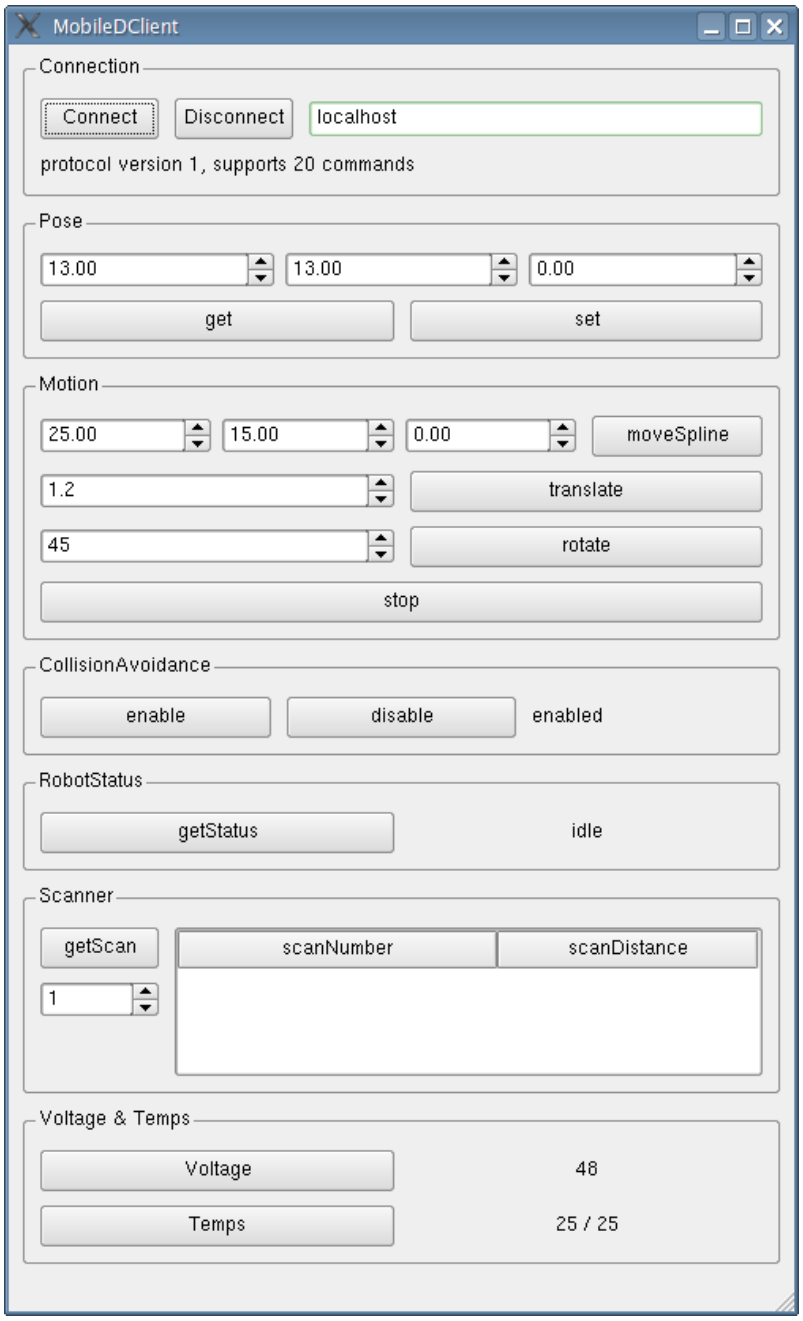


Abbildung 3.8: Der *mobiled*-client

mobiled kann Hindernisse mit Hilfe der Laserscanner erkennen. Diese Hindernisse werden registriert und über JNI mit einem Roblet in genMap eingetragen. Plant der Roboter nun eine neue Fahrt (beispielsweise, weil eines der Hindernisse seinen Weg blockiert), kann auf diese Objekte Rücksicht genommen werden.

3.2.6.3 genPath

genPath ist ein Roblet-Server und zuständig für die Ermittlung von Wegpunkten zwischen zwei Posen. Der Server meldet sich beim Start bei genMediator an und wartet auf eingehende Roblets. Sobald ein Roblet empfangen und ausgeführt wird, kann es zur Planung eines Weges die auf dem Server implementierten Pfad-Algorithmen nutzen.

mobiled wurde angepasst, um genPath direkt zu nutzen. Somit bestehen an mobiled gesendete Fahrbefehle einzig aus der gewünschten Zielpose. Dies bewirkt, dass mobiled ein Roblet mit Start- und Zielpose erstellt, welches anschließend an den genPath-Roblet-Server verschickt wird. Der Rückgabewert dieses Roblets ist eine Liste von Wegpunkten zwischen den übergebenen zwei Posen.

Plant mobiled eine Fahrt der mobilen Plattform, so schickt es über JNI ein Roblet zu genPath, welches die Wegpunkte zwischen Start- und Zielpose ermittelt. Da genPath sich zur Planung der Route der Karte von genMap bedient, kann es auch plötzlich erschienene und erst kürzlich von mobiled erkannte Hindernisse meiden.

3.2.6.4 genView

genView ist eine grafische Benutzeroberfläche mit dem Ziel, eine einfache Robotersteuerung zu ermöglichen. genView erkundigt sich nach dem Start bei genMediator, ob *genMap*, *genPath* und das *mobilerobot*-Modul zur Verfügung stehen. Wird ein laufender genMap-Roblet-Server gefunden, so stellt *genView* die Karte der Roboterumgebung grafisch dar. Wird das mobilerobot-Modul erkannt, so verwendet *genView* dieses zur Steuerung des Roboters und zeigt die mobile Plattform in der Karte an.

Verursacht durch die Änderung des *mobiled*-Protokolls (vgl. Kapitel 3.2.7.9) muss auch das *mobilerobot*-Modul zur Ansteuerung von *mobiled* angepasst werden. Diese wiederum führt zur Notwendigkeit, die Robotersteuerung in *genView* auf das neue *mobilerobot*-Modul zu portieren. Da die Programmquellen von *genView* momentan leider nicht zur Verfügung stehen, werden diese Portierungsarbeiten wohl erst nach Abschluss dieser Arbeit beendet sein.

3.2.6.5 IoWarrior

Das iowarrior-genRob-Modul ist ein weiterer Roblet-Server. Er bietet eingehenden Roblets die Möglichkeit, die lokal per USB angeschlossene IoWarrior-Karte zu steuern. Im Detail funktioniert dies wieder durch ein Roblet, welches zusammen mit Informationen über die zu setzenden Zustände an diesen Roblet-Server geschickt wird. Hier kommt das Roblet zur Ausführung und

ruft entsprechende Teile der iowarrior-genRob Programmierschnittstelle (API) auf. Diese wiederum kapselt die Funktionalität der im Lieferumfang der IoWarrior-Karte befindlichen Bibliothek `libiowarrior.so`. Aus dieser Architektur ergibt sich, dass das iowarrior-genRob-Modul auf dem Computer gestartet werden muss, an dem auch die IoWarrior-Karte per USB angeschlossen ist.

Da die Stromversorgungen der Laserscanner und der Motorlüfter über diese IoWarrior-Karte steuerbar sind, wurde *mobiled* dahingehend erweitert, dass es beim Start die Energieversorgung dieser Geräte aktiviert und beim Herunterfahren wieder deaktiviert. Das hierzu nötige Roblet wird wieder mit Hilfe des Java Native Interface (Kapitel 3.2.7.6) erzeugt.

3.2.7 mobiled

In den nächsten Abschnitten wird zuerst auf die Gesamtarchitektur der Software Bezug genommen. So soll deutlich werden, welche Module während des Betriebs miteinander kommunizieren und warum dies nötig ist. In der Folge werden die einzelnen Programmteile von *mobiled*, deren Implementierungen und dahinter stehende Überlegungen diskutiert.

3.2.7.1 Klassenhierarchie

Beim Betrachten der Abbildung 3.9 wird schnell klar, dass *mobiled* eine relativ komplexe Programmstruktur aufweist. Dies ist leider nicht zu ändern, da viele Programmteile untereinander Daten austauschen müssen. Damit dieses Senden von Daten an andere Module nicht dazu führt, dass überall im Programm (möglicherweise nicht länger gültige) Zeiger umhergereicht werden, wurden viele Klassen als Singletons[4] umgesetzt.

Dies ist auch deswegen vorteilhaft, weil ein Großteil der verwendeten Klassen zur Laufzeit des Programms aus intrinsischen Gründen nur *ein* Objekt ihres Typs instantiieren sollten: Es wäre unsinnig, zwei Objekte vom Typ `Robot` zu verwenden, da *mobiled* nunmal nur einen Roboter steuert. Analog kann für die auch als Singleton implementierten Klassen `Java`, `Correspondence`, `Localization`, `CollisionAvoidance`, `NetworkServer` und `LaserScanner` argumentiert werden (die Klasse LaserScanner kapselt die Funktionen aller Laserscanner).

Konkret bedeutet die Verwendung von Singletons, dass in den Konstruktoren der jeweiligen Klassen keine Zeiger auf Objekte o.g. Klassen übergeben und anschließend gespeichert werden müssen: Die einzig existierende Instanz z.B. der Lokalisierung kann von überall durch Aufruf der statischen Methode `Localization::instance()` referenziert werden. Sollte das Objekt noch nicht instantiiert worden sein, so geschieht dies beim ersten Aufruf für den Aufrufenden völlig transparent. Angesichts der Verwendung von Threads müssen hierbei jedoch Semaphoren bzw. sogenannte Mutexe[5] verwendet werden, um Wettlaufsitationen[6] zu verhindern.

[4] Singleton - zu deutsch „Einzelstück“ - beschreibt eine Klasse, von der während der Laufzeit des Programms nur eine Instanz existieren kann. Anstatt weitere Objekte zu instantiieren, kann vom Singleton jederzeit ein Zeiger auf diese Instanz erfragt und dann verwendet werden.

[5] engl. für „mutual exclusion“, also gegenseitiger Ausschluss

[6] Race-Conditions, zu Deutsch Wettlaufsituationen, sind Programmkonstellationen in denen das Ergebnis von Operationen von deren zeitlichem Ablauf abhängig ist. Da der zeitliche Ablauf mehrerer Threads zueinander nicht vorhersagbar ist, führen Race-Conditions zu unvorhersagbarem Programmverhalten. Dies ist eins der zentralen Problemfelder der parallelen Programmierung.

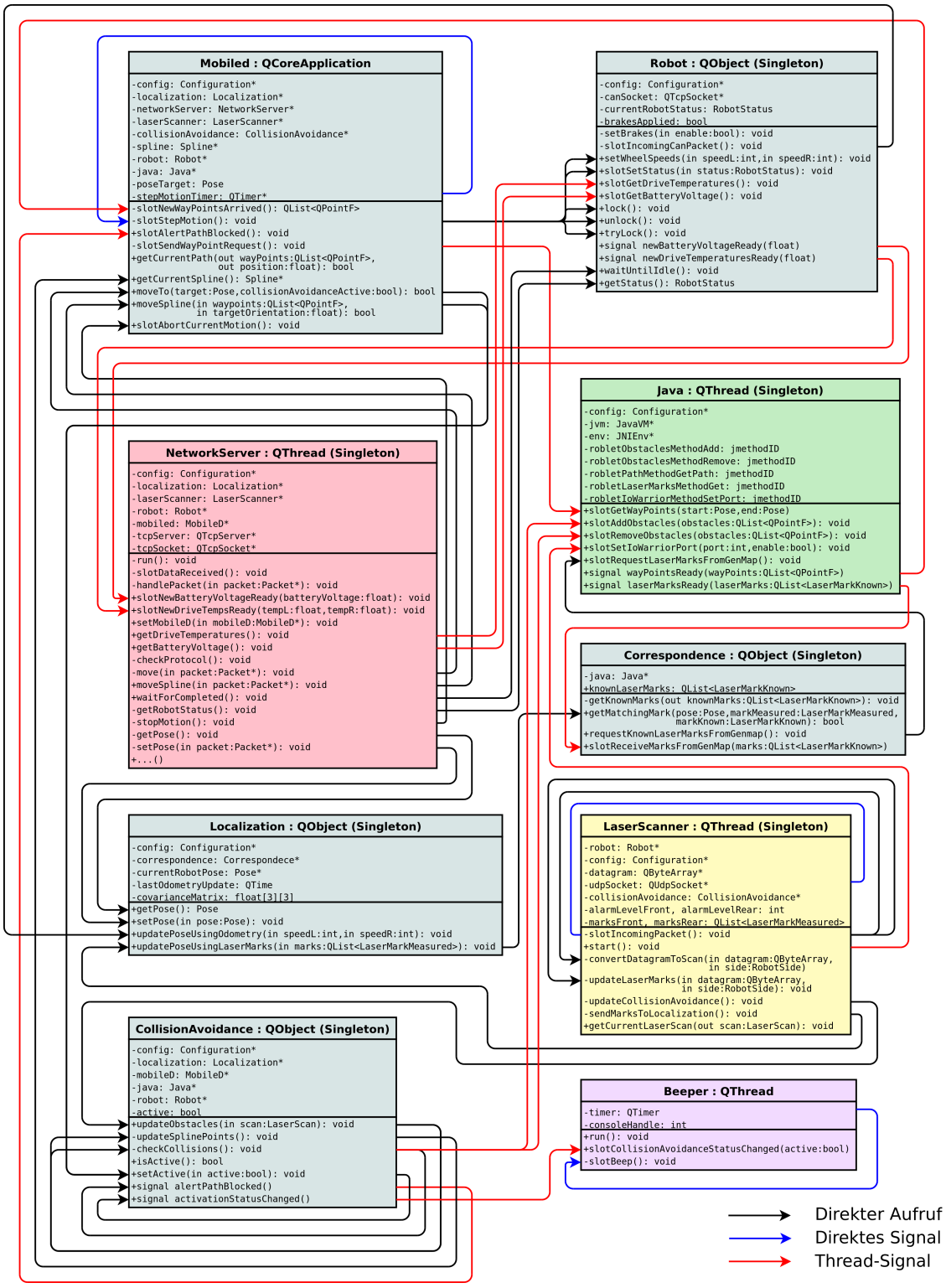


Abbildung 3.9: Das Klassendiagramm von *mobiled* - Objekte gleicher Farbe existieren im gleichen Thread.

Klassen bzw. Objekte, die im Klassendiagramm eine Hintergrundfarbe teilen, laufen im gleichen Thread. Das bedeutet, dass beispielsweise ein Aufruf von `Mobiled::slotStepMotion()` an `Robot::slotSetStatus()` direkt erfolgen kann. Da aber im gleichen Moment aus einem anderen Thread eine andere Klassenmethode von `Robot` aufgerufen und somit in einem für `Robot::slotSetStatus()` kritischen Zeitpunkt eine Klassenvariable verändert werden könnte, kann es auch bei Aufrufen innerhalb eines Threads zu Wettlaufsiutationen kommen. Aus diesem Grund sind alle Methoden, die aus verschiedenen Threads aufrufbar sind mit einem Thread-Mutex gesichert. Rufen also mehrere Threads Methoden auf, die bei gleichzeitigem Ablauf zu ungünstigem Verhalten führen können, bewirkt ein den Methoden gemeinsames Mutex, dass die Threads diese Methode nur nacheinander ausführen können.

3.2.7.2 Logmeldungen

mobiled konnte bislang genau einen Kommandozeilenparameter interpretieren: Der Schalter „-d“ hatte zur Folge, dass sämtliche zur Fehlersuche nützlichen Ausgaben von *mobiled* auf die Konsole geschrieben wurden. In Anbetracht der Tatsache, dass auch die Verarbeitung von eingehenden Daten der Laserscanner und Errechnung der Sollgeschwindigkeiten beider Räder mindestens 32 Mal pro Sekunde erfolgte, ist es unschwer denkbar wie unübersichtlich die Anzeige dieser Meldungen werden konnte.

Aus diesem Grund wurde das System zur Ausgabe von Logmeldungen überarbeitet. Den Schalter „-d“ gibt es weiterhin, jedoch kann ihm eine Liste von Programmmodulen übergeben werden, deren Logmeldungen momentan von Interesse sind - nur diese Meldungen werden dann auf der jeweiligen Konsole ausgegeben.

So ist es nun möglich, die Logmeldungen der Programmmodule

```
BeeperThread, Can, CanDevice, CollisionAvoidance, Conversion
, Correspondence, Battery, Drive, Java, LaserMark, LaserScan
, LaserScanner, Localization, MobileD, MobileDClient,
NetworkServer, Packet, Robot, Spline, CanServer, Simulator,
TcpClient, RemoteControl
```

wahlweise an- oder auszuschalten. Da bei Entwicklung und Benutzung des Programms meist nur das Verhalten sehr weniger Programmmodule von Interesse ist, eröffnet diese Änderung eine deutlich komfortablere Möglichkeit zur Fehlersuche.

3.2.7.3 Lokalisierung

Wie in Abschnitt 2.4 bereits erwähnt, wurde die Lokalisierung aus der Vorversion von *mobiled* größtenteils übernommen. Ihr Herz besteht aus einem erweiterten Kalman-Filter. Benannt ist dieser nach dem ungarischen Mathematiker Rudolf Kalman, der seine Entwicklung schon in den 60er Jahren veröffentlichte (Kal60). Der ursprüngliche Kalman-Filter dient der Schätzung des Zustands eines linearen, dynamischen Systems. In dem hier vorliegenden Fall ist mit „dynamischen System“ der Roboter gemeint, mit dem Zustand seine Pose. Im Lokalisierungsteil der Software findet der erweiterte Kalman-Filter Verwendung, mit dessen Hilfe auch Zustände nichtlinearer

Systeme verarbeitet werden können. Die mathematischen Details des Kalman-Filters finden sich sehr ausführlich und verständlich in (TBF06) beschrieben, so an dieser Stelle dorthin verwiesen sei.

Die Tatsache, dass sich der Zustand des Filters durch iterative Messungen verändern bzw. verbessern lässt, macht ihn zu einem für Echtzeit- bzw. Robotikanwendungen hervorragend geeigneten Instrument. Durch sehr häufig wiederholte Messungen (32 pro Sekunde von den Motoren und 37,5 pro Sekunde von den Laserscannern) wird der Filter dem tatsächlichen Status des Roboters immer näher gebracht.

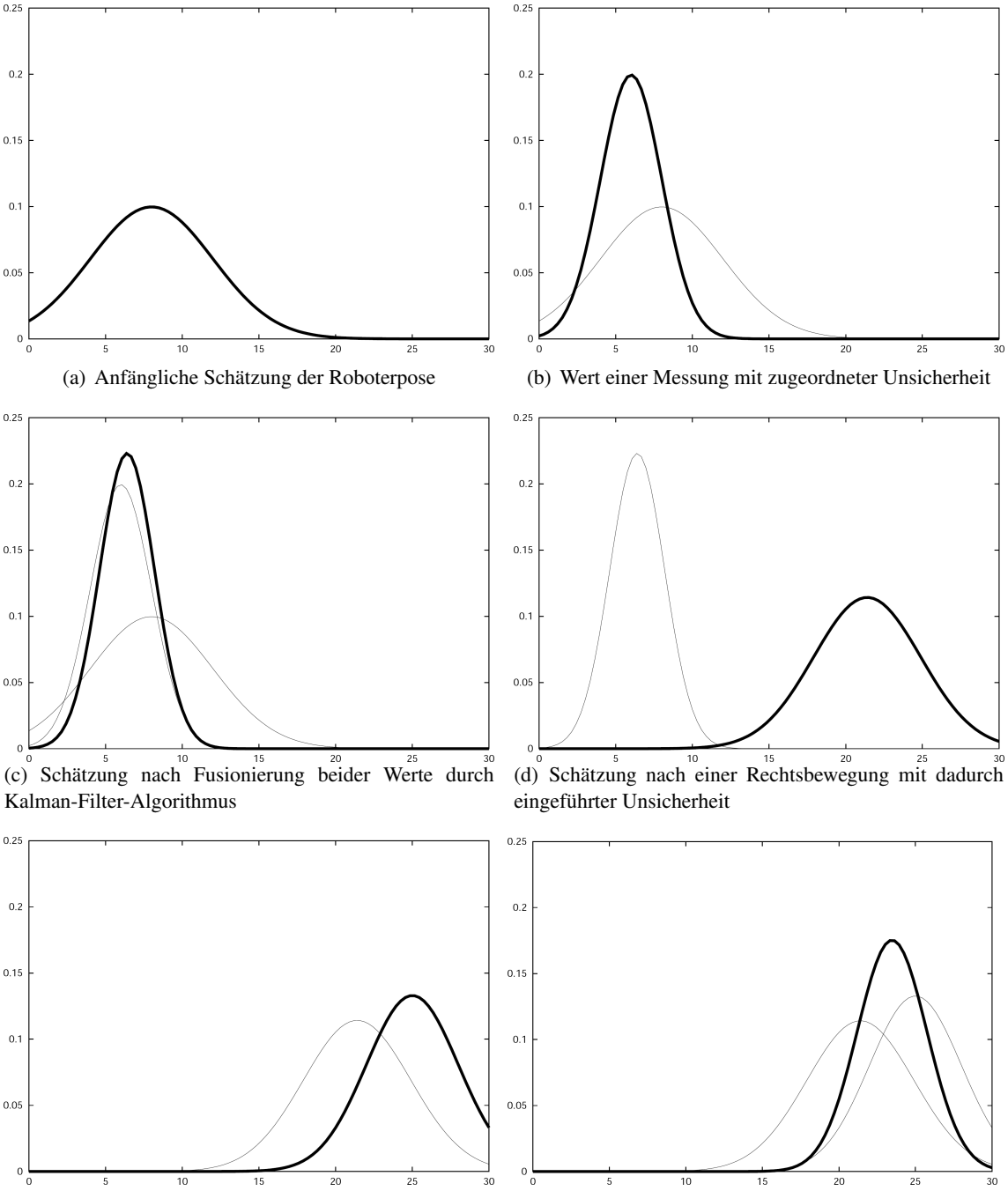
Grundsätzlich eignet sich das Verfahren sehr gut, um aus verrauschten oder anderweitig unzuverlässigen, wiederholten Messungen nicht nur einen Zustand, sondern auch die Wahrscheinlichkeit seiner Gültigkeit abzuschätzen. Die hierfür nötigen Berechnungen finden konkret in zwei Schritten statt:

- Die Eingangsdaten des Systems, also entweder Sollgeschwindigkeiten der Antriebsmotoren oder von den Motoren gemeldete, zurückgelegte Wegstrecken werden verrechnet, um daraus einen Systemzustand vorherzusagen. Dieser Vorhersage-Schritt - in der Literaur meist „predict“ genannt - findet je nach verwendeten Messdaten entweder in der Methode `updatePoseUsingWheelSpeeds(const int speedL, const int speedR)` oder `updatePoseUsingWheelAdvances(const float advL, const float advR)` statt. Hierbei kommt der gleiche Algorithmus (aus `Pose::advance(const int advanceL, const int advanceR)`) zum Einsatz, der auch in Abschnitt 3.2.2 der Simulation des Roboterverhaltens dient.

