Masterarbeit

Pfadplanung für Aufgaben in der realen Welt ausgeführt durch reale Robotersysteme

Masterarbeit

Daniel Leidner

DLR

Institut für Robotik und Mechatronik	BJ.: 2011
	IB.Nr.: 515-11-02

MASTERARBEIT

PFADPLANUNG FÜR AUFGABEN IN DER REALEN WELT AUSGEFÜHRT DURCH REALE ROBOTERSYSTEME

Freigabe:	Der Bearbeiter:	Unterschriften
	Daniel Leidner	D. Leidner
	Betreuer:	
	Franziska Zacharias	F. Zacharias
	Der Institutsdirektor	
	Prof. Dr. G. Hirzinger	G. Hirzinger

Dieser Bericht enthält 92 Seiten, 37 Abbildungen und 5 Tabellen

Ort: Oberpfaffenhofen	Datum: 28.3.11	Bearbeiter:	Zeichen:

Deutsches Zentrum
für Luft- und Raumfahrt e.V.
DLR

hochschule mannheim

Masterarbeit

Pfadplanung für Aufgaben in der realen Welt ausgeführt durch reale Robotersysteme

Daniel Leidner

28. März 2011

Eingereicht am 28. März 2011
von Daniel Leidner

Masterstudiengang Informationstechnik
Hochschule Mannheim

Prüfer: Prof. Dr. Thomas Ihme,
Dipl.-Inform. Franziska Zacharias

Betreuer: Dipl.-Inform. Franziska Zacharias,
Florian Schmidt, MSc

Institut für Robotik und Mechatronik
Deutsches Zentrum für Luft- und Raumfahrt e.V.

Zusammenfassung

Autonome Bewegung ist essentiell für die Erfüllung einer Vielzahl von Manipulationsaufgaben. Speziell in der Servicerobotik ist es wichtig, Bewegungen in einer angemessen Zeit zu planen und dabei die Dynamik des Roboters zu berücksichtigen. In dieser Arbeit wird eine generische Softwarearchitektur zur Parallelisierung verschiedener Elemente der Pfadplanung vorgestellt. Zusätzlich wird der Pfad um Abstandsinformationen zwischen dem Roboter und dessen Umgebung erweitert, welche dann zur Anpassung der Beschleunigungen und Geschwindigkeiten des realen Roboters verwendet werden. Es wird gezeigt, dass die Parallelisierung des kompletten Pfadplanungsvorganges die schnelle Generierung von Trajektorien erlaubt, welche direkt auf dem realen Robotersystem verwendet werden können.

Abstract

Autonomous motion is an essential component for dealing with a wide variety of manipulation tasks. Especially in service robotics, it is important to plan the movements in an appropriate amount of time and take into account the dynamics of the robot. In this work a generic software architecture to parallelize different elements of path planning is introduced. Additionally, the path is augmented with clearance information between the robot and its environment that is then used to adjust the accelerations and velocities of the real robot's motion. It is shown that parallelization of the complete planning process enables the rapid generation of trajectories that can be used directly on the real robot system.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit

„Pfadplanung für Aufgaben in der realen Welt ausgeführt durch reale Robotersysteme“

selbständig und ohne Benutzung anderer als die angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht worden. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Oberpfaffenhofen, den 28. März 2011