- Im zweiten Schritt („update“) werden die Lasermarken-Messungen der Laserscanner verwendet und mit den im aktuellen Zustand erwarteten Messungen verglichen. Die Differenz zwischen beiden bezeichnet man als *Innovation*. Sie wird verrechnet, um die vorhergesagte Pose zu verifizieren bzw. ihre Genauigkeit und Wahrscheinlichkeit zu erhöhen.

Im zweiten Schritt wird die Differenz zwischen der tatsächlichen Messung einer Lasermarke und dem erwarteten Messwert gebildet. Um den erwarteten Messwert bilden zu können, muss zu einer gemessenen Lasermarke ihre korrekte Position aus der Umgebungskarte ermittelt werden. Diese Aufgabe übernimmt die Methode `bool Correspondence::getMatchingMark(const Pose &pose, const LaserMarkMeasured markMeasured, LaserMarkKnown &markKnown)`. Ihr wird die aktuell geschätzte Pose des Roboters im ersten Parameter, sowie Winkel und Distanz zur gemessenen Lasermarke in der zweiten Stelle übergeben. Kann die Methode daraus eine in der Karte registrierte Marke ermitteln, liefert sie deren Koordinaten über den dritten Parameter und gibt `true` zurück. Andernfalls ist der Rückgabewert `false`.

Ein verbleibendes Problem ist die Tatsache, dass der Systemzustand beim Start der Lokalisierung unbekannt ist. Funktionalität, welche allein aus gemessenen Lasermarken auf eine Pose der mobilen Plattform schließen könnte, ist bislang nicht implementiert. Dies hat zur Konsequenz, dass dem Roboter seine eigene Pose nach Ausführung von *mobiled* ein mal mitgeteilt werden muss.



(a) Anfängliche Schätzung der Roboterpose

(b) Wert einer Messung mit zugeordneter Unsicherheit

(c) Schätzung nach Fusionierung beider Werte durch Kalman-Filter-Algorithmus

(d) Schätzung nach einer Rechtsbewegung mit dadurch eingeführter Unsicherheit

(e) Wert einer weiteren unsicheren Messung

(f) Schätzung nach Integration der Messung

Abbildung 3.10: Mit Hilfe eines Kalman-Filters geschätze Zustände eines Roboters. Auf der Abszisse aufgetragen ist seine eindimensionale Pose, die Ordinate ordnet jeder möglichen Pose eine Wahrscheinlichkeit zu. Bei Verwendung zweidimensionaler Posen wandelt sich die Darstellung mit Gaussglocken in eine Darstellung von Ellipsen. Diagramme entnommen aus (TBF06)

3.2.7.4 Konfiguration

Mit Hilfe einer Konfigurationsdatei veränderbare Einstellungen waren in *mobiled* bisher nicht vorgesehen. Durch Einführung einer Klasse `Configuration` und Anpassung aller übrigen Komponenten ist *mobiled* nun in der Lage, seine Konfiguration aus einer Datei zu lesen. Diese liegt standardmäßig in `/etc/mobiled.conf`.

In der Natur vieler Werte liegt die Anforderung begründet, dass sie nicht während mobileds Laufzeit verändert werden sollten. Dies gilt beispielsweise für Netzwerkports und IP-Adressen, Reifenumfang oder die Anzahl der Laserscanner. Bei anderen Parametern ist es nützlich, sie während des Betriebs anpassen zu können. So könnte es durchaus wünschenswert sein, die obere Geschwindigkeitsbegrenzung während des Betriebs an die Umgebung anzupassen. Dies ist nun möglich, da die hierfür nötigen Befehle im Netzwerkprotokoll implementiert sind (siehe Appendix A).

3.2.7.5 Echtzeitfähigkeit

Die Bewegungssteuerung einer mobilen Plattform ist eigentlich eine Aufgabe kontinuierlicher Art. Da Computer allerdings zeitlich diskret arbeiten, muss die Steuerung durch einen Rechner ebenfalls zeitlich diskret erfolgen. Dieser Widerspruch wird gewöhnlich so gelöst, dass der Steuerungscomputer eingehende Sensordaten in ausreichend hoher Frequenz verarbeitet und zur Ansteuerung der Aktoren verwendet. So lässt sich zumindest ein dem zeitkontinuierlichen ähnliches Verhalten erreichen. Die Frequenz, die zu einer zufriedenstellend funktionierenden Bewegungssteuerung notwendig ist, hängt von vielen Faktoren ab, hauptsächlich von der Bewegungsgeschwindigkeit der mobilen Plattform. Auf dieses Thema wird in Kapitel 5.1 eingegangen.

Um zu verhindern, dass der Roboter die für ihn vorgesehene Bahn verlässt, ist es wichtig, dass *mobiled* die beiden Antriebe in regelmäßigen zeitlichen Abständen ansteuert und die entsprechenden Radfortschritte ausliest. Die Steuerung muss also insofern echtzeitfähig arbeiten, als dass die Berechnung der Sollgeschwindigkeiten innerhalb eines zeitlichen Intervalls erfolgen muss. Dies bedeutet einerseits eine obere Schranke für die Zeitkomplexität der zur Berechnung der Radgeschwindigkeit verwendeten Algorithmen. Andererseits muss sichergestellt sein, dass die entsprechenden Methoden zur Bewegungssteuerung zuverlässig und in regelmäßigen Intervallen aufgerufen werden.

Um die Regelmäßigkeit jener Intervalle zu garantieren, wurde bislang die Linux-Echtzeituhr unter (`/dev/rtc`) verwendet. Zuerst wurde durch den E/A-Kontrollaufruf `RTC_IRQP_SET` die Interrupt-Rate festgelegt, dann wurden die Interrupts durch den E/A-Kontrollaufruf `RTC_PIE_ON` gestartet. Durch Aufruf von `select()` an der Echtzeituhr wurde nun sichergestellt, dass `Event()` exakt alle 32 Millisekunden ausgeführt wird.

Seit Version 4.3 verfügt die unter anderem auch für die Threads verwendete Bibliothek Qt[7] über die Möglichkeit, in jedem einzelnen Thread eine eigene Event-Loop einzusetzen. Sobald ein

[7] Qt ist eine plattformübergreifende Bibliothek zur Entwicklung C++-basierter Anwendungen. Obwohl Qt einstmals als Bibliothek zur Erstellung grafischer Oberflächen startete, lassen sich heute auch Programme ohne grafische Oberfläche realisieren. Qt ist kostenlos verfügbar unter http://www.trolltech.com/qt/, letzter Aufruf 2008-05-02

Thread eine eigene Event-Loop besitzt, können mit Hilfe von `QTimer`-Objetken in regelmäßigen Intervallen Signale emittiert werden, die wiederum andere Methoden ausführen können. Die `CPERIODICTIMER`-Klassen wurden durch solche Konstrukte ersetzt. Dies hat den Vorteil, dass eine Abhängigkeit zur Linux-Echtzeituhr nicht länger gegeben ist, was wiederum die Vorraussetzung beseitigt, *mobiled* als Superuser *root* auszuführen.

Details zur Zuverlässigkeit und Leistungsfähigkeit dieses Lösungsweges finden sich wiederum in Kapitel 5.1.

3.2.7.6 Java Native Interface (JNI)

Als Schnittstelle zwischen der C++-Anwendung *mobiled* und den Java-basierten genRob-Anwendungen dient das Java Native Interface. Dieses wird meist eingesetzt, um aus plattformunabhängigen Java-Anwendungen heraus nativen Code auszuführen. Auch möglich ist allerdings, JNI für Aufrufe in die andere Richtung zu nutzen, und genau das macht *mobiled* sich zunutze: Bei Bedarf kann von der C++-Seite eine Java-Methode aufgerufen werden. Hierbei handelt es sich immer um Methoden, die ihrerseits ein Roblet erzeugen (also eine Java-Klasse instantiieren) und dieses an einen passenden Roblet-Server versenden.

Die Klasse `Java` ist dafür verantwortlich, die wenig benutzerfreundliche Programmierschnittstelle des JNI zu kapseln. Sie ist als Singleton implementiert, da bei Erstellung eines Objekts dieser Klasse eine virtuelle Maschine erstellt wird - dies ist ein vergleichsweise hoher zeitlicher Aufwand, der durch Verwendung des Singletons nur ein Mal beim Programmstart zu Verzögerungen führt. Das ist deswegen von Bedeutung, weil einige Methoden der Java-Klasse sehr häufig aufgerufen werden:

- die Methode `Java::slotGetWayPoints(Pose start, Pose end)` wird mit zwei Posen aufgerufen und verursacht die Erstellung eines Roblets, dass auf dem genPath-Roblet-Server einen kollisionsfreien Weg von `start` zu `end` plant. Dies kann einige Sekunden dauern. Kehrt das Roblet vom Server zurück, werden die Daten ausgewertet und in eine Liste von Wegpunkten kopiert. Diese wird dann in dem Signal `wayPointsReady(QList<QPointF> wayPoints)` emittiert. Alle interessierten Programmteile können dieses Signal auffangen und die darin enthaltenen Daten verwenden.

- die Methode `slotRequestLaserMarksFromGenMap(void)` veranlasst, dass ein Roblet zum Kartenserver genMap gesandt wird. Dort lädt sich das Roblet die Position aller bekannter Lasermarken herunter (siehe Abbildung 2.3) und liefert diese zurück. Daraufhin wird das Signal `laserMarksReady(QList<LaserMarkKnown> laserMarks)` emittiert und zum Zwecke der Lokalisierung weiterverwendet.

- der Aufruf der Methode `slotSetIoWarriorPort(int port, bool enable)` schickt ein Roblet zum IoWarrior-genRob-Modul. Hier kann es - je nach übergebenen Daten - einen Ausgang der IoWarrior-Karte ein- oder ausschalten. Eingesetzt wird dies, um LaserScanner und Motorlüfter beim Start von *mobiled* zu aktivieren.

- `slotAddObstaclesToMap(QList<QPointF> &obstacles)` findet Verwendung, um von der Kollisionsvermeidung erkannte Hindernisse in der Umgebungskarte einzutragen. Wird später mit der in Punkt 1 genannten Methode ein neuer Pfad geplant, können diese neuen Hindernisse hierbei berücksichigt werden.

Immer, wenn ein Aufruf die Rücklieferung von Daten erfordert, wurde dies über ein asynchrones, also nicht-blockierendes Interface verwirklicht. So kann gewährleistet werden, dass *mobiled* während längerer Aufrufe (z.B. bei der Fahrtplanung) nicht „hängt“. Wäre dies der Fall, wäre es nicht möglich, eine Fahrtplanung durch Senden eines `CMD_STOP` Paketes abzubrechen.

3.2.7.7 Threads

Mit Threads bezeichnet man zeitlich unabhängig voneinander laufende Programmteile. Threads teilen sich den gleichen Adressraum und können so auf Objekte in anderen Threads des gleichen Programms zugreifen. Es ist jedoch weder vorhersagbar, welcher Thread zu welcher Zeit aktiv ist, noch in welcher Reihenfolge einzelne Threads aktiv (also dem Prozessor zugeteilt) sind. Auf einem Rechner mit n Prozessoren können bis zu n Threads gleichzeitig aktiv sein. Dies bedeutet einerseits eine bis zu n-fache Beschleunigung des Programms, andererseits muss bei ihrer Verwendung mit großer Sorgfalt auf korrektes Sperren und Entsperren von Ressourcen geachtet werden.

Bislang bestand mobiled aus vier verschiedenen Threads:

- MOTORFEEDER war ein Thread, der in festen zeitlichen Abständen von je 32 Millisekunden die Winkeländerungen beider Räder auslas. Diese Werte wurden dann innerhalb der Lokalisierung der Odometrie (also `CGENBASE::UpdateOdometry()`) übergeben.
- LASERFEEDER lief parallel, um eingehende UDP-Pakete mit Lasermessdaten zu empfangen. Diese Pakete wurden anschließend ausgewertet. Nach dem erfolgreichen Empfang und erfolgter Verarbeitung von Paketen beider Laserscanner (hierzu zählt auch das Umwandeln von radialen Scannerkoordinaten in kartesische Plattformkoordinaten) wurden diese Daten ebenfalls der Lokalisierung über `CGENBASE::UpdateLaser()` zur Verfügung gestellt.
- GENBASE war der Haupt-Thread. Die Klasse `CGENBASE` erbte hierbei von `CPERIODICTIMER`, welche wiederum von `CTHREAD` erbte. Im Unterschied zu `CTHREAD` enthält `CPERIODICTIMER` lediglich die zusätzliche virtuelle Methode `CPERIODICTIMER::Event()`, welche innerhalb des Threads in regelmäßigen zeitlichen Abständen aufgerufen wird. Somit wurde auch die von `CGENBASE` zwangsläufig zu implementierende Methode `CGENBASE::Event()` regelmäßig aufgerufen. Innerhalb dieser Methode wurden die Antriebsmotoren (über den `CMOTORFEEDER`-Thread) angesprochen, um der aktuellen Bewegung entsprechende Radgeschwindigkeiten zu setzen.
- CLIENT war für die Verwaltung der Netzwerkverbindung zu einem Netzwerk-Client verantwortlich. Die Klasse `CCLIENT` erbte von `CDETACHEDTHREAD`. Dieser unterscheidet sich von einem normalen `CTHREAD` dadurch, dass er die durch ihn alloziierten Ressourcen bei seiner Beendigung (z.B. nach Abbruch der Netzwerkverbindung) wieder freigibt. Über das Netzwerk eingehende Befehle wurden direkt an GENBASE weitergereicht.

Die Notwendigkeit einer Klasse wie `CDETACHEDTHREAD` wurde in der neuen Version umgangen, indem der `NetworkServer`-Thread zwar weiterhin nur eine Verbindung zur Zeit erlaubt, aber für beliebig viele zeitlich aufeinander abfolgende Verbindungen zuständig ist. Selbst wenn vor jeder Verbindung der alte Thread gelöscht und ein neuer erstellt würde, würde dies zu keinerlei Speicherlecks[8] führen, da ein `QThread` alle ihm als „Kinder“ zugewiesenen Objekte vor seiner eigenen Löschung selber freigibt.

Die Funktionen zum Ansprechen der in Java programmierten Komponenten genMap, genPath und genIoWarrior erfordern die Verwendung des *Java Native Interface*. Dieses wiederum setzt eine laufende *Java Virtual Machine* (JVM) im mobiled-Prozess voraus. Damit diese virtuelle Maschine von den anderen Programmteilen *mobiled*s möglichst unabhängig läuft, wurde sie in einem eigenen Thread implementiert; die Klasse `Java` erbt also von QThread.

Würde die JVM innerhalb des *mobiled*-Threads laufen, so wäre eine Robotersteuerung und zeitgleiche Nutzung der genRob-Komponenten nicht möglich, da einige Methoden der Java-Klasse zwangsläufig blockieren. Durch Auslagerung in einen separaten Programmablauf und eine möglichst asynchrone Schnittstelle ist es nun denkbar, dass *mobiled* dem Fahrtplanungssystem genPath einen Auftrag zu einer erneuten Routenplanung erteilt noch während *mobiled* selber die mobile Plattform aufgrund eines erkannten Hindernisses abbremst.

Um eine möglichst asynchrone Kommunikation zwischen den verschiedenen Threads sicherzustellen, wurden die meisten Informationen über durch Qt bereitgestellte QueuedConnections[9] realisiert. Dem Empfänger wird die über QueuedConnections zugestellte Nachricht in seine Event-Loop injiziert, so dass der Empfang erst dann registiert wird, wenn der Empfänger-Thread im Ruhezustand liegt. In der Praxis bedeutet dies im Vergleich zum direkten Aufruf über verschiedene Threads hinweg eine Verzögerung von einigen Mikrosekunden; diese Latenzen liegen allerdings weit unter den Werten, die den reibungslosen Programmablauf gefährden könnten.

Als Vorteil bedeutet die Verwendung oben genannter Methode allerdings, dass Kommunikation dieser Art auch ohne Verwendung von Semaphoren und Sperren völlig frei von Wettlaufsituationen ist. Dies ist ein nicht zu unterschätzender Vorteil, da die systemimmanent enge Vernetzung der einzelnen Programmkomponenten viele Kommunikationskanäle über Thread-Grenzen hinweg erfordert. Im Klassendiagramm in Abbildung 3.9 sind diese Kanäle in Rot eingezeichnet.

3.2.7.8 Ansteuerung der Bremsen

Die Ansteuerung der Bremsen in den Antriebsmotoren des mobilen Roboters erfolgte bei *mobiled* bislang durch den Klienten, der auch die Fahrbefehle erzeugte, so dass Befehle zum Anziehen und Lösen der Bremsen im mobiled-Protokoll implementiert werden mussten.

Bei der bisherigen Lösung mussten also externe Programme, deren Aufgabe prinzipiell nur die Übermittlung neuer Soll-Posen des Roboters war, zusätzlich auf den jeweiligen Status der Bremsen achten. Dies war nicht nur nötig, um die Energieversorgung des Roboters nicht unnötig zu

[8] Als Speicherlecks („memory leak“) bezeichnet man Arbeitsspeicher, der im gesamten Ablauf eines Programms zwar reserviert, aber nie mehr freigegeben wird.

[9] QueuedConnections liefern einem Empfänger-Thread die gesendeten Daten erst dan aus, wenn dieser sich in seiner Event-Loop (also in Ruhe) befindet. Für Dokumentation siehe http://doc.trolltech.com/latest/threads.html, letzter Aufruf 2008-05-02

belasten, sondern auch weil ein Fahrbefehl an einen Roboter mit festgestellten Bremsen zu einem Fehler geführt hätte.

Im Grunde sind die Bremsen des Roboters ein Implementierungsdetail. Gibt der Klient dem Roboter einen Befehl zur Bewegung, so müssen seine Bremsen vor Antritt der Fahrt deaktiviert sein. Hat der Roboter eine Bewegung abgeschlossen, sollten die Bremsen wieder aktiviert werden. Für Klienten von *mobiled* sollte es zu jedem Zeitpunkt unerheblich sein, ob die Bremsen angezogen sind oder nicht. In der neuen Version von mobiled wurden die Befehle zum Setzen und Lösen der Bremsen entfernt, denn *mobiled* kann diese nun selber zu den richtigen Zeitpunkten steuern.

Zuerst wurde diese Funktion durch das Senden von CAN-Paketen mit dem Kommando `CMD_MOTCTRL_DISABLEBRAKE` implementiert. Dies führte jedoch dazu, dass die Motoren - wie in Abbildung 3.3 zu sehen ist - in den Zustand `BrakeClosed` übergingen. Um nun die Bremsen wieder zu lösen, wurde die erneute Ausführung der kompletten Initialisierungsprozedur notwendig. Dies dauert jedoch für jeden der beiden Motoren einige Sekunden und führt so zu spürbaren Latenzen zwischen dem Zeitpunkt des Fahrkommandos und dem tatsächlichen Beginn der Fahrt.

Deswegen verwendet *mobiled* zum Setzen der Bremsen den Befehl `SET_EMSTOP`[10]. Wie aus dem Zustandsdiagramm der Motoren zu entnehmen ist, wechselt der jeweilige Motor dadurch in den Zustand `EmergencyStop`, die Spannung am Elektromagneten fällt ab und der Metallstift bewegt sich in die Ausgangsposition zurück - der Roboterantrieb ist blockiert.

Das Verlassen dieses Zustands geschieht durch Senden eines CAN-Pakets mit dem Kommando `RESET_EMSTOP`. In der Motorelektronik ist auch dieser Befehl mit nur einem Zustandsübergang abgearbeitet und erfolgt so zeitlich deutlich schneller als es bei Verwendung der `CMD_MOTCTRL_DISABLEBRAKE`-Befehle möglich wäre.

Softwareseitig ist der Zustand der Bremsen eng mit dem Zustand des Roboters gekoppelt, mögliche Werte sind in A.3.2.1 definiert. Befindet sich der Roboter in den Zuständen *idle*, *stalled* oder *movementSetup*, sind die Bremsen angezogen, in den Zuständen *moving* und *aborting* sind sie gelöst. Weitere Dokumentation der Zustände ist im Quelltext der Klasse `Robot` zu finden.

3.2.7.9 Das mobiled-Protokoll

> „Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.“ –Antoine de Saint-Exupery

Das Protokoll des *mobiled* umfasste vor dieser Arbeit 42 Befehle. Darunter befanden sich mindestens 11 Befehle, die entweder nicht implementiert oder zumindest nicht benutzt wurden. Außerdem befanden sich viele Befehlsgruppen im Protokoll, die mit recht wenig Aufwand mit einem einzigen Befehl (evtl. mit Parametern) umgesetzt werden konnten. Somit lag die Idee nahe, unbenutze Befehle zu entfernen und umständliche Befehlsgruppen auf ein Kommando zu reduzieren, um eine übersichtlichere Spezifikation zu erhalten. Dabei handelt es sich konkret um folgende entfernte Kommandos:

[10] EMSTOP steht hierbei für Emergency Stop, also Not-Aus.

- CMD_DEBUGGING zum (De)aktivieren der Logmeldungen
- CMD_GETMODE und CMD_SETMODE zum Verwalten inzwischen unbenutzter Betriebsmodi
- CMD_GETPOSITIONANDVARIANCE und CMD_GETPOSITIONODO zum Debuggen der Odometrie
- CMD_ADDMARK zum Hinzufügen einer Lasermarke
- CMD_MOVEMARK zum Verschieben einer Lasermarke
- CMD_DELETEMARK zum Löschen einer Lasermarke
- CMD_GETALLMARKS zum Auslesen aller aktuell bekannten Lasermarken
- CMD_GETALLMARKSINITIAL zum Auslesen der anfänglich bekannten Lasermarken
- CMD_GETALLMARKSWITHVARIANCES zum Auslesen der Lasermarken und ihrer Varianzen
- CMD_GETSCALE, CMD_SETSCALE und CMD_MODIFYSCALE zum Verwalten der Geschwindigkeitsskalierung
- CMD_FORWARD zum Starten einer Translation (trotz des Namens auch für Rückwärtsbewegungen) ohne Kollisionsvermeidung
- CMD_MOVEPOINT zum Starten einer Translation mit Kollisionsvermeidung
- CMD_ROTATEANGLE zum Starten einer Rotation mit Kollisionsvermeidung
- CMD_ROTATEPOINT und CMD_TURNANGLE zum Starten einer Rotation ohne Kollisionsvermeidung
- CMD_STALLED zur Abfrage, ob der Weg versperrt ist
- CMD_ISCOMPLETED zur Abfrage, ob sich der Roboter noch bewegt
- CMD_APPLYBRAKES zum Anziehen der Bremsen
- CMD_RELEASEBRAKES zum Lösen der Bremsen
- CMD_AREBRAKESRELEASED zur Abfrage des aktuellen Bremsenzustands
- CMD_GETSCANSCANNER zur Abfrage von Scannerdaten in radialen Scannerkoordinaten
- CMD_GETSCANPLATFORM zur Abfrage von Scannerdaten in kartesischen Plattformkoordinaten
- CMD_GETSCANRADIALPLATFORM zur Abfrage von Scannerdaten in radialen Plattformkoordinaten
- CMD_GETSCANWORLD zur Abfrage von Scannerdaten in kartesischen Weltkoordinaten
- CMD_GETALLLINES zur Abfrage aller Linien aus der Karte

- CMD_STARTODOLOGGING und CMD_STOPODOLOGGING zum (de)Aktivieren der Odometrie-Logmeldungen

Anstatt jedes Bewegungskommando zweimal aufzuführen, um die Kollisionsvermeidung während der Bewegung entweder ein- oder auszuschalten und dafür verwirrend ähnliche Bezeichnungen wie TURNANGLE und ROTATEANGLE zu verwenden, wird die Aktivierung der Kollisionsvermeidung nun von der Bewegung völlig unabhängig durch die Befehle GETCOLLISIONAVOIDANCE und SETCOLLISIONAVOIDANCE implementiert (siehe A.3.6.1 und A.3.6.2).

Die Verwaltung von Lasermarken wird nun nicht länger durch *mobiled* selber erledigt. Vielmehr bemüht mobiled seine JNI-Schnittstelle zu genMap, um die Positionen der Lasermarken zu erfragen. In einer frühen Programmversion wurden nicht nur die in der Karte verzeichneten Lasermarken, sondern auch die durch fortwährende Schätzungen der Lokalisierung aktualisierten Marken-Positionswerte verwaltet - dies begründet den Unterschied zwischen MARKS und MARKSINITIAL. Da diese Schätzungen die wirklichen Positionen der Marken nur unwesentlich besser beschrieben, die Lokalisierung auch ohne sie ausreichend präzise ist und die hinter diesen Befehlen stehende Funktionalität in der letzten *mobiled*-Version eh nicht länger vorhanden war, wurden die entsprechenden Befehle und Programmteile entfernt.

Früher war geplant, die Umgebungskarte innerhalb *mobiled* selbst zu verwalten. Bald stellte sich jedoch heraus, dass die Auslagerung dieser Funktion in ein externes Programm sinnvoll ist, weil mehrere Roboter gemeinsam auf eine einzige Karte zugreifen sollten: Die LINES-Befehle wurden überflüssig.

Weiterhin existierte eine Vielfalt von Befehlen, um Scannerdaten in allen möglichen Variationen (Scanner-, Plattform- und Weltkoordinaten in kartesischer oder Polarkoordinatenangabe) anzufordern. Im Sinne der Einfachheit des Protokolls wurden diese Kommandos durch ein einziges GETSCANRADIALSCANNER (A.3.5.3) ersetzt. Die Umrechnung in alle anderen Formate ist mit Hilfe der Befehle GETSCANNERPOSE (A.3.5.2) und GETPOSE (A.3.3.1) möglich.

Das vollständige neue Protokoll wird in Appendix A eingehend beschrieben.

3.2.7.10 Kollisionsvermeidung

Der Impuls $\vec{p}$ eines Objekts ist definiert als das Produkt aus seiner Masse m und seiner Geschwindigkeit $\vec{v}$. Beide Faktoren können bei dem in dieser Arbeit verwendeten Roboter während des Betriebs sehr große Werte annehmen, so dass eine Übertragung des Impulses $\vec{p}$ auf andere Massen - also ein Zusammenstoß - zu verhindern ist.

Nach Erteilung eines Fahrauftrages bemüht *mobiled* die Dienste von genPath, um einen kollisionsfreien Pfad zu erhalten. Dieser wird jedoch zum Zeitpunkt des Starts erstellt und kann seine Eigenschaft der Kollisionsfreiheit in einer dynamischen Umwelt schnell verlieren. Deswegen ist es notwendig, die noch zu fahrende Strecke wiederholt auf Kollisionsfreiheit zu prüfen. Wie weit die Kollisionsvermeidung den Spline von seiner aktuellen Position aus auf Hindernisse absucht, wird durch einen Suchweiten-Parameter (konfigurierbar in `Configuration::getCollisionAvoidanceLookAheadDistance()`) festgelegt.

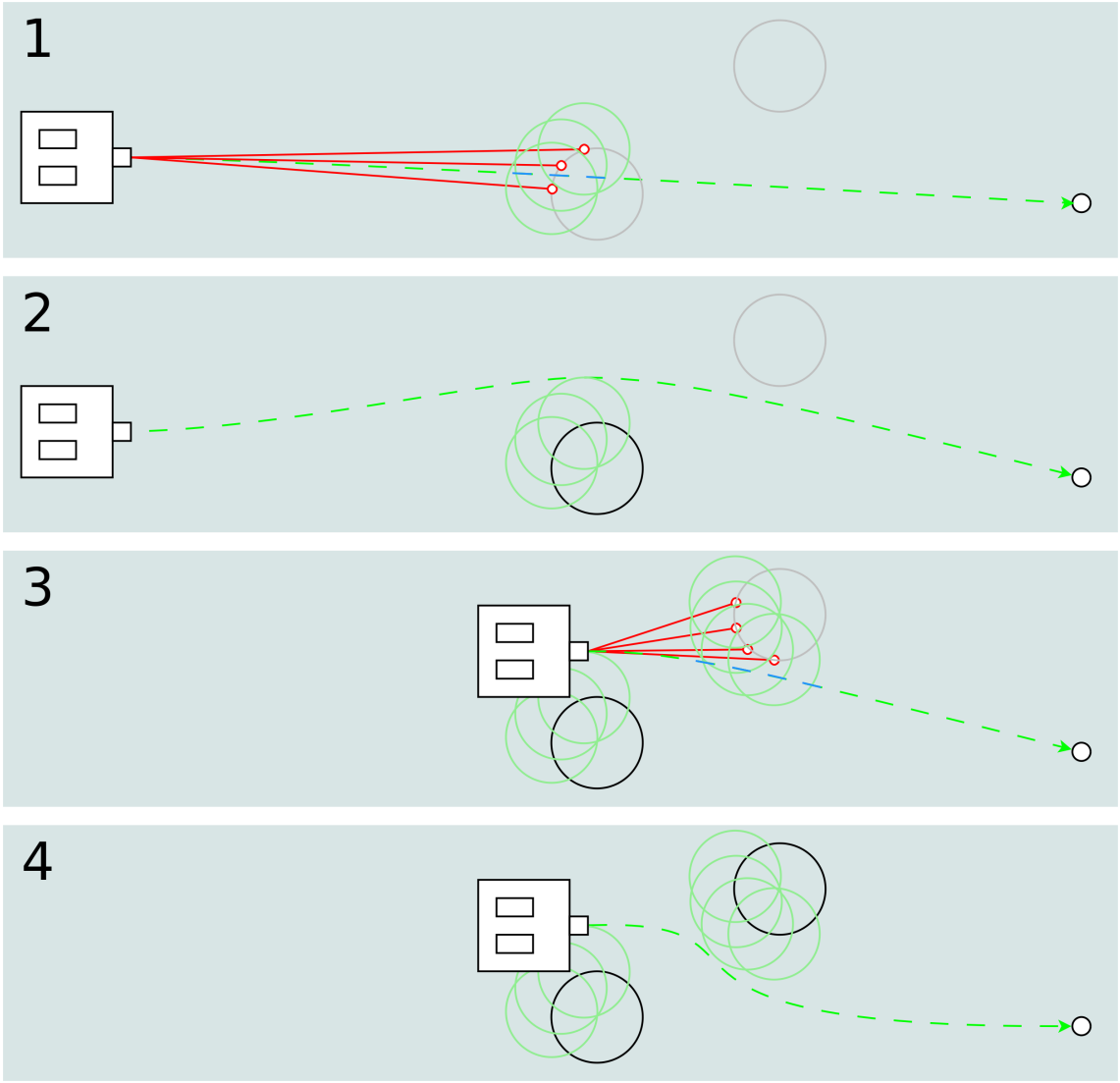


Abbildung 3.11: Arbeitsweise der Kollisionsvermeidung - In Phase 1 erkennt der Roboter mehrere Hindernisse (rote Kreise) innerhalb der Suchweite. Einige der Spline-Punkte (dargestellt in blau) weisen einen zu geringen Abstand zu Hindernissen auf. Diese Hindernisse werden in der Karte eingetragen und um den Radius der mobilen Plattform expandiert (grüne Kreise). Ein neuer Pfad wird geplant und in Phase 2 abgefahren. In Phase 3 werden wieder mehrere Hindernisse erkannt und mit Spline-Punkten verglichen; Da es erneut zur Kollision kommen würde, bremst der Roboter ab, trägt die Hindernisse in der Karte ein und ermittelt in der letzten Phase einen neuen Pfad zum Ziel.

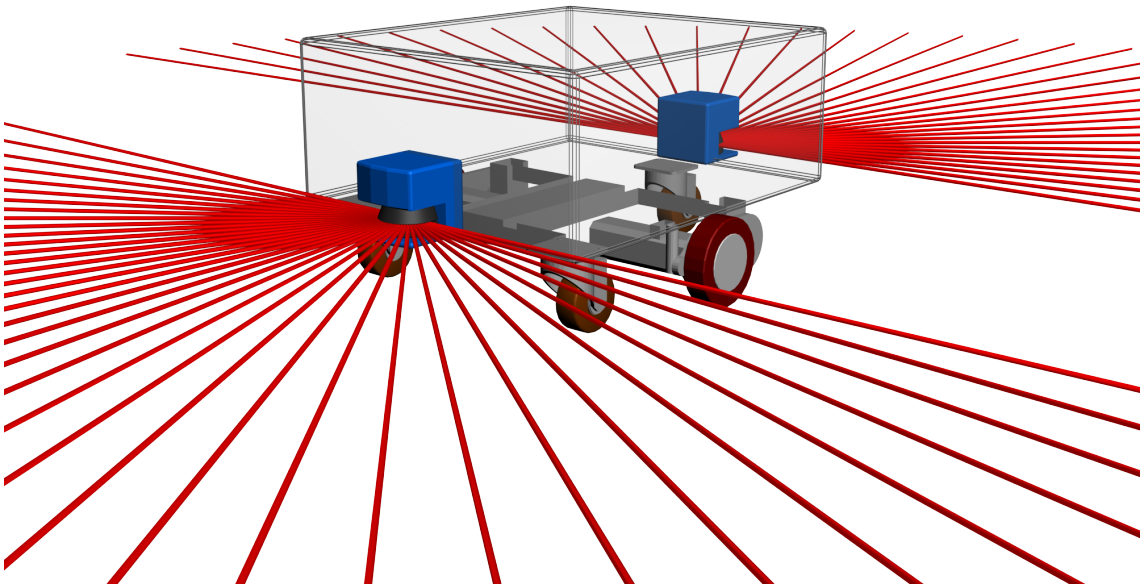


Abbildung 3.12: Anordnung der Laserscanner an der mobilen Plattform. Der Übersichtlichkeit halber ist nur jeder zehnte Strahlengang eingezeichnet. Sichtbar werden zwei „tote Winkel“ zu beiden Seiten der mobilen Plattform

Zur Erkennung der Hindernisse werden beide Laserscanner verwendet. Durch ihre Anordnung sind sie zwar nicht in der Lage, Objekte direkt links und rechts des Roboters zu erkennen; da die mobile Plattform als nicht-holonomes System auch nicht in diese Richtungen fahren kann, stellt dies kein großes Problem dar. Im Gegensatz dazu ist es durchaus problematisch, dass die Laserscanner ihre Umwelt auf lediglich einer Ebene abtasten können. Diese Ebene liegt (wie in Abbildung 3.12 angedeutet) ca. 20cm oberhalb des Bodens. Während die meisten Objekte wie Füße, Beine und Wände erkannt werden können, stellen schon Tische den Roboter vor große Herausforderungen: Die Beine des Tisches werden zwar registriert und folgerichtig gemieden; das Wissen, dass sich zwischen diesen Beinen aber eine ebensowenig passierbare Tischplatte befindet, muss der Roboter aus anderen Quellen beziehen.

Aus der Klasse `LaserScanner` wird in zeitlichen Abständen die Methode `CollisionAvoidance::updateObstacles(LaserScan *scan)` aufgerufen. Diese Methode konvertiert zuerst die in `scan` erkannten und innerhalb der Suchweite befindlichen Hindernisse in Weltkoordinaten. Anschließend wird durch einen Aufruf von `updateSplinePoints()` der noch zu fahrende Teil des Splines vom Roboter bis zur maximalen Suchweite in Punkte mit einem Abstand von jeweils `Configuration::getCollisionAvoidanceLookAheadInterval()` unterteilt. Nun überprüft die Methode `checkCollisions()`, ob die Abstände zwischen den in der Suchweite erkannten Hindernissen und den Spline-Punkten ausreichend groß sind:

Falls ja, geschieht nichts. Falls nein, fügt die Methode die problematischen Hindernisse der Karte (also genMap) hinzu und emittiert das Signal `alertPathBlocked()`, welches wiederum in `Mobiled::slotAlertPathBlocked()` dazu führt, dass die mobile Platfform ihre Bewegung abbricht und einen neuen Pfad planen lässt.

Die Berechnungen der Kollisionsvermeidung erfordern, dass die Distanz zwischen jedem der er-

rechneten Spline-Punkte mit jedem potentiellen Hinderniss errechnet und mit dem Roboterradius verglichen wird. Da jeder Laserscan bis zu $\frac{180°}{0.5°} = 360$ verschiedene Punkte enthält und der Spline in viele Unterpunkte unterteilt wird, kommt es bei jeder Überprüfung der verbleibenden Strecke zu sehr vielen Distanzberechnungen und -vergleichen. Um diesen sekündlich mehrmaligen Vorgang (teils auf Kosten der Genauigkeit bzw. „Weitsicht“) weniger rechenintensiv zu gestalten, wurden die beiden Parameter Suchweite und Abstand zwischen den Spline-Punkten eingeführt. Um das Ziehen einer Quadratwurzel bei jedem einzelnen Vergleich zu sparen, wird stattdessen der quadrierte Wert der Distanz verglichen.

Gerade für extrem kurze Fahrten ist es manchmal wünschenswert, die Kollisionsvermeidung gänzlich abzuschalten. Die Aktivierung der Kollisionsvermeidung ist nunmehr ein Parameter zu jedem Bewegungsbefehl im *mobiled*-Protokoll und läßt sich letztlich durch Aufruf der Methode `CollisionAvoidance::setActive(bool active)` mit entsprechendem Parameter erreichen. Um die Umwelt des Roboters auf das nun erhöhte Gefahrenpotential aufmerksam zu machen, emittiert die Kollisionsvermeidung bei ihrer Deaktivierung ein Signal, welches den `BeeperThread` veranlasst, regelmäßige Warntöne über den eingebauten PC-Lautsprecher auszugeben.

Von der Kollisionsvermeidung erkannte Hindernisse haben eine Lebensdauer definiert durch `Configuration::getCollisionAvoidanceObstacleLifetime()` ; nach deren Ablauf sie also aus der Karte entfernt werden. Dies ist notwendig, damit die Umgebung des Roboters nicht mehr und mehr von Hindernissen vergangener Zeit versperrt wird. Es gilt, diesen Wert geschickt zu wählen: Ein zu kurzer Wert könnte dazu führen, dass der Roboter immer wieder zwischen zwei versperrten Wegen hin- und her fährt. Ein zu langer Wert könnte die Plattform in Umgebungen mit vielen bewegten Objekten dauerhaft blockieren.

3.2.8 Dokumentation

Ein Ziel dieser Arbeit ist, *mobiled* erschöpfend zu dokumentieren, um eine spätere Weiterentwicklung auch durch andere Personen zu vereinfachen. Aus diesem Grunde befindet sich zusätzlich zu dieser Arbeit im *mobiled*-Unterverzeichnis `doc/html` eine vollständige Dokumentation aller benutzten Klassen, ihrer Eigenschaften und Methoden. Funktionsparameter sind zusammen mit ihren Einheiten und zulässigem Wertebereich dokumentiert. Dies ist schon deswegen nötig, weil viele Stellen im Code durch Zusicherungen (ASSERTs) gesichert sind. Auch Datenformate, z.B. für Protokollpakete finden in den relevanten Teilen des Quelltextes Erklärung. Weiterhin sind nicht nur die Deklarationen (also „Header“) beschrieben, auch in den Implementierungen werden die einzelnen Schritte erklärt.

Um ein möglichst schnelles Zurechtfinden in den Programmquellen von *mobiled* zu erleichtern, wurde bewusst auf komplizierte Entwurfsmuster verzichtet. Statt tiefer (Mehrfach-)Vererbung machen die meisten Klassen eher von der Delegation Gebrauch (GHJV95). Durch Verwendung von „Signals & Slots“ werden Funktionszeiger und Rückrufe (callbacks) überflüssig und das Programm besser lesbar.

Zur automatischen Dokumentation wurde das Programm *doxygen*[11] verwendet. Eine Aktualisierung ist durch einen Aufruf von `doxygen` im `doc`-Verzeichnis möglich. Hierbei werden auch

[11] `http://www.doxygen.org`, letzter Aufruf 2008-04-23

Diagramme von Klassen und Vererbungshierarchien erstellt.

4 Fahrtplanung und -steuerung

Die neue Version von *mobiled* soll neben reinen Rotationen und reinen Translationen der mobilen Plattform nun auch Bewegungen beherrschen, bei denen sowohl Rotation als auch Translation ungleich 0 sind; Der Roboter soll also Kurven fahren. Da die ersten beiden Bewegungsprofile schon vor dieser Arbeit implementiert und nur unwesentlich verändert worden sind, beschreibt dieses Kapitel die nötigen Verfahren, um einen mobilen Roboter zuverlässig einer ebenen Raumkurve entlang zu steuern.

Die hierfür nötigen Schritte lassen sich in zwei Phasen unterteilen: die erste Phase ist die der Fahrtplanung, in welcher eine Kurve im Raum konstruiert wird, auf dem sich die mobile Plattform später bewegen soll. Im Anschluss daran folgt die Phase der Fahrtsteuerung. Hier werden u.a. aus der gegebenen Kurve Signale zur Ansteuerung der Aktoren des Roboters erzeugt, um ihn während seiner Bewegung auf der vorgegebenen Bahn zu halten.

4.1 Fahrtplanung

Die Planung einer Fahrt beginnt in dem Moment, in dem *mobiled* über seine Netzwerschnittstelle einen Fahrauftrag erhält. Die gewünschte Zielpose wird von `NetworkServer::moveSpline()` durch Aufruf von `MobileD->moveSpline(const Pose target)` an *mobiled* übergeben. Hier wird das gewünschte Ziel gespeichert und via `MobileD::slotSendWayPointRequest(void)` weiter an die Fahrtplanung übergeben, welche nun nach einem Weg von dem aktuellen Standpunkt des Roboters zu der Zielpose sucht.

4.1.1 Wegsuche

Die Suche eines Weges zwischen zwei gegebenen Punkten im Raum ist an sich ein eigenständiges Forschungsgebiet. Einen Weg bei möglichem Vorhandensein von Hindernissen zu ermitteln, ist laut (LaV06, Kapitel 6) sogar ein NP-vollständiges Problem.

mobiled nutzt das in der genRob-Architektur vorhandene genPath, welches wiederum auf eine Implementierung des A*-Algorithmus (HNR68) zurückgreift. Sofern vom Fahrtplanungsmodul (vgl. Kapitel 3.2.6.3) ein Weg gefunden werden kann, wird dieser je nach seiner Länge und Komplexität aus mehreren Wegpunkten bestehend an *mobiled* zurückgeliefert. Liegen die Wegpunktdaten nach Übermittlung via JNI in der `Java`-Klasse vor, meldet sie dies durch das Signal `wayPointsReady(QList<QPointF> wayPoints)`. Dieses wird von der Methode `MobileD::slotNewWayPointsArrived(QList<QPointF> wayPoints)` aufgefangen. Nun erfolgt durch Interpolation dieser Punkte mit Hilfe von Splines die Erstellung einer Raumkurve, die dann zu Beginn der nächsten Phase der Fahrtsteuerung übergeben wird. Diese Raumkurven werden in den nun folgenden Unterkapiteln thematisiert.

4.1.2 Splines

Ein Spline ist eine Funktion, die ihrerseits aus mehreren stückweise zusammengesetzten Polynomen besteht. Sind die eingesetzten Polynome vom Grad n, so spricht man von einem Spline n-ten Grades. Die eingesetzten Polynome werden meist über zwei (manchmal auch vier) sogenannte Stützpunkte definiert, so dass ein Spline mit k Stützpunkten $k-1$ Polynome verwendet und auch aus $k-1$ Segmenten besteht. Abbildung 4.1 zeigt einen Spline, erstellt aus den fünf Stützpunkten $p0$, $p1$, $p2$, $p3$, $p4$ mit den vier Segmenten $s0$, $s1$, $s2$, $s3$, welche durch vier Polynome erzeugt werden.

Verwendung finden Splines bei der Interpolation bzw. Approximation von diskreten Daten, z.B. Messwerten. Häufig ist hier gewünscht, einen nicht gemessenen Wert zu erhalten, der aber zwischen zwei bekannten Werten liegt. Mit Hilfe der Spline-Interpolation ist es nun möglich, einen weichen (und somit meist auch für die Bedeutung der Daten schlüssigen) Verlauf zwischen diesen Werten zu generieren.

Die Bewegungssteuerung einer mobilen Plattform bringt eine prinzipiell ähnliche Problemstellung mti sich: Die Fahrtplanung generiert aus einem gewünschten Zielpunkt eine Menge von Daten bzw. Wegpunkten. Zwischen diesen soll eine Kurve einen möglichst weichen Weg weisen, welcher anschließend vom Roboter abgefahren wird.