Daniel Leidner

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr. Thomas Ihme für die Betreuung von Seiten der Hochschule bedanken. Von Seiten des Instituts für Robotik des Deutschen Zentrums für Luft- und Raumfahrt möchte ich besonders zwei Personen danken: Franziska Zacharias für die engagierte administrative und fachliche Unterstützung und Florian Schmidt für die zahlreichen technischen Hilfestellungen. Des Weiteren möchte ich mich bei meinen Eltern, für die finanzielle Unterstützung während des Studiums und meiner Freundin, die mich während meiner Zeit in München begleitet hat, bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Heutige humanoide Serviceroboter vereinen eine Vielzahl technischer Innovationen in sich. Leichtbaukonstruktionen, leistungsfähige Antriebe und hoch akkurate Sensoren bilden dabei nur das äußere Erscheinungsbild des Roboters. Mit der zunehmenden Komplexität der heutigen Roboter erweitert sich jedoch auch deren Tätigkeitsbereich. Dies treibt die Entwicklung immer effizienterer Softwarelösungen voran. Autonomie, Perzeption, und Kognition zählen dabei zu den Kernaspekten der Forschung. Ein wichtiges Ziel zur Umsetzung einer gewissen Autonomie ist dabei die *Fähigkeit zur autonomen Manipulation*. Der zukünftige Einsatz von Servicerobotern in Alltagsumgebungen führt somit schließlich zur immer engeren Verknüpfung zwischen Mensch und Maschine. Mit *Rollin' Justin* wird bereits seit einigen Jahren ein Roboter entwickelt, der sich in diese Richtung entwickeln soll. Mit seinem antropomorphen Auftreten ist es ihm möglich komplexe Aufgaben in der Umgebung des Menschen zu lösen.

Abbildung 1.1: In Zukunft können Serviceroboter wie Justin Schlagzeug spielen oder noch komplexere Fähigkeiten erlangen und somit vielseitig eingesetzt werden.

1.1 Motivation

Autonome Pfadplanung ist bis heute noch nicht sehr weit in der Robotik verbreitet. Besonders deren Einsatz in realen Robotersystemen ist bisher kaum vorangeschritten. Dabei hat die autonome Pfadplanung enormes Potential, besonders im Bereich der Servicerobotik in der mobile Roboter meist über Arme mit Händen und einen sehr großen Aktionsradius verfügen. Auch für antropomorphe Systeme mit zwei Armen und einem vielseitigen Oberkörper kann Pfadplanung Trajektorien generieren, bei denen beide Manipulatoren kooperativ arbeiten. Zusätzlich kann der Torso dazu verwendet werden, den erreichbaren Arbeitsbereich des Roboters zu vergrößern, sodass auch weiter entfernte Ziele direkt mithilfe der Pfadplanung angesteuert werden können. Leider steigt mit der wachsenden Komplexität eines Roboters auch der Aufwand zur Berechnung kollisionsfreier Trajektorien. Die Planung erfolgt daher meist offline und kann nicht in Live-Labordemonstrationen eingesetzt werden. Mit der zunehmenden Verbreitung von *Mehrkernprozessoren* und der Idee des *Cloud Computing* kann dem Rechenaufwand jedoch entgegen gesteuert werden. Selbst unter der zusätzlichen Bedingung, die Dynamik des Roboters zu berücksichtigen, kann so effizient Pfadplanung betrieben werden. Verteilte und parallele Architekturen können dazu beitragen, dass auch die autonome Pfadplanung in Zukunft auf realen Robotersystemen verwendet werden kann. Der Schritt zur parallelisierten Pfadplanung ist notwendig um die Autonomie heutiger Serviceroboter zu erhöhen.

1.2 Zielsetzung

Pfadplanungsalgorithmen werden oft nur in der Simulation entwickelt und nicht auf realen Robotersystemen eingesetzt. Versucht man allerdings die in der Simulation entwickelten Algorithmen im realen Betrieb zu testen, treten oft unvorhergesehene Probleme auf. Vor allem die unzureichende Modellierung der Umgebung und fehlende Berücksichtigung der Dynamikmodelle des Roboters führen häufig zu Fehlern bei der Ausführung in der Realität. Zusätzlich treten bei der Pfadplanung für komplexe humanoide Roboter in realistisch modellierten Umgebungen immer noch sehr hohe Planungszeiten auf. Dies führt dazu, dass Planungsalgorithmen noch immer nicht im regulären Betrieb auf Robotern vorhanden sind. Im Rahmen dieser Masterarbeit soll untersucht werden, wie ein bestehendes Planungssystem für den Einsatz unter realen Bedingungen modifiziert werden kann. Dazu sollen Algorithmen entworfen werden, die durch parallele Ausführung einen entscheidenden Geschwindigkeitsvorsprung gegenüber deren sequentiellen Vorbildern erhalten. Zusätzlich sollen Informationen über den minimalen Abstand des Roboters zu seiner Umgebung dazu genutzt werden, die Geschwindigkeit des Roboters nahe Objekten zu limitieren. Ziel der Limitierung ist die Vermeidung von zu hohen Beschleunigungen, die zu Überschwingern der Leichtbaukonstruktion des Roboters führen. Die entwickelten Algorithmen sollen auf dem humanoiden Roboter *Rollin' Justin* verifiziert werden.