4.1.3 Krümmung

Die Krümmung einer Kurve ist definiert als die Richtungsänderung der Kurve in einem Abschnitt a dividiert durch die Länge von a.

$$K = \frac{\text{Richtungsänderung in a}}{\text{Länge von a}} \tag{4.1}$$

Bei Splines, also parametrisch notierten Kurven, kann die Krümmung an jeder beliebigen Stelle der Kurve (symbolisiert durch den Parameter t im Wertebereich $0 <= t <= 1$) mit Hilfe folgender Formel aus (Hen98) errechnet werden:

$$K = \frac{\dot{x}(t) * \ddot{y}(t) - \ddot{x}(t) * \dot{y}(t)}{(\dot{x}(t)^2 + \dot{y}(t)^2)^{3/2}} \tag{4.2}$$

Hierbei bedeutet $\dot{x}(t)$ die erste Ableitung der Funktion zur Errechnung des X-Wertes aus t und $\ddot{x}(t)$ die zweite Ableitung. Analog gilt dies natürlich auch für $\dot{y}(t)$ und $\ddot{y}(t)$. Ein negatives Ergebnis bedeutet hierbei eine Krümmung nach links bzw. eine Drehung im mathematisch positiven Sinne, positive Werte entsprechen Drehungen in mathematisch negativer Richtung.

Aus dem Krümmungswert lässt sich durch eine einfache Umrechnung eine für die Fahrtsteuerung sehr viel interessantere Größe errechnen: der Kurvenradius r ist definiert als der Kehrwert der Krümmung:

$$r = \frac{1}{K} \tag{4.3}$$

Wie aus Formel 4.2 ersichtlich, ist die Verfügbarkeit einer zweiten Ableitung Voraussetzung zur Ermittlung der Krümmung einer Kurve. Nicht jede Splinefunktion zur Ermittlung der zu einem t zugehörigen Wert lässt sich aber zwei Mal ableiten: kubischer spline.

4.1.4 Stetigkeit

Verschiedene Splines erfüllen unterschiedliche Anforderungen in Bezug auf ihre Stetigkeit. Während die Stetigkeitsbedingungen für die gesamte Splinekurve gelten, sind diese innerhalb der einzelnen Splinepolynome meist gegeben; somit verbleibt nur die Frage nach der Stetigkeit an den Übergängen vom Endpunkt a eines Segments zum Anfangspunkt b des folgendes Segments.

Eine Kurve ist C^k stetig genau dann, wenn alle k Ableitungen der Kurve stetig sind.

- C^0-Stetigkeit: Einen Spline nennt man C^0-stetig, wenn alle Übergänge zwischen den Segmenten C^0-stetig sind. Ein Segmentübergang ist genau dann C^0-stetig, wenn dort die beiden Endpunkte der Kurvenstücke zusammenfallen, wenn dort also kein Sprung vorliegt, d.h.:

  $$S(a) = S(b) \tag{4.4}$$

  Wäre eine Kurve nicht C^0-stetig, könnte der Roboter ihr nicht folgen, ohne mit Zwischenschritten vom Ende eines Segments zum Anfang des nächsten zu fahren. Von einer natürlichen Bewegung wäre dieses Verhalten weit entfernt.

- Ein Spline ist C^1-stetig, wenn alle Segmentübergänge C^0-stetig sind und an allen Übergängen End- und Startrichtung übereinstimmen. Bildlich gesprochen darf am Übergang kein Knick vorliegen, mathematisch gesehen müssen die Werte der ersten Ableitung übereinstimmen:

  $$\dot{S}(a) = \dot{S}(b) \tag{4.5}$$

  Bei einer C^0-, aber nicht C^1-stetigen Kurve müsste die mobile Plattform an den Übergängen der Polynome ihre Orientierung ändern. Aufgrund ihrer Masse und ihrer damit einhergehenden Trägheit würde resultieren, dass die Fahrt an jedem Übergang gestoppt werden müsste. Anschließend könnte die Ausrichtung dem neuen Splinesegement angepasst werden und der Roboter könnte dem Polynom bis zu seinem Ende folgen.

- C^2-stetig ist ein Spline dann, wenn die Krümmung der Kurve über alle Segmentübergänge gleich ist. Für den Roboter bedeutet dies, dass er bei den Segmentübergängen die Radgeschwindigkeiten nicht sprunghaft zu verändern braucht. Auch dies wäre wegen der Trägheit des Roboters nicht möglich und würde so zu Abweichungen vom geplanten Pfad führen. Mathematisch gesehen bedeutet dies, dass sich die Werte der zweiten Ableitungen direkt vor und nach jedem Übergang gleichen müssen.

  $$\ddot{S}(a) = \ddot{S}(b) \tag{4.6}$$

Für eine natürlich anmutende Robotersteuerung ist also die Verwendung eines mindestens C^1-stetigen Splines erforderlich. Sind die Sollgeschwindigkeiten der Antriebsräder eine Funktion der momentan vorliegenden Splinekrümmung, so ist die Verwendung eines C^2-stetigen Splines Voraussetzung, da die Gleichung 4.2 die zweite Ableitung des Splines benötigt.

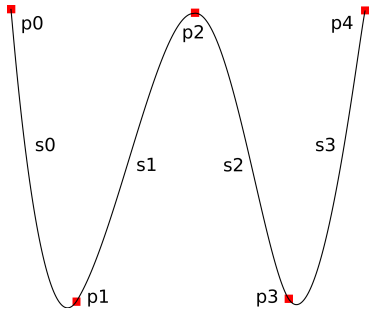


Abbildung 4.1: Stützpunkte und Kurvensegmente eines Splines

4.1.5 Interpolierende und approximierende Splines

Für Splines jeglicher Art existiert eine Dichotomie, nach welcher sich eben jene in entweder interpolierende oder approximierende Splines unterteilen lassen. Diese Klassifizierung bezieht sich auf das Verhalten der resultierenden Kurve in Bezug auf ihre Stützpunkte:

Interpolierende Splines sind - wie der Name schon sagt - geeignet, um fehlende Daten zwischen verschiedenen Datenpunkten zu interpolieren. Das heißt, die entstehende Kurve läuft immer durch die gegebenen Stützpunkte hindurch. Beispiele für interpolierende Splines sind (b-e) in Abbildung 4.2.

Approximierende Splines verwenden ihre Stützpunkte lediglich zur Orientierung der entstehenden Kurve. Es ist zwar möglich, den Kurvenverlauf durch Verschieben der Stützpunkte zu beeinflussen; die genauen Punkte, die diese Kurve dann durchlaufen wird, sind aber nur durch „Ausrechnen“ des Splines zu ermitteln. Der Spline berührt seine Stützpunkte also nicht, er approximiert sie lediglich. Beispiele für approximierende Splines sind in Abbildung 4.2 (f-h) gegeben.

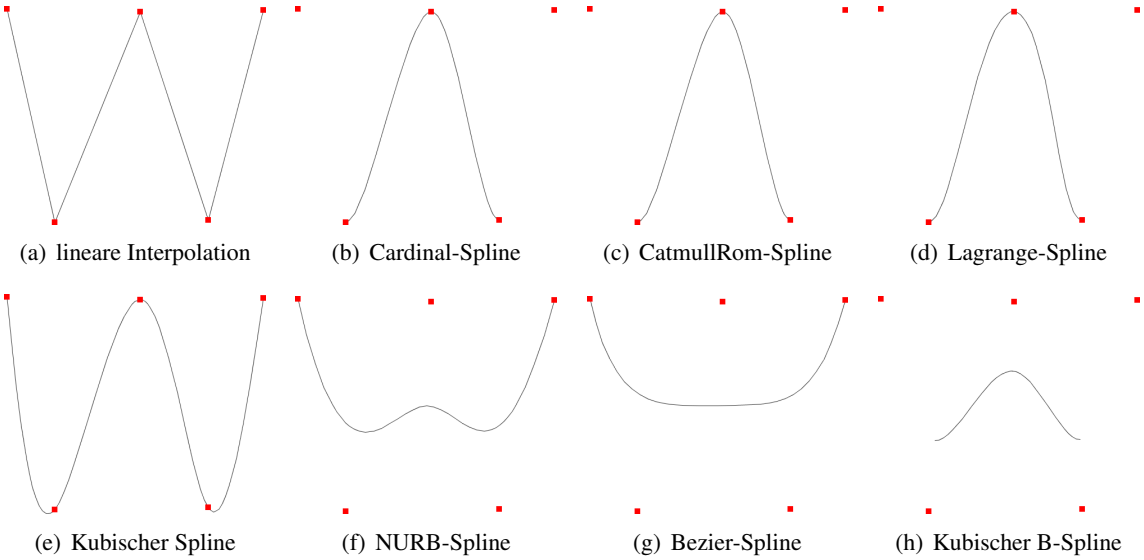
Um den Aufwand der Fahrtplanung in Grenzen zu halten, wird vorrausgesetzt, dass die mobile Plattform während ihrer Fahrt zur Vermeidung von Kollisionen möglichst durch die ermittelten Punkte fährt. Weiterhin ist es natürlich wichtig, dass die ermittelte Kurve auch tatsächlich beim momentanen Standort des Roboters startet und ihr Ende beim letzten gegebenen Stützpunkt findet. Für die Steuerung der Bewegung eines mobilen Systems *durch* eine Menge von gegebenen Punkten sind approximierende Splines somit nicht brauchbar und wurden deswegen in dieser Arbeit nicht weiter untersucht.

4.1.6 Erster und letzter Stützpunkt

Wird ein Spline durch $n + 1$ Stüzpunkte definiert, muss der Algorithmus stets n zwischenliegende Kurvensegmente berechnen. Viele Spline-Algorithmen benötigen jedoch zur Berechnung des Kurvensegments n nicht nur die Stützpunkte p_n und p_{n+1} (die das Segment beschränken), sondern auch die beiden Stüzpunkte p_{n-1} und p_{n+2}, also den Vorgänger des linken und den Nachfolger des rechten Stützpunktes.

Daraus folgt, dass bei einigen Splines in Hinsicht auf die gegebenen Stützpunkte eine unvollständige Kurve entsteht (siehe Abbildung 4.2 (b,c,d,h)). Diesem Problem kann begegnet werden,

indem die fehlenden Stüzpunkte p_{-1} und p_{n+1} nachträglich aus den gegebenen Stüzpunkten generiert werden. Dabei ist allerdings zu beachten, dass die Position dieser generierten Stützpunkte den Verlauf der Kurve durchaus beeinflusst.



(a) lineare Interpolation (b) Cardinal-Spline (c) CatmullRom-Spline (d) Lagrange-Spline

(e) Kubischer Spline (f) NURB-Spline (g) Bezier-Spline (h) Kubischer B-Spline

Abbildung 4.2: Verschiedene Splines über einer Menge von 5 Stützpunkten

Zum Einsatz in der Fahrtplanung empfehlen sich interpolierende Splines, weil so sichergestellt ist, dass alle von der Fahrtplanung ermittelten Punkte tatsächlich von der mobilen Plattform durchfahren werden. Ferner hält sich die Kurve eines interpolierenden Splines näher am vom Fahrtplaner erstellten Pfad, was wiederum die Wahrscheinlichkeit minimiert, dass der generierte Pfad mit Hindernissen kollidiert.

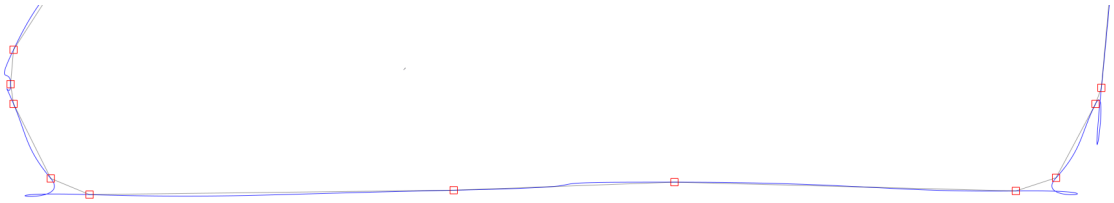
4.1.7 Überschwingungen

In der Fahrtplanung wird - genau wie vor Beginn dieser Arbeit - weiterhin eine Folge von Wegpunkten errechnet. Von Seiten dieses Algorithmus ist nun garantiert, dass die mobile Plattform mit keinem auf der Karte eingetragenen Hinderniss kollidiert, solange sie sich auf dem linear interpolierten Pfad zwischen diesen Punkten bewegt.

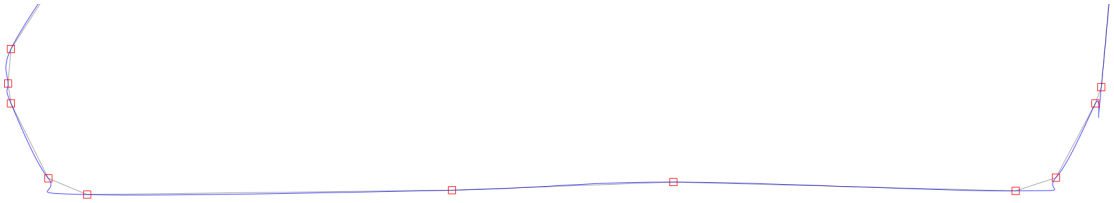
Splines jedoch interpolieren nicht linear. Daraus folgt, dass der Roboter diesen vom Fahrtplaner vorgegebenen Pfad verlassen wird und somit eine kollisionsfreie Fahrt nicht länger garantiert ist. In Abbildung 4.3 ist zu sehen, wie verschiedene Splines gerade in Gegenden mit vergleichsweise hoher Wegpunktdichte zum Überschwingen neigen. Problematisch daran ist nicht nur, dass die mobile Plattform bei größeren Überschwingungen ein größeres Kollisionsrisiko eingeht. Die bei starken Überschwingungen entstehenden Schleifen können auch zur einer sehr unnatürlichen und für den Betrachter unverständlichen Fahrweise führen.

Kollisionen durch Überschwingen des verwendeten Splines können minimiert werden, indem:

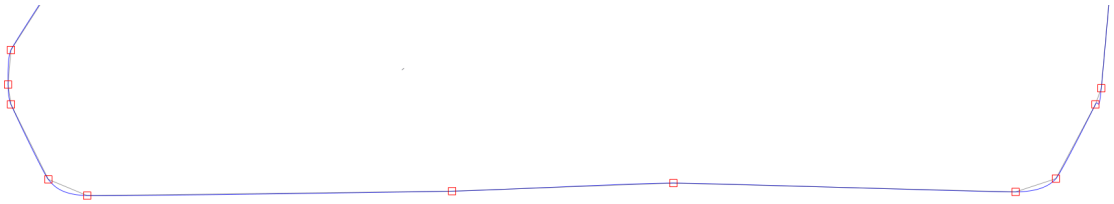
- bei Verwendung von hermitischen (bzw. Kochanek-Bartels- oder Catmull-Rom-) Splines der Spannungsparameter so lange erhöht wird, bis ein Kollisiontest negativ ist. Die Erhö-



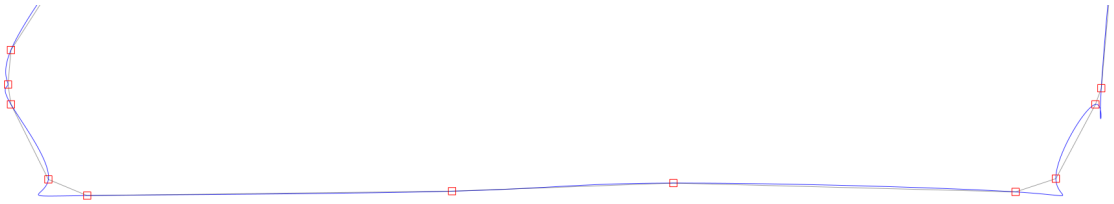
(a) Hermitischer Spline mit einer Spannung von -1.0



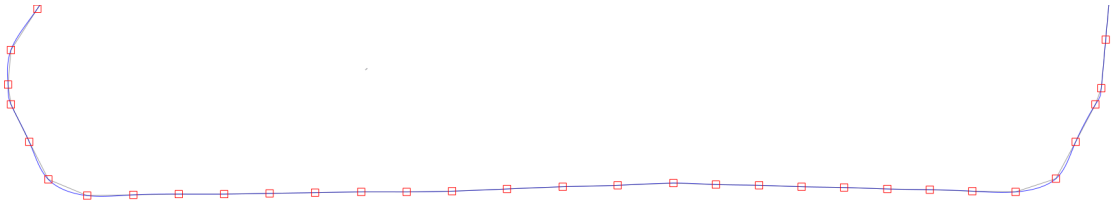
(b) Hermitischer Spline mit einer Spannung von 0.0



(c) Hermitischer Spline mit einer Spannung von 0.6



(d) Kubischer Spline



(e) Kubischer Spline, optimiert durch Einfügen von Zwischenwegpunkten

Abbildung 4.3: Überschwingungen verschiedener Splines, graue Linien entsprechen einer linearen Interpolation der Wegpunkte

hung der Spline-Spannung hat auch den Vorteil, dass unnötige Spline-Schleifen hierbei minimiert werden. Die Auswirkungen verschiedener Spannungswerte sind in den Abbildungen 4.3a, 4.3b und 4.3c zu erkennen. Schwierig ist hierbei die Wahl eines probaten Spannungswertes, da dieser abhängig von den vorliegenden Stützpunkten durchaus unterschiedliche Werte annehmen kann. Ein zu kleiner Wert führt zu unnötigen Überschwingungen und Schleifen, ein zu groß gewählter Wert führt zu einer wenig weichen, unnatürlichen Fahrt.

- an die Fahrtplanung anschließend der Spline erstellt und bei positivem Kollisionstest die Stützpunkte so lange verändert werden, bis der Kollisionstest negative Ergebnisse liefert. Hier kann allerdings selbst komplizierte Logik nicht in einer kurzen Zeit garantiert kollisionsfreie Pfade konstruieren.

- das Auftreten von dicht aufeinander folgenden Stützpunkten durch Einfügen von Zwischenwegpunkten relativiert wird. Ergebnisse dieses Verfahrens sind in Abbildung 4.3e dargestellt.

Aus Gründen, die später im Kapitel 4.2 erläutert werden, verwendet *mobiled* kubische Splines, deren Überschwingungen durch Einfügen von Zwischenpunkten minimiert werden. Zuständig für diese Optimierung ist die Methode `Java::optimizeWayPoints(QList<QPointF> &points)`. Sie fügt zwischen zwei Wegpunkten a und c immer dann einen Zwischenwegpunkt b ein, wenn der Abstand $\overline{ac}$ größer als der von `Configuration::getPathOptimizationInterval()` zurückgelieferte Wert ist.

4.1.8 Splinelängen

Wie alle Strecken haben auch Splines eine Länge. Für die Steuerung der Fahrt eines mobilen Roboters ist es wichtig, diese Länge zu kennen. Hatten die beiden Antriebsräder der mobilen Plattform beim Abfahren eines Splines innerhalb eines kurzen Zeitabschnitts $\Delta t = t_2 - t_1$ die Geschwindigkeiten v_l und v_r, so beträgt die vom Plattformmittelpunkt innerhalb Δt zurückgelegte Distanz d

$$d = \frac{(v_l * t) + (v_r * t)}{2} \tag{4.7}$$

Nachdem Δt verstrichen ist, muss die Fahrtsteuerung in Erfahrung bringen, wo genau auf dem Spline sich der Roboter befindet. Dazu ist es notwendig zu wissen, um welchen Wert der Spline-Parameter u (normalerweise als t deklariert, hier zum Zwecke der Eindeutigkeit umbenannt) inkrementiert werden muss. Da sich u zum Zeitpunkt t_2 aus seinem Wert zum Zeitpunkt t_1, der Länge des Splines l und d errechnet [1], muss l bekannt sein.

[1] Näheres zum Zusammenhang zwischen u und der zugeordneten Position auf dem Spline ist in Kapitel 4.1.9 dokumentiert.

Diese zu berechnen, gestaltet sich leider sehr viel schwieriger als auf den ersten Blick vermutet: Es wurde bislang kein Verfahren gefunden, das es ermöglichen würde, die Länge einer jeden Splinekurve exakt zu berechnen[2]. Statt dessen behilft man sich mit meist einfachen, dafür langsamen und ungenauen Verfahren.

Eine Möglichkeit ist, den Spline in n Intervallen abzuschreiten und jeweils den euklidischen Abstand zwischen den ermittelten Punkten zu summieren. Für eine höhere gewünschte Präzision kann der Parameter n auf Kosten der Rechenzeit beliebig vergrößert werden. Als Länge l eines zweidimensionalen Splines ergibt sich damit:

$$dist(p_1, p_2) = \sqrt{(p_{2x} - p_{1x})^2 + (p_{2y} - p_{1y})^2} \tag{4.8}$$

$$l = \sum_{x=n}^{1} dist(S(x * \frac{1}{n}), S((x-1) * \frac{1}{n})) \tag{4.9}$$

Dieser Algorithmus liegt offensichtlich in der Komplexitätsklasse $O(n)$, was sich für die Zwecke dieser Arbeit als völlig ausreichend erwiesen hat.

Der Vollständigkeit halber soll noch eine stark optimierte Variante eines weiteren iterativen Verfahrens aus (VF01)[3] aufgeführt werden: Hierbei werden Abschnitte der zu untersuchenden Kurve nur durch Bestimmung einiger Splinepunkte zu Kreisausschnitten zusammengefasst. Regionen des Splines mit höherer Krümmung werden erkannt und rekursiv in kleinere Kreisausschnitte unterteilt. So passt sich der Algorithmus der „Komplexität“ der Kurve an und kann für weniger stark gegrümmte Kurven schneller (aber nicht weniger genau) arbeiten. Insgesamt ist dies zwar etwas langsamer als die einfache Summation der gleichen Menge von Teilabschnitten, führt aber zu deutlich genaueren Ergebnissen.

Vorraussetzung zur Verwendung dieses Algorithmus' ist - wie auch beim ersten vorgestellten Verfahren - lediglich die Möglichkeit, einzelne Splinepunkte aus einem Parameter t zu errechnen; der Einsatz einer Methode zum Ermitteln einer ersten oder zweiten Ableitung ist nicht erforderlich. Somit sind beide Verfahren bei allen bekannten Splines anwendbar.

4.1.9 Nichtlinearität des Kurvenparameters t

Spline-Algorithmen gleichen sich insofern, als dass stets aus einem gegebenen Spline-Parameter (meist durch t beschrieben), dessen Wert zwischen 0 und 1 liegt, ein je nach Dimensionalität n des Splines n-dimensionaler Vektor generiert wird. Bei einem zweidimensionalen Spline, wie er in dieser Arbeit Verwendung findet, erhält man also durch Iteration des Parameters t von 0 nach 1 alle auf der Kurve liegenden Punkte.

[2] Ein beispielhafter Versuch, dieses Problem zumindest für Bezier-Splines mit Hilfe elliptischer Intergrale zu lösen, ist unter http://www.tinaja.com/glib/bezlenjf.pdf dokumentiert.

[3] Der Quelltext einer Implementierung dieses Verfahrens ist unter http://jgt.akpeters.com/papers/VincentForsey01/CurveLength.html einsehbar.

t	0.0000	0.0001	0.0002	...	0.9997	0.9998	0.9999	1.0000
s(t)	0.0000	0.0153	0.0228	...	16.281	16.457	16.764	16.913

Tabelle 4.1: Beispielhafte Zuordnung von t zu $s(t)$

Unglücklicherweise verhält sich t nicht linear zur Position des ihm zugeordneten Punktes auf der Kurvenstrecke. In Abbildung 4.4[4] ist beispielhaft ein Spline aufgetragen; zu erkennen ist, daß gleiche numerische Abstände beim Kurvenparameter t (in weiß dargestellt) nicht gleiche Längen der Zwischenstrecken der jeweils zugeordneten Punkte (schwarze Pfeile) auf der Kurve bedeuten. Generell führt eine Inkrementierung des Kurvenparameters t in stärker gekrümmten Bereichen zu weniger Fortschritt auf der Kurvenstrecke.

Die Sollgeschwindigkeit der Antriebsräder des Roboters wird aus der aktuellen Kurvenkrümmung errechnet. Am Beginn jeder Iteration des Steuerungsalgorithmus werden die einzelnen Radfortschritte des mobilen Systems seit dem letzten Schritt ermittelt und in eine Gesamttranslation verrechnet (siehe Gleichung 4.1.8). Nun wird näherungsweise davon ausgegangen, dass sich das System genau auf dem vorliegenden Spline bewegt hat, so dass zur erneuten Ermittlung der Kurvenkrümmung zu dem der aktuellen Position des Roboters auf der Kurve entsprechende Kurvenparameter t_2 der alte Kurvenparameter t_1 mit einem Δt entsprechend der zwischenzeitlich vom Roboter zurückgelegten Distanz d addiert werden muss. Diese Distanz liegt aber nicht als Differenz Δt vor, sondern als Strecke in Metern.

$$t_2 = t_1 + \Delta t \tag{4.10}$$

Würde t sich linear zur Länge l_S des Splines entwickeln, wäre die Umrechnung trivial:

$$\Delta t = \frac{1}{l_S} * d \tag{4.11}$$

Um nun trotz des nichtlinearen Verhaltens von t aus einer zurückgelegten Strecke d von einem beliebigen t_1 zum korrekten t_2 zu gelangen, ist eine Zuordnungstabelle notwendig. In dieser werden alle möglichen Werte von t (also im Intervall $[0, 1]$) auf die entsprechenden Strecke vom Beginn der Kurve bis zum durch t repräsentierten Punkt $S(t)$ abgebildet. Der Aufbau dieser Zuordnungstabelle geschieht aus Gründen der Effizienz zeitgleich mit der Ermittlung der Gesamtlänge des Splines, wie in Kapitel 4.1.8 beschrieben. Ergebnis ist nun eine Verknüpfung wie in Tabelle 4.1.

Diese Tabelle kann nun je nach gewünschter Präzision aus beliebig vielen Zuordnungen bestehen. In Tests hat sich eine Anzahl von 10.000 Abbildungen als guter Kompromiss zwischen Rechenzeit und resultierender Bahntreue erwiesen. Da dieser Wert durch Aufruf von `Configuration::getSplineMappingSampleCount()` definiert wird, kann er durch Anpassung der Konfiguration `motion/splineMappingSampleCount` modifiziert werden.

Mit Hilfe dieser Zuordnung kann nun also nach Ermittlung des Roboterfortschritts seit der letzten Iteration das gewünschte t_2 bestimmt werden. In Folge dessen kann die bei diesem t vorliegende

[4] Zeichnung entnommen aus http://de.wikipedia.org/wiki/Computeranimation, letzter Aufruf 2008-05-07

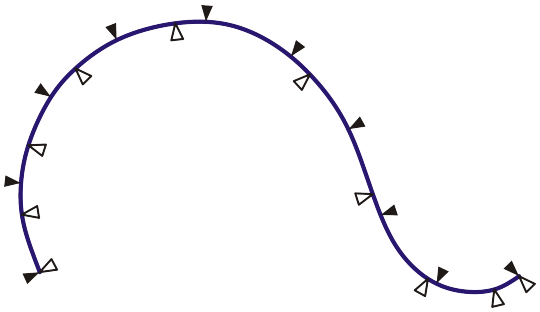


Abbildung 4.4: nichtlineares Verhalten des Kurvenparameters t

Krümmung bestimmt und zur erneuten Berechnung der Sollgeschwindigkeiten beider Antriebsräder herangezogen werden.

Da der Parameter t letztlich nichts anderes als eine zwar notwendige aber unpraktische Alternative zur Angabe der Splineposition in Metern ist, wurde seine Verwendung innerhalb *mobiled* weitestgehend beschränkt: t findet allein innerhalb der *Spline*-Klasse Verwendung, außerhalb dieser wird mit Distanzen in Metern gearbeitet. Der Algorithmus zur Umrechnung einer Distanz in das jeweilige t findet sich in der Datei `spline.cpp`, Methode `Spline::distanceToT(float distance)`. Zusätzlich zum hier beschriebenen Verhalten wurde der Code aufgrund seiner häufig wiederholten Ausführung zur Optimierung seines Laufzeitverhaltens mit Hilfe einer Art von Zwischenspeicher enorm beschleunigt: So lange die angefragten (und somit in ein t umzurechnenden) Distanzen zwischen den Aufrufen der Methode monoton steigen, wird die die Abbildungen enthaltende Datenstruktur nicht jedes mal von vorne durchlaufen; die Suche nach einem passenden t wird dort fortgesetzt, wo die letzte Suche nach dem vorigen t endete. Dies bedeutet bei einer angemessenen Gesamtanzahl von Abbildungen pro Methodenaufruf nur sehr wenige Iterationen.

4.2 Fahrtsteuerung

Bei der Steuerung eines mobilen Roboters entlang einer Kurve unterscheidet man zwischen zwei verschiedenen Verfahren:

- Mit „path following“ bezeichnet man ein Verfahren, bei dem ein System einem vorgegebenen Pfad folgt.
- Man spricht von „trajectory tracking“, wenn das mobile System einem Pfad nicht nur folgen soll, sondern sich auch zu jedem Zeitpunkt an einer a priori bestimmten Stelle auf diesem Pfad befinden soll. Dieses Verfahren ist also mit zeitlichen Bedingungen verknüpft.

Wie (AHK08) und (EHS98) zeigen, ist letztere Variante sehr viel schwieriger und nur unter bestimmten Bedingungen umzusetzen. Ferner führt die dann erforderliche Rücksichtnahme auf die Beschränkungen des realen Roboters zu einem ungleich höheren Aufwand bei der Fahrtplanung. Glücklicherweise ist es bei Fahrten des Roboters von einer Start- zu einer Zielkonfiguration nebensächlich, zu welchem Zeitpunkt er sich auf einem Punkt auf der Kurve zwischen diesen Konfigurationen befindet. Somit kann sowohl *mobiled* als auch dieses Kapitel diese Probleme umgehen.

4.2.1 Ermittlung der Rad-Sollgeschwindigkeiten

Um einem Spline zu folgen, muss der Roboter seine Radgeschwindigkeiten dem aktuellen Verlauf der Kurve anpassen. Nach Kapitel 4.1.3 sind die korrekten Radgeschwindigkeiten also eine Funktion des aktuellen Kurvenradius des Splines, welcher wiederum dem Kehrwert der Krümmung entspricht. Diese wird mit der ebenfalls in Kapitel 4.1.3 vorgestellten Formel 4.2 errechnet. Sie benötigt sowohl die erste als auch die zweite Ableitung des Splines an der jeweiligen Stelle. Da hermitische Splines nicht zweimal ableitbar bzw. C^2-stetig sind, kann ihr Kurvenradius also höchstens numerisch bestimmt werden.

Aus diesem Grunde verwendet die Bewegungssteuerung im weiteren Verlauf der Arbeit vorerst kubische Splines, deren Überschwingungen nach dem in Kapitel 4.1.7 vorgestellten Verfahren minimiert werden.

Die Bestimmung der beiden Radgeschwindigkeiten eines Roboters, der sich d Meter am Spline entlang bewegt hat, geschieht folgendermaßen: Zuerst wird durch Aufruf von `double Spline::getCurvature(d)` mit Hilfe von Formel 4.2 die Krümmung am entsprechenden Punkt des Splines berechnet. Dieser Wert wird der Methode `Conversion::curvature2wheelSpeeds(const double curvature, float &speedL, float &speedR)` übergeben. Sie berechnet zuerst mit einer einfachen Division den Kurvenradius aus dem ersten Parameter. Aus diesem lässt sich in Abhängigkeit der Achslänge das Verhältnis der Radgeschwindigkeiten ermitteln. Ein Sonderfall tritt dann ein, wenn der Kurvenradius kleiner als die Achslänge ist, dann nämlich muss ein Rad rückwärts drehen, um dem Spline weiterhin folgen zu können. Nach Ermittlung des Verhältnisses der Radgeschwindigkeiten werden die konkreten Geschwindigkeitswerte so festgelegt, dass das jeweils schnellere Rad mit der maximal erlaubten Geschwinigkeit dreht. Der vollständig dokumentierte Quelltext der Methode ist in der Datei `conversion.cpp` zu finden.

Nun werden die beiden Geschwindigkeiten `speedL` und `speedR` jeweils an den entsprechenden Motor geschickt.

4.2.2 Kurskorrektur

In der Theorie sollte das eben beschriebene Verfahren perfekte Ergebnisse liefern. In der Praxis führen allerdings Trägheit, Reibung, Unebenheiten im Boden, geringfügig unterschiedliche Reifenumfänge und viele andere Faktoren zu kleinen Abweichungen der Bewegung des Roboters relativ zum geplanten Pfad. Werden diese Fehler nicht korrigiert, führen sie im Verlauf der Fahrt zu immer größeren Abweichungen, was die Gefahr von Kollisionen erhöht und die Wahrscheinlichkeit eines Fahrtendes am gewünschten Zielpunkt senkt.

Deswegen vergleicht *mobiled* während der Fahrt ständig die von der Lokalisation ermittelte Ist-Pose des Roboters mit der vom Spline vorgegebenen Soll-Pose. Abbildung 4.5 zeigt einen mobilen Roboter, dessen Mittelpunkt sich von der geplanten Bahn entfernt hat. In diesem Fall muss die Position der Plattform durch Anpassung der Rotationsgeschwindigkeit korrigiert werden.

In *mobiled* findet momentan ein einfaches Verfahren Verwendung: In jeder Steuerungsiteration werden Soll- und Ist-Position verglichen. Hierbei wird der Winkel zwischen der Orientierung des Roboters zur Soll-Position ermittelt. Liegt dieser beispielsweise zwischen $0°$ und $90°$ (also in

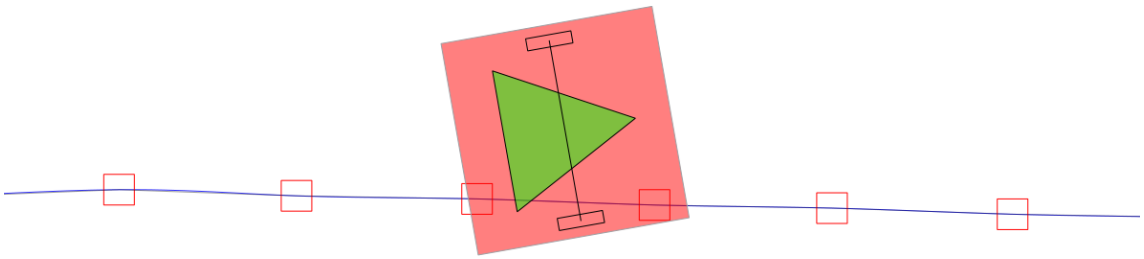


Abbildung 4.5: Kurskorrekturen sind notwendig, um eine von der vorgegebenen Bahn abweichende mobile Plattform auf diese zurückzuführen

Quadrant 2 von Abbildung 4.6), so muss der Roboter das rechte Rad beschleunigen, um sich über mehrere Iterationen dem Sollzustand zu nähern. Zusätzlich wirkt die Entfernung zwischen Roboter und Sollposition als Faktor auf die Veränderung der Radgeschwindigkeit ein; größere Entfernungen haben also größere Geschwindigkeitskorrekturen zur Folge.

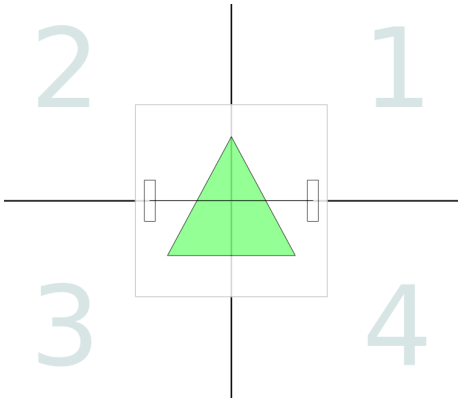


Abbildung 4.6: Das Verhalten der Kurskorrektur ist abhängig von der Soll-Position relativ zur Position des Roboters

Obwohl diese Methode sehr simpel ist, führt ihr Einsatz zu akzeptablen Ergebnissen. Experimentelle Werte werden in Kapitel 5.4 präsentiert.

5 Experimente

5.1 Echtzeitfähigkeit

Wie in Kapitel 3.2.7.5 schon angesprochen wurde, benötigt die Robotersteuerung zur sicheren und weichen Bewegung des Roboters eine zuverlässige Grundlage: Sie muss in regelmäßigen Abständen aufgerufen werden und innerhalb eines festen Intervalls abgeschlossen sein. Um diese Tatsachen experimentell nachzuweisen, wurden während einer Kurvenfahrt von *mobiled* (diese Fahrt enthält auch reine Rotationen auf ihrem Start- und Endpunkt) Zeitmessungen durchgeführt.

Der Graph 5.1 trägt die Verteilung der zeitlichen Iterationsdauer einer Fahrt mit 2697 Iterationen auf. Zu erkennen ist, dass die Abarbeitung eines Steuerungsdurchlaufs zwischen 50 und 180 Mikrosekunden dauert. Ferner zeichnen sich im Graphen zwei Spitzen ab: die erste liegt bei einer Iterationsdauer von ca. 80, die zweite bei ca. 118 Mikrosekunden. Ursache zweier Maxima ist der Umstand, dass die Steuerung zwei verschiedene Programmpfade abläuft: der Code, der den Roboter am Anfang und Ende einer Fahrt um die eigene Achse rotieren lässt, benötigt durchschnittliche 120 Mikrosekunden. Der Programmteil zum Steuern des Roboters auf dem Spline ist dagegen in 80 Mikrosekunden abgearbeitet.

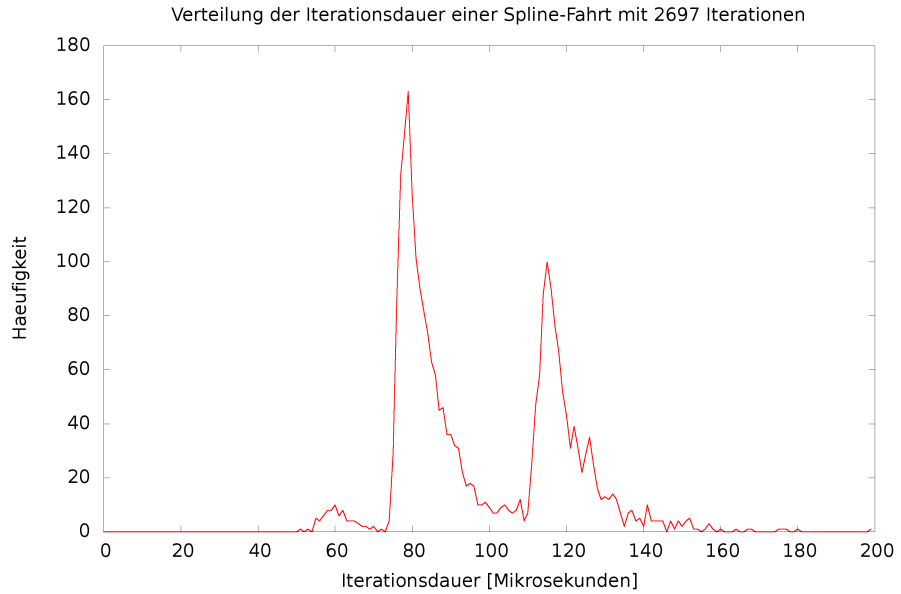


Abbildung 5.1: Häufigkeitsverteilung der Dauer von Steuerungsiterationen - Ein Steuerungsablauf nimmt maximal 180 Mikrosekunden Zeit in Anspruch.

Pessimistisch betrachtet kann also angenommen werden, dass ein Steuerungsdurchlauf in einer Zeitspanne von 180 Mikrosekunden abgeschlossen ist. Somit liegt die obere Schranke der Aufruf-

frequenz bei

$$\frac{1000000\mu s}{180\mu s} \approx 5556\text{Hz}. \tag{5.1}$$

Natürlich ist dies ein theoretischer Wert, in der Praxis würden sich schon bei kleineren Frequenzen Probleme ergeben. Zu bedenken ist auch, dass bei einem verteilten Einsatz von *mobiled* die Übermittlung von 5556 Paketen zum Setzen der Radgeschwindigkeit plus 5556 Paketen zum Auslesen der Radfortschritte sowie noch einmal $2 * 37,5 = 75$ UDP-Paketen von den Laserscannern (BP06, Kapitel 6.1), also insgesamt 11187 Paketen in jeder Sekunde eine unmöglich zu leistende Anforderung an die IEEE802.11b-Netzwerkverbindung der mobilen Plattform darstellen würde.

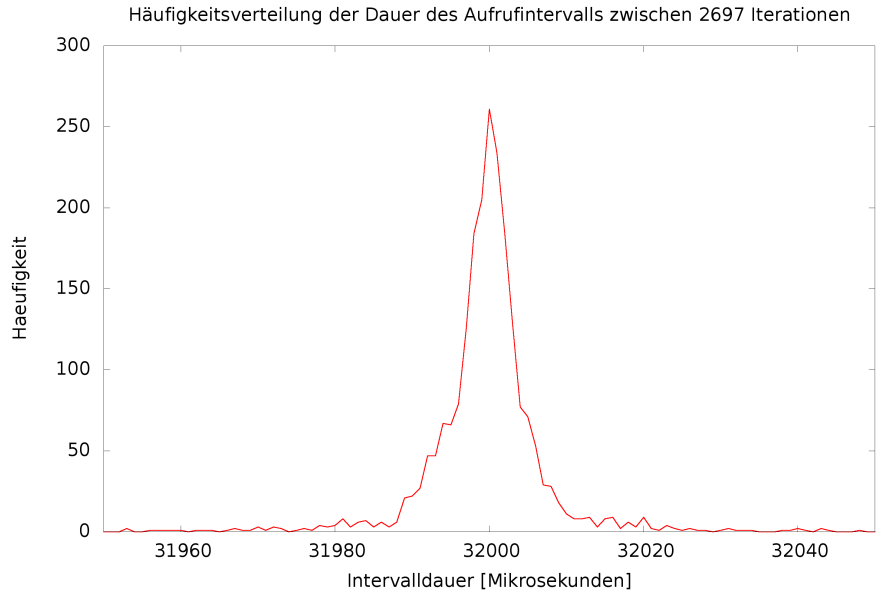


Abbildung 5.2: Häufigkeitsverteilung der Dauer des Aufrufintervalls. Bei Beachtung der Abszissenskalierung wird die zeitliche Präzision des *QTimer*s deutlich.

Die zweite Anforderung an die Echtzeitfähigkeit ist, dass die Steuerungsfunktionen nicht nur innerhalb einer vorgegebenen Zeit abgeschlossen sind, sondern auch regelmäßig aufgerufen werden. Dazu ist es wichtig, dass der `QTimer` im `MobileD`-Thread in festen Intervallen Signale sendet, die dann wiederum die Steuerungsfunktionen aktivieren. Graph 5.2 basiert auf Daten der gleichen Fahrt; diesmal ist die Häufigkeitsverteilung der Dauer zwischen zwei Aufrufen des Steuerungscodes aufgetragen. Der `QTimer` war so konfiguriert, dass er alle 32 Millisekunden ein Signal feuert: deutlich zu erkennen ist sein sehr zuverlässiges Zeitverhalten - in über 99,9% der Fälle liegt die Dauer des Intervalls zwischen 31900 und 32100 Mikrosekunden.

5.2 Prozessor-Auslastung

Das nun folgende Diagramm demonstriert die durchschnittliche Auslastung des Prozessors während der im vorigen Kapitel durchgeführten Fahrten. Die Ausgabe von Debugmeldungen ist bei beiden Programmen deaktiviert.