2 Grundlagen und der aktuelle Stand der Forschung

Zu Beginn dieser Arbeit werden einige Grundlagen der Robotik erläutert um den Einstieg in den aktuellen Stand der Forschung im Bereich der robotischen Manipulationsplanung und Pfadplanung zu erleichtern. Dazu werden bereits etablierte Pfadplanungs- und Pfadoptimierungsalgorithmen untersucht. Der Fokus dieser Arbeit liegt auf der Verkürzung der Rechenzeit von Pfadplanungsalgorithmen unter Verwendung von verteilten Systemen oder Mehrprozessorsystemen. Die in diesem Kapitel beschriebenen Grundlagen zur Robotik, die darauf aufbauenden Pfadplanungsalgorithmen und der Aufbau des humanoiden Roboters Justin, basieren auf den entsprechenden Abschnitten einer früheren Arbeit [27].

2.1 Grundlagen der Robotik

Dieser Abschnitt dient als Einstieg in den aktuellen Stand der Forschung. Es werden die Grundlagen der *Manipulation* und der *Pfadplanung* erläutert. Weiterhin wird ein Einblick in *Mehrprozessorsysteme* und *verteilte Systeme* gegeben.

2.1.1 Koordinatensysteme und Transformationen

Die Position eines Roboters lässt sich in mehreren Koordinatensystemen bestimmen. Das Bekannteste dabei ist das kartesische Koordinatensystem. Mittels x-, y- und z-Koordinaten lässt sich ein Punkt im Raum definieren. Die Orientierung eines Objektes kann durch Rotationen um eine definierte Hauptachse dargestellt werden. Eine so beschriebene Pose reicht aus, um die genaue Lage des *Tool Center Points (TCP)*, zum Beispiel des Mittelpunkts des Endeffektors, im Raum zu bestimmen. Allerdings können damit nur begrenzt Schlüsse auf die zugehörigen Gelenke des Manipulators gezogen werden. Der *Konfigurationsraum (C_{space})* wird definiert durch die Menge aller realisierbaren Gelenkwinkelstellungen eines Roboters [26]. Der Konfigurationsraum besteht bei einem Roboter mit n Freiheitsgraden *(degrees of freedom, DOF)* aus n Dimensionen. Die Pose des Endeffektors kann, ausgehend von den bekannten Gelenkwinkeln, mit der Vorwärtskinematik des Roboters berechnet werden [30]. Die Inverskinematik bildet das Gegenstück zur Vorwärtskinematik. Mit ihr lassen sich Posen des Endeffektors in Gelenkwinkelstellungen des Roboters überführen. Dies wird sehr häufig gebraucht, da meist bekannt ist *wo* ein Objekt gegriffen werden soll, aber nicht mit welcher Stellung seiner Gelenke der Roboter dorthin gelangt.