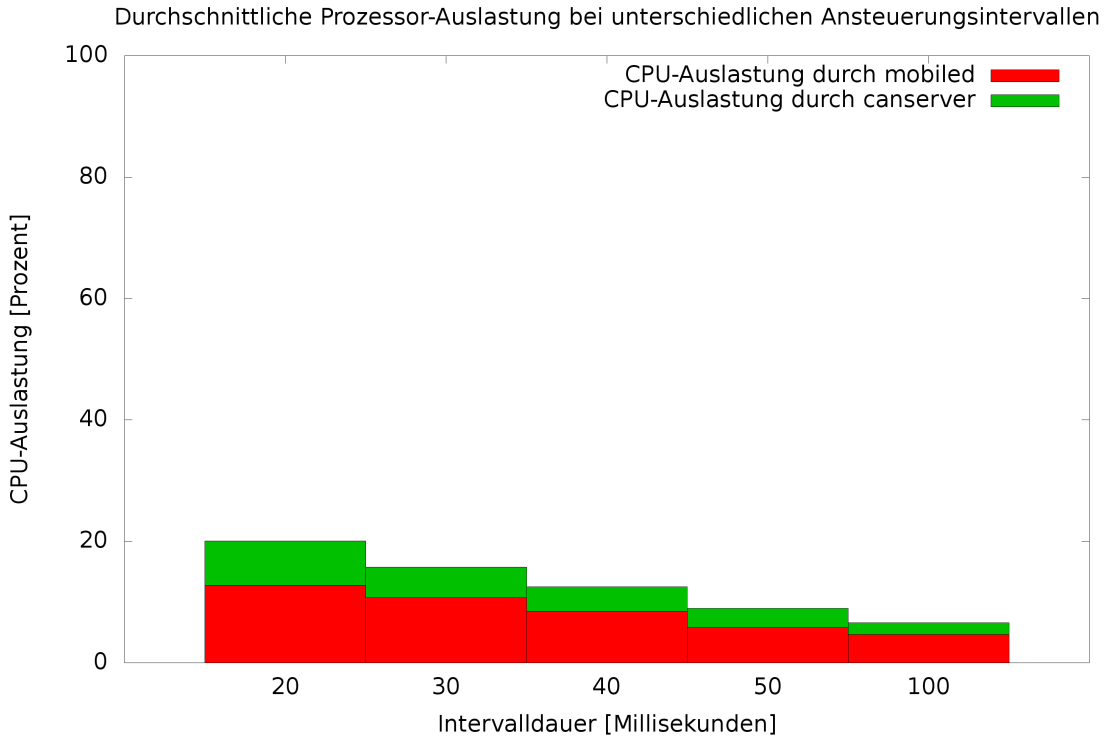


Abbildung 5.3: Durchschnittliche Prozessorauslastung durch *mobiled* und CanServer während der Fahrten aus Kapitel 5.3, ausgeführt auf einem Intel Pentium 4 Prozessor mit 2,4 GHz Taktfrequenz

CanServer zeigt ein zu erwartendes Verhalten: Da er lediglich Datenpakete zwischen CAN-Bus und TCP/IP-Netzwerk vermittelt, steigt sein Anspruch an Rechenleistung fast genau linear mit dem vorliegenden Paketdurchsatz. Werden alle 30 Millisekunden zwei Pakete (jeweils eins zur Übermittlung der Soll-Radgeschwindigkeiten an die Motoren und eins zur Übermittlung des zurückgelegten Weges zu *mobiled*) übertragen, lastet CanServer den Prozessor zu 5,1% aus.

Die CPU-Auslastung von *mobiled* ist offensichtlich auch von der Steuerungsfrequenz abhängig. Liegt diese bei 50 Hertz, benötigt mobiled 12,8% der CPU-Zeit, sinkt die Frequenz auf 10 Hertz, sind es noch 4,7%. Gut zu erkennen ist aber auch, dass *mobiled* selbst ohne Ansteuerung der Antriebe CPU-Last generieren würde. Dies hängt natürlich damit zusammen, dass neben der Motorregelung auch das Verarbeiten von Scandaten den Prozessor beansprucht. Dieser Rechenbedarf ist von der Rate eintreffender Laserscan-Daten abhängig.

5.3 Steuerungsfrequenz

In diesem Abschnitt soll experimentell ermittelt werden, inwieweit sich die Frequenz der Steuerung herabsenken lässt, ohne den Bewegungsablauf der mobilen Plattform merkbar negativ zu beeinflussen. Wie im vorigen Kapitel beschrieben, wird die Steuerung momentan alle 32 Millisekunden aufgerufen. Aus Sicherheitsgründen bremsen die Motoren ihre Fahrt automatisch, sollten sie innerhalb einer Zeit von 100 Millisekunden keine Rotationskommandos empfangen haben - somit ist die obere Schranke dieses Intervalls für *mobiled* definiert.

Der Bewegungsablauf des Roboters wird subjektiv anhand einer flüssig erscheinenden Bewegung und objektiv anhand eines Vergleichs zwischen geplantem und tatsächlich gefahrenem Pfad *bei deaktivierter Kurskorrektur* beurteilt. Wesentlich ist hierbei, wie weit sich der mobile Roboter im Verlauf der Fahrt von seiner Soll-Position entfernt. Genau dies zeigt Diagramm 5.4 für Steuerungsintervalle von 20, 30, 40, 50 und 100 Millisekunden.

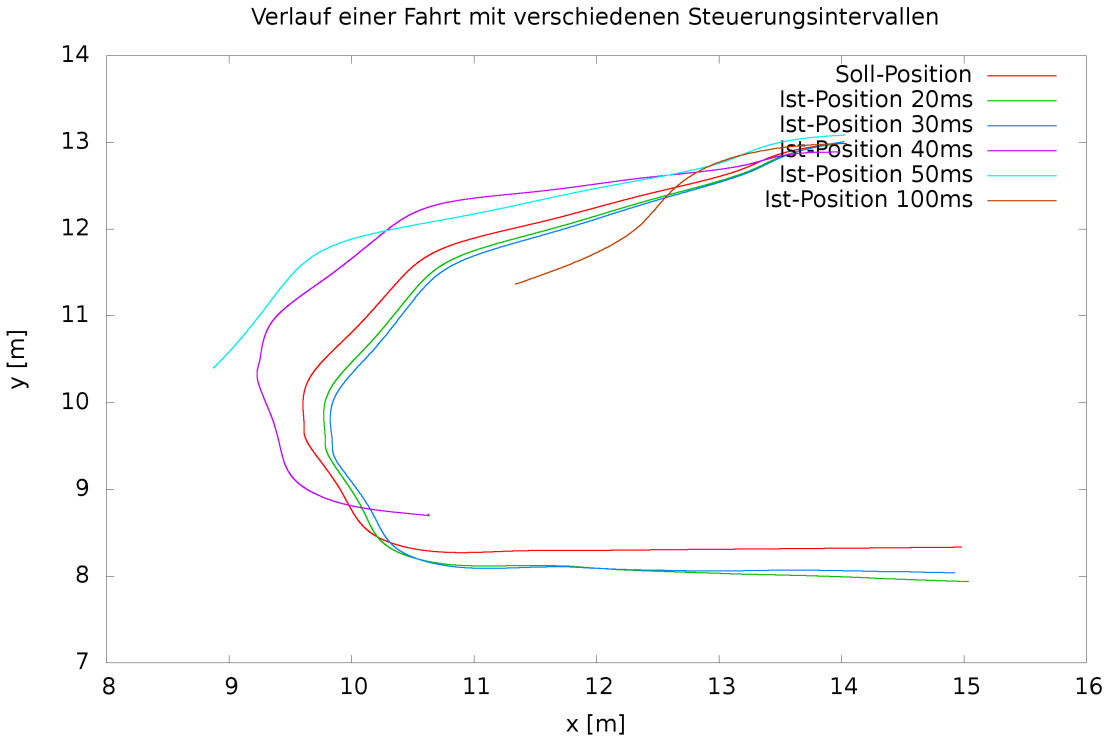


Abbildung 5.4: Distanz zwischen Soll- und Ist-Position bei verschiedenen Steuerungsintervallen

Bei Betrachtung des Diagramms wird zuerst deutlich, dass auch eine hochfrequente Regelung der Motoren einer korrigierenden Steuerung bedarf. So hatte der Roboter auch am Ende der Fahrt mit einer Ansteuerungfrequenz von 50 Hertz einen Abstand von fast 50cm zu seiner Soll-Position.

Erfolgt die Ansteuerung in Intervallen unterhalb 40 Millisekunden, stimmt die Form der abgefahrenen Kurve recht gut mit der geplanten Route überein. Die Fahrtwege im Falle von 50ms und 100ms zeigen jedoch die obere Schranke sinnvoller Ansteurungsfrequenzen deutlich auf: während dem Fahrtprofil bei 50 Millisekunden bis zu seinem Abbruch noch Ähnlichkeit zum geplanten Weg attestiert werden kann, endete die Bewegung bei 100 Millisekunden recht früh - die Kollisionsvermeidung stoppte die Fahrt, bevor ein angrenzendes Möbelstück in Mitleidenschaft gezogen werden konnte.

Ist die Kurskorrektur deaktiviert zeigt sich auch, wie wichtig eine korrekte Ausrichtung des Roboters zu Beginn der Kurvenfahrt ist. Am Beispiel der Fahrt mit Ansteuerungsintervall von 40ms lässt sich recht gut erkennen, dass der Roboter nicht vor einem Türpfosten hätte Halt machen müssen, wäre seine Anfangsorientierung präziser gewesen.

Nach Betrachtung dieser Daten erscheint eine Pause von 32 Millisekunden zwischen zwei Auf-

einander folgenden Ansteuerungen der Antriebsmotoren ein guter Kompromiss zwischen CPU-Auslastung und Spurtreue zu sein. Wahrscheinlich ließe sich dieser Wert mit einer guten Kurskorrektur noch ein wenig nach oben verschieben. Im nächsten Abschnitt wird sich aber zeigen, dass hierfür keine wirkliche Notwendigkeit besteht.

5.4 Positionskorrektur

Der folgende Abschnitt soll untersuchen, wie sehr der verwendete Algorithmus zur Kurskorrektur geeignet ist, die mobile Plattform auf ihrer Bahn zu halten. Um eine Referenz zu bieten, wurde eine Testfahrt der mobilen Plattform mit deaktivierter Kurskorrektur durchgeführt. Solche Bewegungen haben sich während der Tests als unberechenbar erwiesen: In einigen Fällen kommt die mobile Plattform immer mehr von ihrem Pfad ab, da besonders Orientierungsfehler mit der Zeit immer gravierendere Folgen haben. In anderen Fällen wiederum bleibt die Plattform dem vorhergesehenen Pfad erstaunlich nah. Entweder treten weniger störende Einflüsse wie beispielsweise Schlupf auf, oder sie scheinen sich während der Fahrt gegenseitig auszugleichen. Im vorliegenden Fall musste die Fahrt der Plattform auf halber Strecke abgebrochen werden, weil sie einem baulichen Hinderniss zu nah kam.

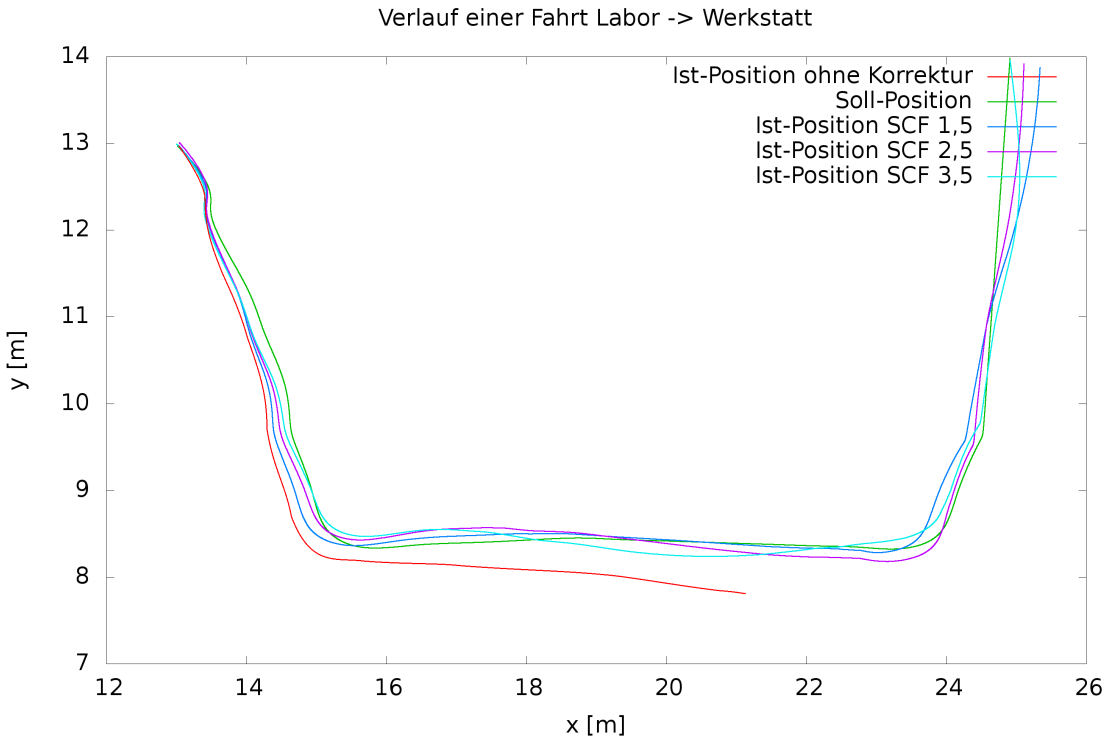


Abbildung 5.5: Präzision der Pfadverfolgung

Die anderen drei Fahrten wurden mit einer Kurskorrektur mit jeweils unterschiedlich starken Korrekturfaktoren durchgeführt. Im Detail steht die Abkürzung SCF für „SpeedCorrectionFactor“ und gibt ein Maß für die Skalierung der Geschwidigkeitsveränderung durch die Kurskorrektur an. Zu

erkennen ist, dass alle drei Fahrten sich relativ dicht an dem vorgegebenen Pfad halten. Je größer der Korrekturfaktor gewählt wird, desto mehr neigt die Plattform dazu, sich um den Pfad herum aufzuschwingen. So kommt der Roboter bei einem Faktor von 3,5 während seiner zweiten Durchquerung einer Tür (bei ca. 23,8 / 8,5) dem Türrahmen gefährlich nah.

Während der Testfahrten hat sich ein Wert von 2,5 als recht gute Größe erwiesen. Angesichts von Positionsdifferenzen bis zu 20 Zentimetern wäre ein besseres Bahnfolgeverhalten des Roboters trotzdem wünschenswert.

6 Zusammenfassung und Ausblick

In diesem letzten Kapitel soll diese Arbeit ihren Abschluss finden. Zuerst sollen die erreichten Änderungen in Hinsicht auf die zu Beginn formulierten Ziele diskutiert werden.

Ein großer Teil dieser Arbeit bestand aus der Dokumentation des Programms, seiner Architektur und seines Quelltextes, sowie die Beschreibung des Arbeitsablaufes selbst. Wie gut dies gelungen ist, möge der Leser beurteilen.

Das Hauptanliegen war, die Freiformkurve als Bewegungsprofil in *mobiled* zu etablieren. Dies ist durch Einarbeitung vieler Ergebnisse anderer Arbeiten geschehen, und so ist der TAMS-Roboter nun in der Lage, neben den althergebrachten, grundlegenden Bewegungsformen auch Raumkurven zu folgen. Die Kombination mit einer verteilten Architektur führt zu der seltenen Möglichkeit, mehrere Roboter im gleichen Arbeitsbereich ohne gegenseitige Störung betreiben zu können. Die Präzision bzw. Bahntreue der Bewegungsteuerung kann sicherlich noch weiter verbessert werden, darauf wird in einem folgenden Unterpunkt noch Bezug genommen werden.

Die Kollisionserkennung wurde im Zuge dieser Arbeit umgestaltet, ihre Abbildung im Netzwerkprotokoll deutlich vereinfacht und in Hinsicht auf das neue Bewegungsprofil erweitert. Sie ist sehr flexibel konfigurierbar, arbeitet mit hoher Frequenz und relativ geringen Ansprüchen an Rechenzeit. Die Vermeidung von Zusammenstößen funktioniert dann einwandfrei, wenn die Sensorik (also die Laserscanner) in der Lage sind, die entsprechenden Hindernisse zu erkennen. Selbstverständlich muss sich das Programmmodul auch deaktivieren lassen, beispielsweise bei manueller Annäherung an einen Arbeitsplatz des Roboters. Dies ist natürlich möglich; um Gefahren vorzubeugen, macht *mobiled* in diesem Falle seine Umwelt durch wiederholtes Piepen des PC-Lautsprechers während der Fahrt auf diesen Umstand aufmerksam.

Die Integration der Robotersteuerung in das genRob-Rahmenwerk ist, wie in Kapitel 3.2.1 erläutert, Voraussetzung für eine fehlerfreie Fahrtplanung im Beisein von Hindernissen. Der Datenaustausch zwischen mobiled und Java-Programm ist prinzipiell sehr einfach, war in der Umsetzung jedoch recht aufwändig. Es existieren verschiedene Versionen des Java Native Interface mit jeweils subtilen Unterschieden, so dass der JNI kapselnde Code Besonderheiten jeder Version berücksichtigen muss. Weiterhin erschwerte die Programmierung über Thread- und sogar Prozessgrenzen hinweg das Aufspüren von Fehlern erheblich.

Zukünftige Erweiterungen der Steuerungssoftware um neue Raumkurven und verbesserte Regelungsalgorithmen sollten nun ohne Änderungen der Architektur möglich sein. Da jede neue Bewegung ein einfaches Interface implementiert, durch das ihr alle zur Bewegungssteuerung notwendigen Daten zur Verfügung stehen, kann eine weitere Bewegung (wie z.B. mit hermitischen Splines) durch Hinzufügen der entsprechenden Klasse und Austausch einer Zeile in `MobileD::slotNewWayPointsArrived(QList<QPointF> wayPoints)` hinzugefügt werden.

Das Interface zur Implementierung einer Regelung ist ebenfalls sehr einfach gestaltet. Die Korrekturfunktion bekommt lediglich die beiden von der Fahrtsteuerung berechneten Sollgeschwindig-

keiten sowie die momentane und die gewünschte Pose des Roboters übergeben. Basierend darauf kann sie die Geschwindigkeitswerte korrigieren und an die Fahrtplanung zurückgeben, damit sie von dort weiter zu den Motoren geschickt werden.

6.1 Weitere Entwicklungsmöglichkeiten

Im Folgenden werden mehrere Möglichkeiten aufgeführt, deren Umsetzung innerhalb *mobiled* zu einem besseren Verhalten des Roboters führen könnte. Zwar arbeitet die Software Fahraufträge schon recht zuverlässig ab, trotzdem bleiben noch viele Gelegenheiten, die Bewegungssteuerung zu verbessern.

6.1.1 Lokalisierung und Kartierung

Gleichzeitig naheliegend und nützlich erscheint, als nächsten Schritt ein SLAM-Verfahren zu implementieren. SLAM ist eine Abkürzung von „Self Localization and Mapping“ und beschreibt einen Algorithmus, mit dessen Hilfe ein mobiler Roboter durch Einsatz von Laserscannern seine Umgebung völlig ohne Vorwissen selbst erkundet. So kann nicht nur eine Karte im Vergleich zur manuellen Eingabe sehr viel schneller und meist auch präziser erstellt werden - praktisch als Nebenprodukt weiß der Roboter sich in dieser entstehenden Karte selbst einzuordnen; eine Selbstlokalisierung wäre also auch ohne Lasermarken möglich. Eine Herausforderung ist hierbei wohl die Segmentierung der vom Laserscanner gelieferten Punktwolken, um die von genMap zu verwaltende Datenmenge zu beschränken. Ferner wäre es nötig, mit Hilfe weiterer Sensoren sicherzustellen, dass die mobile Plattform beim Erkunden gefährliche Bereiche wie Treppen meidet. Hierfür scheint z.B. ein Laserscanner auf einer neige- und Schwenkeinheit geeignet.

6.1.2 Kollisionsvermeidung

Sobald *mobiled* eine Kollision vorhersagt, wird die Plattform abgebremst und ein komplett neuer Pfad geplant. Dies bedeutet jedes Mal eine zeitliche Verzögerung, die dem Beobachter als unnötige „Gedenkpause“ erscheint.

Schön wäre es, der Kollisionsvermeidung zumindest teilweise reaktive Komponenten hinzuzufügen, um so bei einer drohenden Kollision die Phase der Fahrtplanung zu verkürzen. So könnte ein Hindernis bei ausreichendem seitlichen Freiraum durch Einfügen eines Punktes in die momentan vorliegende Raumkurve umfahren werden. Dies würde zuerst eine Neuberechnung des Splines (und somit auch der Zuordnung von Kurvenparameter zu Kurvenlänge) erfordern. Anschließend daran gilt es den Punkt auf der Kurve zu ermitteln, dem die mobile Plattform momentan am nächsten ist. Je nach zeitlichem Anspruch des Verfahrens könnte die Fahrt fortgesetzt werden, ohne dass der Roboter abbremsen muss.

6.1.3 Weitere Raumkurven

Denkbar ist auch die Fahrtplanung derart zu erweitern, dass außer kubischen Splines auch andere Raumkurven zum Einsatz kommen können. Dies ist in großen Teilen bereits am Beispiel von her-

mitischen Kurven implementiert (in der Klasse `SplineHermite`). Natürlich könnte man hier auch mit Catmull-Rom-, Cardinal- oder Kochanek-Bartels-Splines experimentieren - bei den ersteren beiden handelt es sich lediglich um hermitische Splines mit eingeschränkter Parameterwahl, letzere stellen eine Erweiterung dieser dar (KB84). Das Optimieren der nun zur Verfügung stehenden zusätzlichen Kurvenparameter Tension, Bias und Contiunity könnte zu besseren Trajektorien als bei kubischen Splines führen. Splinewinkel und Krümmug müssten bei diesen Splines allerdings numerisch bestimmt werden, dies sollte hinreichend schnell und präzise möglich sein.

6.1.4 Optimierung der Fahrtprofile

Zu Beginn einer Bewegung dreht die mobile Plattform sich momentan zuerst in den Anfangswinkel der Raumkurve und beginnt dann die eigentliche Kurvenfahrt. Analog stimmen beim Ende der Fahrt die Orientierungen von Spline und mobiler Plattform überein, somit bleibt eine Drehung in die Orientierung der Zielpose nötig. Sofern an Start- und Endpunkt ausreichend Platz vorhanden ist, könnten die Tangenten des Splines an seinen Anfangs- und Endpunkten der Orientierung des Roboters bzw. der Zielpose angepasst werden. So blieben der mobilen Plattform zwei zeitraubende und unnatürlich anmutende Manöver erspart.

Die Prüfung, ob die Änderung der Splinetangenten zu Kollisionen führen wird, erfordert jedoch die Einführung des Spline-Verfahrens in den Fahrtplaner. Dieser verarbeitet zum jetzigen Zeitpunkt außerdem keine Posen, sondern nur Punkte: das Konzept einer Orientierung fehlt ihm also. Diese Entwicklung würde vornehmlich Änderungen nicht in *mobiled*, sondern im genRob-Rahmenwerk bedeuten.

6.1.5 Kurskorrektur

Wie in Abschnitt 4.2.2 aufgeführt, ist die momentan zur Kurskorrektur verwendete Regelung recht einfacher Natur. Die Implementierung eines komplexeren Algorithmus hätte jedoch weit mehr Aufwand erfordert und war nicht Thema dieser Arbeit. Ferner zeigt der verwendete Algorithmus im Betrieb auf TASER ein zufriedenstellendes Regelungsverhalten (vgl. Diagramm 5.4). Wahrscheinlich ist TASER aufgrund seines Gewichts, seiner Vollgummiräder, seiner präzisen Motorsteuerung und der relativ geringen Beschleunigungen vergleichweise wenig anfällig für Kursabweichungen.

Würde *mobiled* auf anderen Robotern (wie z.B. denen der Pioneer-Serie[1]) Verwendung finden, wäre eine bessere Regelung von Nöten. Durch höheren Schlupf sowie Änderung des Reifendurchmessers durch Abnutzung, Luftdruck und Aufnahme einer möglicherweise nicht ausbalancierten Nutzlast gäbe es hier weit größeren Bedarf nach einer guten Gegensteuerung. Letztlich ist diese aber auch für Roboter sinnvoll, deren Konstruktion mit TASER vergleichbar ist, die aber in anderen Bedingungen betrieben werden: Würde TASER über ausgedehnte Zeiträume größeren Verschleißerscheinungen ausgesetzt sein, könnte sich auch hier eine Roboterdynamik ergeben, die besserer Regelung bedarf.

[1] Die Pioneer-Roboter der Firma ActivMedia Robotics (`http://www.activrobots.com/`) sind in der Forschung der mobilen Robotik sehr verbreitet und auch am Fachbereich TAMS vorhanden, insofern ist ein Einsatz von *mobiled* auf Pioneer-Robotern durchaus denkbar.

Ansätze hierfür existieren reichlich, eine Untermenge ist bereits in Kapitel 2 vorgestellt worden. So wäre es denkbar, die mobile Plattform mit Hilfe eines Fuzzy-Logik-Controllers oder gar durch Verwendung neuronaler Netze auf seiner Bahn zu halten. Letztere hätten den Vorteil, den Verschleiß verschiedener Bauteile „lernen“ und so kompensieren zu können.

6.1.6 Fahrtplanung im Vorhandensein von Hindernissen

Die Hindernisverwaltung in *mobiled* ist zum jetzigen Zeitpunkt sehr grundlegend implementiert: erkannte Blockaden werden für einen konfigurierbaren (`Configuration::getObstacleTimeout()`) Zeitraum in der Karte gespeichert und somit von der Fahrtplanung gemieden. Leider sind viele Szenarien denkbar, in denen diese Methodik zu Problemen führt. Führen beispielsweise drei verschiedene Wege zu einem gewünschten Ziel und die beiden kürzesten Wege sind versperrt, so wird *mobiled* den Roboter abhängig von der Verweildauer von Hindernisen in der Karte immer wieder zwischen den beiden kürzeren, versperrten Wegen oszillieren lassen. Die Möglichkeit, den dritten Weg einzuschlagen, wird dabei nie in Betracht gezogen.

Letztlich lässt sich dieses Problem nicht allgemein durch Anpassung der Hindernisverweildauer lösen. Vielmehr ist zusätzliches Wissen auf einer höheren Ebene nötig, um dem Roboter den dritten Weg nahezubringen. Eine relativ einfache Methode ist, Regionen für jedes vorkommende Hindernis eine negative Bewertung zuzuordnen, so dass Routen mit weniger Barrieren dem Fahrtplaner attraktiver erscheinen.

Weiter wäre denkbar, das Verfahren zur Wegsuche für den Einsatz in Mehrroboterumgebungen zu verbessern. So könnte die Effizienz mehrerer simultan betriebener mobiler Plattformen verbessert werden, indem ein von einer Plattform befahrener Weg bei Planungen für andere Roboter nach Möglichkeit von vornherein gemieden wird.

Literaturverzeichnis

[AHK08] AGUIAR, A. P. ; HESPANHA, Joao P. ; KOKOTOVIC, Petar V.: Performance limitations in reference tracking and path following for nonlinear systems. In: *Automatica* 44 (2008), Nr. 3, S. 598–610. – ISSN 0005–1098 60

[BP06] BISTRY, Hannes ; PÖHLSEN, Stephan: *Entwicklung eines Eingebetteten Systems zur ressourcenschonenden und plattformunabhängigen Anbindung von SICK-Lasermesssystemen*, Universität Hamburg, Diplomarbeit, 2006 64

[BPWZ07] BISTRY, Hannes ; POHLSEN, Stephan ; WESTHOFF, Daniel ; ZHANG, Jianwei: Development of a Smart Laser Range Finder for an Autonomous Service Robot. In: *Proceedings of the 2007 IEEE International Conference on Integration Technology (ICIT 2007)*. Shenzhen, China, März 2007 5

[Bro85] BROOKS, Rodney A.: A Robust Layered Control System For a Mobile Robot. Cambridge, MA, USA : Massachusetts Institute of Technology, 1985. – Forschungsbericht 13

[Bro87] BROOKS, R. A.: A robust programming scheme for a mobile robot. In: *Proc. of the NATO Advanced Research Workshop on Languages for sensor-based control in robotics*. London, UK : Springer-Verlag, 1987. – ISBN 0–387–17665–9, S. 509–522 14

[Br9] BRÄUNL, Thomas: EyeBot: a family of autonomous mobile robots. In: *In Proceedings of 6th International Conference on Neural Information Processing (ICONIP'99)*. Perth (Australia), 1999, S. 645–649a 9

[BS02] BATAVIA, Parag ; SINGH, Sanjiv: Obstacle Detection in Smooth High Curvature Terrain. In: *Proceedings of the IEEE Conference on Robotics and Automation (ICRA '02)*, 2002 9

[BS07] BAKARI, Mohamed J. ; SEWARD, Derek W.: Human arm-like mechanical manipulator - the design and development of a multi -arm mobile robot for nuclear decommissioning. In: *ICINCO-RA*, 2007, S. 168–175 1

[CMLL00] COOMBS ; MURPHY ; LACAZE ; LEGOWIK: Driving Autonomously Offroad up to 35 km/h. In: *Proceedings of the IEEE Intelligent Vehicles Symposium 2000*. Detroit, USA, Oktober 2000, S. 186–191 9

[CMPV06] CHESI, G. ; MARIOTTINI, G.L. ; PRATTICHIZZO, D. ; VICINO, A.: Epipole-Based Visual Servoing for Mobile Robots. In: *Advanced Robotics* 20 (2006), Februar, Nr. 2 17

[EHS98] EGERSTEDT, Magnus ; HU, X. ; STOTSKY, A.: Control of a car-like robot using a virtual vehicle approach. In: *Proceedings of the 37th IEEE Conference on Decision and Control, Tampa, FL*, 1998, S. 1502–1507 60

[Gaw05] GAWORSKI, Björn: *Ein probabilistischer Ansatz für robuste Lokalisierung eines mobilen Robotersystems mittels omnidirektionaler Sichtsysteme*, Universität Hamburg, Diplomarbeit, 2005 17

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995 48

[GSN00] GRAF, B. ; SCHRAFT, R. ; NEUGEBAUER, J.: *A Mobile Robot Platform for Assistance and Entertainment*. 2000 1

[HAK03] HWANG, Jung-Hoon ; ARKIN, Ronald C. ; KWON, Dong-Soo: Mobile robots at your fingertip: Bezier curve on-line trajectory generation for supervisory control. In: *Proceedings of the the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)* Bd. 2, 2003, S. 1444–1449 9

[Heb49] HEBB, D.O.: *The Organization of Behavior*. New York, USA : Wiley, 1949 10

[Hen98] HENDERSON, David W.: *Differential geometry*. Prentice Hall, Inc, 1998 52

[HK06] HOWARD, Thomas ; KELLY, Alonzo: Trajectory and Spline Generation for All-Wheel Steering Mobile Robots. In: *Proceedings of the the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006)*, 2006, S. 4827–4832 1

[HK07] HOWARD, Thomas M. ; KELLY, Alonzo: Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots. In: *Int. J. Rob. Res.* 26 (2007), Nr. 2, S. 141–166. – ISSN 0278–3649 9

[HNR68] HART, P. E. ; NILSSON, N.J. ; RAPHAEL, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics SSC* 4 (1968), Nr. 2, S. 100–107 51

[Hor94] HORN, Joachim: Cartesian Motion Control of a Mobile Robot. In: *Proceedings of the Intelligent Vehicles '94 Symposium*, 1994. – ISBN 0–7803–2135–9, S. 443–448 9

[Kal60] KALMAN, R. E.: A New Approach to Linear Filtering and Prediction Problems. In: *Transactions of the ASME–Journal of Basic Engineering* 82 (1960), Nr. Series D, S. 35–45 36

[Kan04] KANARAT, Amnart: *Motion Planning and Robust Control for Nonholonomic Mobile Robots under Uncertainties*, Virginia Polytechnic Institute and State University, Dissertation, Mai 2004 9

[KB84] KOCHANEK, Doris H. U. ; BARTELS, Richard H.: Interpolating splines with local tension, continuity, and bias control. In: *SIGGRAPH Comput. Graph.* 18 (1984), Nr. 3, S. 33–41. – ISSN 0097–8930 71

[KC02] KOPACEK, P. ; CHIVAROV, N.: Modular Approach in Projecting of Intelligent Mobile Robots. In: *Problems of Engineering Cybernetics and Robotics* 53 (2002), S. 36–48 1

[KNDG02] KALMAR-NAGY, Tamas ; D'ANDREA, Raffaello ; GANGULY, Pritam: Near-optimal dynamic trajectory generation and control of an omnidirectional vehicle. In: *Robotics and Autonomous Systems* 46 (2002), Nr. 1, S. 47–64. – ISSN 0921–8890 9

[KPH88] KUAN, D. ; PHIPPS, G. ; HSUEH, A.-C.: Autonomous Robotic Vehicle Road Following. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 10 (1988), Nr. 5, S. 648–658. – ISSN 0162–8828 9

[LaV06] LAVALLE, S. M.: *Planning Algorithms*. Cambridge, U.K. : Cambridge University Press, 2006. – Available at http://planning.cs.uiuc.edu/ 27, 28, 51

[LL04] LINDEMANN, S. ; LAVALLE, S.: Current issues in sampling-based motion planning, 2004 10

[LTF97] LEYDEN, Mark ; TOAL, Daniel ; FLANAGAN, Colin: A Fuzzy Logic Based Navigation System for a Mobile Robot. In: *Journal of Experimental and Theoretical Artificial Intelligence, special issue on Software Architectures for Physical Agents* (1997) 16

[Mar07] MARTYNENKO, Yu G.: Motion Control of Mobile Wheeled Robots, 2007, S. 6569–6606 9

[Mat97] MATARIC, Maja J.: Behavior-Based Control: Examples from Navigation, Learning, and Group Behavior. In: *Journal of Experimental and Theoretical Artificial Intelligence, special issue on Software Architectures for Physical Agents* 9 (1997), S. 323–336 10

[MKS97] MA, Y. ; KOSECK'A, J. ; SASTRY, S.: *Vision guided navigation for a nonholonomic mobile robot*. 1997 17

[MS07] MASEHIAN, Ellips ; SEDIGHIZADEH, Davoud: Classic and Heuristic Approaches in Robot Motion Planning – A Chronological Review. In: *Proceedings of World Academy of Science, Engineering and Technology* Bd. 23. japan, 2007. – ISSN 1307–6884, S. 101–106 9

[Neo04] NEOBOTIX: *Neobotix IOBoard 16/16, Technical Description*. http://www.neobotix.de/en/downloads/docs/IOBoard-TechnicalDescription.pdf, November 2004 23

[Nil84] NILSSON, Nils J.: Shakey The Robot / AI Center, SRI International. 333 Ravenswood Ave., Menlo Park, CA 94025, Apr 1984 (323). – Forschungsbericht 9

[OS97] OMIDVAR, Omid M. (Hrsg.) ; SMAGT, Patrick Van D. (Hrsg.): *Neural Networks for Robotics*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. – ISBN 0125262809 11, 12

[Oub06] OUBBATI, Mohamed: *Neural Dynamics for Mobile Robot Adaptive Control*, Universität Stuttgart, Diplomarbeit, 2006 11

[PA01] PAROMTCHIK, Igor E. ; ASAMA, Hajime: Optical Guidance System for Multiple Mobile Robots. In: *ICRA*, IEEE, 2001. – ISBN 0–7803–6578–X, 2935-2940 9

[PN95] PAROMTCHIK, I. ; NASSAL, U.: *Reactive Motion Control for an Omnidirectional Mobile Robot*. 1995 9

[RHSH05] ROTH, Stephan ; HAMNER, Bradley ; SINGH, Sanjiv ; HWANGBO, Myung: Results in Combined Route Traversal and Collision Avoidance. In: *International Conference on Field & Service Robotics (FSR 2005)*, 2005 9

[Ror05] RORIE, Demetrus: Autonomous Ground Vehicle (AGV) Project. (2005) 9

[Sch04] SCHERER, Torsten: *A mobile service robot for automisation of sample taking and sample management in a biotechnological pilot laboratory*, Universität Bielefeld, Diss., 2004 5

[Sim90] SIMMONS, Reid: Concurrent Planning and Execution for a Walking Robot / Robotics Institute, Carnegie Mellon University. Pittsburgh, PA, Juli 1990 (CMU-RI-TR-90-16). – Forschungsbericht 10

[SKLL05] SÜNDERHAUF, N. ; KONOLIGE, K. ; LEMAIRE, T. ; LACROIX, S.: Comparison of Stereovision Odometry Approaches. In: *IEEE International Conference on Robotics and Automation ICRA05 Barcelona, Planetary Rover Workshop*, 2005 17

[SRK97] SAFFIOTTI, A. ; RUSPINI, E. H. ; KONOLIGE, K.: Using Fuzzy Logic for Mobile Robot Control. In: DUBOIS, H. Prade D. (Hrsg.) ; ZIMMERMANN, H. J. (Hrsg.): *International Handbook of Fuzzy Sets and Possibility Theory* Bd. 5. Norwell, MA, USA, and Dordrecht, The Netherlands : Kluwer Academic Publishers Group, 1997 15, 16

[TA01] TOURINO, Sergio Roberto G. ; ALVARES, Alberto J.: Fuzzy Logic Control System to the Mobile Robot Motion Through Web. (2001) 15, 16

[TBB03] THOMAS, Jason ; BLAIR, Alan ; BARNES, Nick: Towards an Efficient Optimal Trajectory Planner for Multiple Mobile Robots. (2003) 9

[TBF06] THRUN, Sebastian ; BURGARD, Wolfram ; FOX, Dieter: *Probabilistic Robotics*. MIT Press, 2006 13, 37, 38

[T1] TOURINO, Sérgio Roberto G. ; ÁLVARES, Alberto J.: Fuzzy Logic Control System to the Mobile Robot Motion Through Web. (2001) 9

[VF01] VINCENT, Stephen ; FORSEY, David: Fast and Accurate Parametric Curve Length Computation. In: *Journal of Graphics Tools* 6 (2001), Nr. 4, S. 29–40 58

[WS02] WESTHOFF, Daniel ; SCHNEIDER, Axel: *Autonomous Navigation and Control of a Mobile Robot in a Cell Culture Laboratory*, Universität Bielefeld, Diplomarbeit, 2002 5

[YP95] YEN, J. ; PFLUGER, N.: A fuzzy logic based extension to Payton and Rosenblatt's command fusion method for mobile robot navigation. In: *IEEE Trans. on Systems, Man, and Cybernetics* 25 (1995), Nr. 6, S. 971–978 16

[Zad65] ZADEH, L.A.: Fuzzy Sets. In: *Information Control* 8 (1965), S. 338–353 15

[ZRH94] ZHANG, Jianwei ; RACZKOWSKY, Jörg ; HERP, Andreas: Emulation of Spline Curves and Its Applications in Robot Motion Control. (1994) 16

[ZS98] ZHANG, Jianwei ; SCHWERT, Volkmar: Rapid Learning of Sensor-Based Behaviours of Mobile Robots Based on B-Spline Fuzzy Controllers. In: *In Proceedings of the 1998 IEEE Int. Conference on Fuzzy Systems, Anchorage*, 1998 16

[ZTA] ZAVLANGAS, Panagiotis G. ; TZAFESTAS, Prof. Spyros G. ; ALTHOEFER, Dr. K.: *Fuzzy Obstacle Avoidance and Navigation for Omnidirectional Mobile Robots* 16

A Das *mobiled*-Protokoll

Appendix A spezifiziert das Kommunikationsprotokoll zwischen der Steuerung *mobiled* des mobilen Roboters des AB TAMS am Fachbereich Informatik der Universität Hamburg und Client-Anwendungen, die den Roboter steuern sollen. Auch das Roblet-Server-Modul `uhh.fbi.tams.mobilerobot` nutzt dieses Protokoll um den Roboter zu steuern.

A.1 Protokolleigenschaften

- Es handelt sich um ein zustandsloses Protokoll, d.h. 1 Kommando hat genau 1 Antwort zur Folge, es erfolgt keine Speicherung von Kontextinformationen über zwei Kommandos hinweg.
- Alle Daten werden in 32 Bit Worten übertragen (damit alles immer *aligned* ist). Es gibt also nur die Typen *int* (S32), *unsigned int* (U32) und *float* (F32). Gezählt wird aber in einzelnen Bytes, so dass die Längenangabe für das ganze Telegramm ein Vielfaches von 4 sein muss.
- Negative Statuswerte sind Fehler, echt positive Zahlen Warnungen und 0 ist OK.
- Die Kodierung der Worte erfolgt im *little-endian* (Intel®) Format, d.h. 0x12345678 liegt linear im Speicher als 0x78 0x56 0x34 0x12 und wird auch in dieser Reihenfolge als Bytestrom übertragen. Anderes Beispiel: Der MAGIC Wert „HELO“ (0x4f4c4548) wird als 0x48 0x45 0x4c 0x4f übertragen.
- Die Kodierung von Fliesskommawerten erfolgt im *little-endian* 32 Bit IEEE-754-Format. Als Beispiel für den Bytestrom mögen gelten:
  - -1.0 → 0x00 0x00 0x80 0xbf
  - +0.0 → 0x00 0x00 0x00 0x00
  - +1.0 → 0x00 0x00 0x80 0x3f
  - +3.14159 → 0xd0 0x0f 0x49 0x40
  - +INF → 0x00 0x00 0x80 0x7f
  - -INF → 0x00 0x00 0x80 0xff
  - NAN → 0x00 0x00 0xc0 0x7f
- Die Implementierung des Paketformats erfolgt in *mobiled* durch die Klasse `Packet`, zu finden in den Dateien `packet.cpp` und `packet.h`.
- Die Implementierung der Netzwerkkommunikation erfolgt in *mobiled* durch die Klasse `NetworkServer`, zu finden in den Dateien `networkserver.cpp` und `networkserver.h`.

- Jegliche Winkel werden in der Einheit Radiant und im mathematisch positiven Sinne übertragen. Positive Winkeldeltas bedeuten hier also Linksdrehungen. Ein absoluter Winkel von 0 rad bedeutet eine Roboterausrichtung von links nach rechts auf der Karte bzw. vom Flurende zum Fahrstuhl im TAMS-Flur.

A.2 Paketaufbau

Im Folgenden soll der Aufbau der Pakete erläutert werden.

A.2.1 Allgemeiner Paketaufbau

Unterschieden wird (nur) zwischen Kommandos und Antworten, wobei letztere am Bit 0x80000000 im Kommandofeld zu erkennen sind. Kommandos haben eine Mindestlänge von 4 Worten (à 4 Byte), Antworten von 5 Worten weil sie immer mindestens 1 Statuswort enthalten müssen. Dieses Wissen kann mit dem Magic Byte, der Länge und der Prüfsumme für Plausibilitätstests benutzt werden.

A.2.2 Generelles Layout

Wort	0				1				2				3	...	N-2	N-1			
Byte	0	1	2	3	4	5	6	7	8	9	10	11	...						
Bezeichnung	<MAGIC>				<SIZE>				<CMD>				<DATA>			<CRC>			

- MAGIC: Datentyp U32 Wert ist stets 0x4f4c4548 ("HELO")
- SIZE: Datentyp U32, gezählt in Bytes über das gesamte Telegramm. Die untersten 2 Bits sind immer 0 weil nur Vielfache von 4 erlaubt sind.
- CMD : Datentyp U32, Wert ist 0x00?????? bei einem Befehl und 0x80?????? bei dem zugehörigen Antwortpaket. Abgesehen vom Antwort-Flag gleichen sich die CMD-Werte von Befehl- und Antwortpaket.
- DATA : Datentyp ist kommandoabhängig, Werte sind beliebig viele Nutzdaten
- CRC : Datentyp U32, Wert ist die CRC-32 Prüfsumme

A.3 Kommandos

A.3.1 Protokoll-Kommandos

Die nun folgenden Befehle dienen zur Verwaltung des Protokolls selbst.

A.3.1.1 PING – Überprüfen der Verbindung

Kommando	Argument	Erklärung
0x00000000	-	-
0x80000000	STATUS	S32, Sollte stets 0 sein.

`PING` fordert ein Antwortpaket von *mobiled*.

A.3.1.2 CHECKPROTOCOL – Überprüfen der unterstützten Kommandos

Kommando	Argument	Erklärung
0x00000001	-	-
0x80000001	STATUS	S32
	VERSION	U32, Protokollversion, Wert ist 1
	foreach(Kommando) {	
	CMD	U32, ID des Kommandos
	}	

`CHECKPROTOCOL` fordert *mobiled* auf, ein Antwortpaket mit der Protokollversion und den IDs aller unterstützten Kommandos zu senden. Die Protokollversion ist stets 1 (im Gegensatz zum Wert 0 der bisherigen mobiled-Versionen). Die Anzahl der unterstützten Kommandos ist nicht explizit aufgeführt, ergibt sich aber aus der Länge des Pakets.

A.3.2 Roboterstatus-Kommandos

Diese Befehle dienen zur Abfrage aktueller Roboterparameter

A.3.2.1 GETROBOTSTATUS – Abfrage des aktuellen Roboterstatus

Kommando	Argument	Erklärung
0x00010000	-	-
0x80010000	STATUS	S32
	ROBOTERSTATUS	U32, Aktueller Zustand des Roboters

`GETROBOTSTATUS` fordert *mobiled* auf, ein Antwortpaket mit dem aktuellen Zustand des Roboters zu senden. Diese Zustände entsprechen den Werten des in `robot.h` definierten `enum RobotStatus`:

```
enum RobotStatus
{
        idle,           // 0    robot is bored.

        stalled,        // 1    robot is blocked. Waiting for new
                        //      path and/or obstacle to disappear.

        movementSetup,  // 2    robot is waiting for initial path
                        //      or is constructing motion profile.
```

```
        moving,          // 3    robot is moving, please don't talk
                         //      to the driver.

        aborting         // 4    robot is braking, will then go idle.
                         //      Reason is an obstacle or user abort.
};
```

A.3.2.2 GETBATTERYVOLTAGE – Abfrage der aktuellen Akkuspannung

Kommando	Argument	Erklärung
0x00010001	-	-
0x80010001	STATUS	S32
	BATTERYVOLTAGE	F32, Aktuelle Akkuspannung

`GETBATTERYVOLTAGE` fordert *mobiled* auf, ein Antwortpaket mit der aktuellen Akkuspannung zu senden.

A.3.2.3 GETDRIVETEMPERATURES – Abfrage der aktuellen Temperaturen der Antriebsmotoren

Kommando	Argument	Erklärung
0x00010002	-	-
0x80010002	STATUS	S32
	TEMP LEFT	F32, Aktuelle Temperatur links
	TEMP RIGHT	F32, Aktuelle Temperatur rechts

`GETDRIVETEMPERATURES` fordert *mobiled* auf, ein Antwortpaket mit den aktuellen Temperaturen der Antriebsmotoren zu senden. Die Einheit der Temperaturen ist Grad Celsius.

A.3.2.4 GETCURRENTPATH – Abfrage des aktuellen Bewegungspfades

Kommando	Argument	Erklärung
0x00010003	-	-
0x80010003	STATUS	S32
	foreach(Wegpunkt) {	
	WEGPUNKT-X	F32, X-Koordinate des Wegpunkts
	WEGPUNKT-Y	F32, Y-Koordinate des Wegpunkts
	}	

`GETCURRENTPATH` fordert *mobiled* auf, ein Antwortpaket mit den Wegpunkten des aktuellen Bewegungspfades zu senden. Status ist genau dann 0, wenn der Roboter eine Bewegung ausführt und sich in einem der Zustände *moving*, *stalled* oder *aborting* befindet, sonst -1.

Ist die Bewegung ein Spline, werden alle seine Wegpunkte zurückgesendet. Ist die Bewegung eine Rotation, wird ein Wegpunkt mit der aktuellen Position gesendet. Ist die Bewegung eine Translation, werden die aktuelle und die Zielposition als Wegpunkte zurückgesendet.

A.3.3 Selbstlokalisierungs-Kommandos

Die nun folgenden Befehle betreffen die Selbstlokalisierung des Roboters.

A.3.3.1 GETPOSE – Abfrage der aktuellen Roboterpose

Kommando	Argument	Erklärung
0x00020000	-	-
0x80020000	STATUS	S32
	POSE-X	F32, X-Koordinate der Pose
	POSE-Y	F32, Y-Koordinate der Pose
	POSE-WINKEL	F32, Winkel der Pose

`GETPOSE` fordert *mobiled* auf, ein Antwortpaket mit der aktuellen Roboterpose zu senden. Status ist stets 0.

A.3.3.2 SETPOSE – Setzen der aktuellen Roboterpose

Kommando	Argument	Erklärung
0x00020001	POSE-X	F32, X-Koordinate der Pose
	POSE-Y	F32, Y-Koordinate der Pose
	POSE-WINKEL	F32, Winkel der Pose
0x80020001	STATUS	S32
	POSE-X	F32, X-Koordinate der Pose
	POSE-Y	F32, Y-Koordinate der Pose
	POSE-WINKEL	F32, Winkel der Pose

`SETPOSE` fordert *mobiled* auf, die übergebene Pose als eigene Pose für die Selbstlokalisierung zu übernehmen. Im Antwortpaket ist der Status (stets 0) und die nun in der Lokalisierung gespeicherte Pose enthalten.

A.3.3.3 UPDATEMARKS – Neuladen bekannter Lasermarken

Kommando	Argument	Erklärung
0x00020002	-	-
0x80020002	STATUS	S32, Sollte stets 0 sein.

`UPDATEMARKS` fordert *mobiled*s Selbstlokalisierung auf, die bekannten Lasermarken neu einzulesen. Dies kann je nach Implementierung das Neuladen der entsprechenden Datendatei oder eine Anfrage an einen laufenden genMap-Server sein.

A.3.4 Bewegungs-Kommandos

Die nächsten Befehle ermöglichen die Bewegungssteuerung der Roboters

A.3.4.1 STOP – Abbruch einer Bewegung bzw. ihrer Planung

Kommando	Argument	Erklärung
0x00030000	-	-
0x80030000	STATUS	S32, Sollte stets 0 sein.

`STOP` fordert *mobiled* auf, die aktuelle Bewegung abzubrechen. Ein fahrender Roboter wird vom Zustand *moving* zu *aborting* wechseln, abbremsen und dann im Zusatnd *idle* enden. Ist der Roboter erst in der Phase der Bewegungsplanung (z.B. auf Antwort vom Pfadplaner wartend), wird vom Zustand *movementSetup* zu *aborting*, dann zu *idle* gewechselt. Ist der Roboter vor Eintreffen des Pakets im Zustand *idle*, so hat das Paket keine Auswirkung.

Der Status im Anwtortpaket ist stets 0. Der Roboter wird sich erst wieder bewegen, wenn ihm ein neuer Bewegungsbefehl zugestellt wird.

A.3.4.2 MOVETRANSLATE – Geradlinige Bewegung

Kommando	Argument	Erklärung
0x00030001	Streckenlänge	F32, Streckenlänge der Bewegung
0x80030001	STATUS	S32

`MOVETRANSLATE` fordert *mobiled* auf, eine geradlinige Bewegung auszuführen. Ist die übergebene Streckenlänge negativ, fährt der Roboter rückwärts.

Der im Antwortpaket enthaltene Status ist 0 wenn der Roboter bisher im Status *idle* war. Er wechselt dann in den Status *movementSetup* und schließlich *moving*. Der Status ist -1, wenn sich der Roboter vorher nicht im Zustand *idle* befand; die Bewegung wird dann nicht ausgeführt.

A.3.4.3 MOVEROTATE – Drehung um den Robotermittelpunkt

Kommando	Argument	Erklärung
0x00030002	Drehwinkel	F32, Drehwinkel der Rotation
0x80030002	STATUS	S32

`MOVEROTATE` fordert *mobiled* auf, eine Rotation um den eigenen Mittelpunkt auszuführen.

Der im Antwortpaket enthaltene Status ist 0 wenn der Roboter bisher im Status *idle* war. Er wechselt dann in den Status *movementSetup* und schließlich *moving*. Der Status ist -1, wenn sich der Roboter vorher nicht im Zustand *idle* befand; die Bewegung wird dann nicht ausgeführt.

A.3.4.4 MOVESPLINE – Weiche Bewegung zu einem Zielpunkt

Kommando	Argument	Erklärung
0x00030003	POSE-X	F32, X-Koordinate der Zielpose
	POSE-Y	F32, Y-Koordinate der Zielpose
	POSE-WINKEL	F32, Winkel der Zielpose
0x80030003	STATUS	S32

MOVESPLINE fordert *mobiled* auf, eine Spline-Bewegung von der aktuellen Roboterpose zum angegebenen Zielpunkt durchzuführen. Die nötigen Wegpunkte werden von *mobiled* durch Befragung von genPath ermittelt.

Der im Antwortpaket enthaltene Status ist 0 wenn der Roboter bisher im Status *idle* war. Er wechselt dann in den Status *movementSetup* und schließlich *moving*. Der Status ist -1, wenn sich der Roboter vorher nicht im Zustand *idle* befand; die Bewegung wird dann nicht ausgeführt.

A.3.4.5 MOVESPLINEALONG – Weiche Bewegung durch angegebene Wegpunkte

Kommando	Argument	Erklärung
0x00030004	foreach(Wegpunkt) {	
	WEGPUNKT-X	F32, X-Koordinate des Wegpunkts
	WEGPUNKT-Y	F32, Y-Koordinate des Wegpunkts
	}	
	POSE-WINKEL	F32, Winkel der Zielpose
0x80030004	STATUS	S32

MOVESPLINE fordert *mobiled* auf, eine Spline-Bewegung von der aktuellen Roboterpose durch die angegebenen Wegpunkte durchzuführen. Anschließend rotiert der Roboter in den angegebenen Zielwinkel.

Der im Antwortpaket enthaltene Status ist 0 wenn der Roboter bisher im Status *idle* war. Er wechselt dann in den Status *movementSetup* und schließlich *moving*. Der Status ist -1, wenn sich der Roboter vorher nicht im Zustand *idle* befand; die Bewegung wird dann nicht ausgeführt.

Der erste übergebene Wegpunkt muss der aktuellen Position des Roboters entsprechen.

A.3.4.6 WAITFORCOMPLETED – Benachrichtigung bei Bewegungsende

Kommando	Argument	Erklärung
0x00030005	-	
0x80030005	STATUS	S32, Sollte stets 0 sein

WAITFORCOMPLETED fordert *mobiled* auf, ein Antwortpaket zu senden, sobald der Roboter den Zustand *idle* erreicht hat.

A.3.5 Laserscanner-Kommandos

Folgende Kommandos ermöglichen eine Verwaltung der Laserscanner

A.3.5.1 GETNUMSCANNERS – Ermitteln der Anzahl verfügbarer Scanner

Kommando	Argument	Erklärung
0x00040000	-	-
0x80040000	STATUS	S32, Sollte stets 0 sein
	ANZAHL	U32, Anzahl der Laserscanner

GETNUMSCANNERS fordert *mobiled* auf, ein Paket mit der Anzahl der verfügbaren Laserscanner zu schicken.

A.3.5.2 GETSCANNERPOSE – Abfrage einer Scannerpose

Kommando	Argument	Erklärung
0x00040001	SCANNERNUMMER	U32, Nummer des betreffenden Scanner
0x80040001	STATUS	S32
	POSE-X	F32, X-Koordinate der Pose
	POSE-Y	F32, Y-Koordinate der Pose
	POSE-WINKEL	F32, Winkel der Pose

GETPOSE fordert *mobiled* auf, ein Antwortpaket mit der Pose des entsprechenden Scanners zu senden. Diese Pose enthält Werte in Plattformkoordinaten. Der erste Scanner hat die Nummer 0. Status ist 0 oder -1 wenn für die angegebene Scannernummer keine Pose vorhanden ist.

A.3.5.3 GETSCANRADIALSCANNER – Abfrage von Scandaten

Kommando	Argument	Erklärung
0x00040002	SCANNERNUMMER	U32, Nummer des betreffenden Scanner
0x80040002	STATUS	S32
	foreach(Scanwert) {	
	DISTANZ	F32, Distanz zwischen Scanner und Hindernis
	}	

GETSCANRADIALSCANNER fordert *mobiled* auf, ein Antwortpaket mit den Scandaten des angegebenen Scanners zu senden. Status ist genau dann 0, wenn *mobiled* Scandaten ermitteln konnte, sonst -1.

Der erste Wert entspricht der ersten Scannermessung. Der Scanner misst in einem Halbkreis von rechts nach links. Die Scanwerte werden alle 0.5 Grad ermittelt.

A.3.6 Kollisionsvermeidungs-Kommandos

Folgende Kommandos ermöglichen die Steuerung der Kollisionsvermeidung

A.3.6.1 GETCOLLISIONAVOIDANCE – Ermitteln des Zustands der Kollisionsvermeidung

Kommando	Argument	Erklärung
0x00050000	-	-
0x80050000	STATUS	S32, Sollte stets 0 sein
	ACTIVATION	U32, Aktivierungszustand

GETCOLLISIONAVOIDANCE fordert *mobiled* auf, mit einem Paket mit dem Aktivierungszustand der Kollisionsvermeidung zu antworten. Ist die Kollisionsvermeidung aktiv, ist der Wert von ACTIVATION 1, sonst 0.

A.3.6.2 SETCOLLISIONAVOIDANCE – Setzen des Zustands der Kollisionsvermeidung

Kommando	Argument	Erklärung
0x00050001	ACTIVATION	U32, Aktivierungszustand
0x80050001	STATUS	S32, Sollte stets 0 sein

`SETCOLLISIONAVOIDANCE` fordert *mobiled* auf, den im Paket übergebenen Zustand für die eigene Kollisionsvermeidung zu übernehmen. Ein Wert von 1 steht für eine aktive Kollisionsvermeidung, 0 für eine inaktive.

A.3.7 Konfigurations-Kommandos

Folgende Kommandos ermöglichen die Konfiguration von *mobiled* über das Netzwerk.

A.3.7.1 GETSPEEDFACTOR – Ermitteln des Geschwindigkeitsmultiplikators

Kommando	Argument	Erklärung
0x00060000	-	-
0x80060000	STATUS	S32, Sollte stets 0 sein
	SPEEDFACTOR	F32, aktueller Geschwindigkeitsmultiplikator

`GETSPEEDFACTOR` fordert *mobiled* auf, mit einem Paket mit dem aktuellen Geschwindigkeitsmultiplikator zu antworten. Hat der Geschwindigkeitsmultiplikator einen Wert von 1.0, fährt der Roboter auf geraden Strecken mit der Höchstgeschwindigkeit.

A.3.7.2 SETSPEEDFACTOR – Setzen des Geschwindigkeitsmultiplikators

Kommando	Argument	Erklärung
0x00060001	SPEEDFACTOR	F32, gewünschter Geschwindigkeitsmultiplikator
0x80060001	STATUS	S32

`SETSPEEDFACTOR` fordert *mobiled* auf, den übergebenen Geschwindigkeitsmultiplikator zu übernehmen. Der Wert muss zwischen 0 und 1 liegen. Status ist genau dann 0, wenn ein legaler Wert gefunden und übernommen wurde, sonst -1.

A.3.7.3 GETSPEEDLIMIT – Ermitteln der Geschwindigkeitsbegrenzung

Kommando	Argument	Erklärung
0x00060002	-	-
0x80060002	STATUS	S32, Sollte stets 0 sein
	SPEEDLIMIT	F32, aktuelle Geschwindigkeitsbegrenzung

`GETSPEEDLIMIT` fordert *mobiled* auf, mit einem Paket mit der aktuellen Geschwindigkeitsbegrenzung zu antworten. Die Einheit der Geschwindigkeitsbegrenzung ist Meter pro Sekunde.

A.3.7.4 SETSPEEDLIMIT – Setzen der Geschwindigkeitsbegrenzung

Kommando	Argument	Erklärung
0x00060003	SPEEDLIMIT	F32, gewünschte Geschwindigkeitsbegrenzung
0x80060003	STATUS	S32, Sollte stets 0 sein

`SETSPEEDLIMIT` fordert *mobiled* auf, die übergebene Geschwindigkeitsbegrenzung zu übernehmen. Der Wert muss positiv sein und sollte einen realistischen Wert von ca. 1,5m/s nicht überschreiten.

A.3.7.5 GETABORTMOTIONDECELERATION – Ermitteln der Bremsentschleunigung

Kommando	Argument	Erklärung
0x00060004	-	-
0x80060004	STATUS	S32, Sollte stets 0 sein
	ABORTMOTIONDECELERATION	F32, aktuelle Bremsentschleunigung

`GETABORTMOTIONDECELERATION` fordert *mobiled* auf, mit einem Paket mit der aktuellen Bremsentschleunigung zu antworten. Die Einheit der Bremsentschleunigung ist Meter/Sekunde^2.

A.3.7.6 SETABORTMOTIONDECELERATION – Setzen der Bremsentschleunigung

Kommando	Argument	Erklärung
0x00060005	ABORTMOTIONDECELERATION	F32, gewünschte Bremsentschleunigung
0x80060005	STATUS	S32

`SETABORTMOTIONDECELERATION` fordert *mobiled* auf, die übergebene Bremsentschleunigung zu übernehmen. Der Wert muss positiv sein. Die Einheit der Bremsentschleunigung ist Meter/Sekunde^2.

A.3.7.7 GETSMALLESTHIGHSPEEDCURVERADIUS – Ermitteln des kleinsten Kurvenradius mit Höchstgeschwindigkeit

Kommando	Argument	Erklärung
0x00060006	-	-
0x80060006	STATUS	S32, Sollte stets 0 sein
	SMALLESTHIGHSPEEDCURVERADIUS	F32, aktueller kleinster Kurvenradius

`GETSMALLESTHIGHSPEEDCURVERADIUS` fordert *mobiled* auf, mit einem Paket mit dem aktuellen kleinsten Kurvenradius mit Höchstgeschwindigkeit zu antworten. Die Einheit ist Meter.

A.3.7.8 SETSMALLESTHIGHSPEEDCURVERADIUS – Setzen des kleinsten Kurvenradius mit Höchstgeschwindigkeit

Kommando	Argument	Erklärung
0x00060007	SMALLESTHIGHSPEEDCURVERADIUS	F32, gewünschter kleinster Kurvenradius
0x80060007	STATUS	S32

`SETSMALLESTHIGHSPEEDCURVERADIUS` fordert *mobiled* auf, den übergebenen kleinsten Kurvenradius mit Höchstgeschwindigkeit zu übernehmen. Der Wert muss positiv sein. Die Einheit ist Meter.

A.3.7.9 GETROBOTRADIUS – Ermitteln des Roboterradius

Kommando	Argument	Erklärung
0x00060008	-	-
0x80060008	STATUS	S32, Sollte stets 0 sein
	ROBOTRADIUS	F32, aktueller Roboterradius

`GETROBOTRADIUS` fordert *mobiled* auf, mit einem Paket mit dem aktuellen Roboterradius zu antworten. Die Einheit des Roboterradius ist Meter. Dieser Wert hat Einwirkung auf die Pfadplanung.

A.3.7.10 SETROBOTRADIUS – Setzen des Roboterradius

Kommando	Argument	Erklärung
0x00060009	ROBOTRADIUS	F32, gewünschter Roboterradius
0x80060009	STATUS	S32

`SETROBOTRADIUS` fordert *mobiled* auf, den übergebenen Roboterradius zu übernehmen. Der Wert muss positiv sein. Die Einheit des Roboterradius ist Meter.

A.3.7.11 GETPATHPLANNERUPDATEINTERVAL – Ermitteln des Fahrtplaner-Aktualisierungsintervalls

Kommando	Argument	Erklärung
0x00060010	-	-
0x80060010	STATUS	S32, Sollte stets 0 sein
	PATHPLANNERUPDATEINTERVAL	U32, aktueller Aktualisierungsintervall

`GETPATHPLANNERUPDATEINTERVAL` fordert *mobiled* auf, mit einem Paket mit dem aktuellen Fahrtplaner-Aktualisierungsintervall zu antworten. Die Einheit des Intervalls ist Millisekunden. Findet der Fahrtplaner keinen Pfad, fragt *mobiled* nach Ablauf dieses Intervalls erneut an.

A.3.7.12 SETPATHPLANNERPOLLINTERVAL – Setzen des Fahrtplaner-Aktualisierungsintervalls

Kommando	Argument	Erklärung
0x00060011	PATHPLANNERUPDATEINTERVAL	U32, gewünschter Aktualisierungsintervall
0x80060011	STATUS	S32

`SETPATHPLANNERPOLLINTERVAL` fordert *mobiled* auf, den übergebenen Fahrtplaner-Aktualisierungsintervall zu übernehmen. Der Wert muss positiv sein. Die Einheit des Fahrtplaner-Aktualisierungsintervalls ist Millisekunden.

B Danksagungen

Zu guter Letzt möchte ich mich bei den Personen bedanken, die mir die Erstellung dieser Arbeit ermöglicht oder erleichtert haben.

Herrn Prof. Dr. Jianwei Zhang danke ich für das mir entgegengebrachte Vertrauen. Dies zeigte sich für mich einerseits darin, dass er mir einen entsetzlich teuren Roboter zur Verfügung stellte - wohlwissend, dass es bei unvorsichtiger Handhabung zu großem Sach- oder gar Personenschaden kommen kann. Andererseits darin, dass er mir ermöglichte, sehr eigenständig zu arbeiten.

Für die Übernahme des Zweitgutachtens und der wirklich sehr hilfreichen Anregungen zu Beginn dieser Arbeit danke ich Herrn Prof. Werner Hansmann ganz herzlich.

Daniel Westhoff, der Dank an Dich reicht gleich mehrere Jahre zurück. Hätte das Serviceroboter-Projekt im Jahr 2004 nicht mein Interesse an der Thematik geweckt, wären die vorigen 102 Seiten wohl deutlich langweiliger ausgefallen. Auch während der Arbeit warst Du mein erster Ansprechpartner bei Fragen zur alten *mobiled*-Version, Kalman-Filtern und mobiler Robotik allgemein.

Denis Klimentjew, mit Dir assoziiere ich nicht nur freundliche Pöbeleien im Robotiklabor, sondern auch eine ungeheure Hilfsbereitschaft und viele wertvolle Hinweise.

Martin Weser, vielen Dank für Deinen Fahrtplaner! Er funktioniert!

Hagen Stanek danke ich für ein Ausmaß an Hilfe, dass man Unbekannten normalerweise nicht angedeihen lassen würde. Er hat einige Teile der genRob-Architektur für meine Zwecke erweitert, andere mir erläutert.

Bernd Schütz - dem Sonnenschein am Ende des Flurs - bin ich dankbar, weil sich meine Stimmung automatisch verbessert, wenn er nur an der Tür vorbeigeht. Und da das Robotik-Labor gleich vor seinem Büro liegt, passiert das gar nicht so selten.

Hannes Bistry, zuerst möchte ich Dir ein großes Lob für das Produkt Deiner Diplomarbeit aussprechen. Der Kasten funktioniert zuverlässig, ist einfach zu verwenden, sehr gut dokumentiert und schindet mit seinen vielen Lämpchen echt Eindruck! Danken möchte ich Dir aber auch für Dein Interesse an meiner Arbeit, hilfreiche Ideen sowie Deinen Beistand und Dein Fahrrad; beide zusammen haben mir aus der Patsche geholfen.

Allen anderen Mitarbeitern, Studenten, Doktoranden, und was bei TAMS sonst noch so über den Flur läuft, danke ich für das nette Klima im Arbeitskreis!

Meinem Bruder Guido danke ich, weil er mir trotz seiner Überqualifizierung außerhalb gewerkschaftlich abgesegneter Arbeitszeiten mehrere Stunden mit bibtex geholfen hat.

Meine Schwester Henny hat mir Essen vorbeigebracht, als ich schon vergessen hatte, wo in meiner Wohnung die Tür nach draußen ist. Herzlichen Dank!

Alesig, herzlichen Dank auch an Dich. Für alles.

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe.

Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereichs einverstanden.