2.1.2 Manipulationsplanung und Pfadplanung

Manipulationsplanung beschäftigt sich mit der Frage, wie Objekte verschiedenster Art von einem Roboter im Raum bewegt werden können. Dabei bezieht sich das Wort *Manipulation* auf die bewusste Veränderung der Umgebung des Roboters durch Bewegung der umliegenden Objekte. Eine Manipulation ist dabei eine Aneinanderreihung von mehreren Bewegungsabläufen zum Lösen einer Manipulationsaufgabe. Bewegungen oder auch Pfade, die zu einem Objekt hinführen, werden als Transitpfade bezeichnet. Pfade auf denen ein Objekt bewegt wird, werden Transferpfade genannt. Die Planung der einzelnen Schritte einer Aufgabe heißt im Forschungsbereich der künstlichen Intelligenz Aufgabenplanung. Die Planung der Bewegung eines Roboters erfolgt durch Pfadplanung [38]. Die bei der Pfadplanung gewonnenen Pfade bestehen aus mehreren einzelnen Konfigurationen des Roboters. Ziel der Pfadplanung ist es, einen realisierbaren, nicht kollidierenden Weg von einer Startkonfiguration q_{init} des Roboters zu einer Zielkonfiguration q_{goal} zu finden. Dabei ist es wünschenswert, dass der Pfad zur Laufzeit, also zeitnah, erstellt wird. So kann in einem gewissen Rahmen auf Ereignisse in der Umgebung reagiert werden. Bei der Pfadplanung wird zwischen lokalen und globalen Bahnplanern unterschieden. Ein lokaler Planer überprüft nur Verbindungen von Konfigurationen im unmittelbaren Umfeld eines gegebenen Zwischenergebnisses. Dabei werden lediglich einfache Wege gesucht. Im besten Fall ist dies eine kollisionsfreie Gerade im Konfigurationsraum. Sollte doch eine Kollision existieren, so wird versucht diese lokal zu umgehen. Da dies nicht immer möglich ist, verharren lokale Planer oft in lokalen Minima. Globale Planer hingegen betrachten das Gesamtproblem. Hierbei wird versucht, über alle möglichen Konfigurationen des erreichbaren Raumes einen realisierbaren Pfad zur Zielkonfiguration zu finden. Um Hindernisse und Engstellen zu umgehen, werden im Gegensatz zu lokalen Planern alternative Konfigurationen im gesamten C_{space} gesucht.

2.1.3 Mehrprozessorsysteme und verteilte Systeme

Sowohl *Mehrprozessorsysteme* als auch *verteilte Systeme* sind heutzutage weit verbreitet. Die meisten aktuell erhältlichen Heimcomputer besitzen ein Mehrprozessorsystem, beziehungsweise ein *Mehrkernsystem*. Eine oft genutzte verteilte Systemarchitektur ist die Client-Server Struktur. Diese wird typischerweise bei zentral gelagerten Daten oder Ressourcen benutzt. Im Zuge der immer günstiger werdenden Rechenleistung und der gleichzeitigen Miniaturisierung, sind auch in erste mobile Robotikplattformen mehrere leistungsfähige Rechner integriert [37].

2.1.3.1 Architektur und Programmierung von Mehrprozessorsysteme und verteilten Systeme

Mehrprozessorsysteme besitzen mehrere Prozessoren auf einem Mainboard. Die jeweiligen Prozessoren haben dabei oftmals mehrere Rechenkerne auf einem Chip integriert. Alle Prozessoren und Kerne können dabei auf den gleichen Hauptspeicher zugreifen [35]. Ein verteiltes System besteht dagegen aus mehreren separaten Computern, die durch ein Netzwerk miteinander verbunden sind und sich somit keinen Arbeitsspeicher teilen [2]. Die Kommunikation kann daher ausschließlich über das Netzwerk ablaufen. Oft wird ein Algorithmus so über mehrere Computer hinweg implementiert. In einem geeigneten Netzwerk können auch Programmteile über mehrere Systeme verteilt werden. Bei der Kommunikation über Rechnergrenzen hinaus können die bekannten Protokolle *TCP/IP* oder *UDP/IP* verwendet werden. Eine weitere Eigenschaft von verteilten Systemen und Mehrprozessorsystemen ist die Synchronisation. Diese ist notwendig, da sich Programme für beide Architekturen von sequentiellen Programmen durch das simultane Abarbeiten mehrerer Programmelemente unterscheiden. Dieses Vorgehen wird als *Nebenläufigkeit* bezeichnet. Dabei ist vor allem bei gemeinsam genutztem Speicher, wie es bei Mehrprozessorsystemen üblich ist, darauf zu achten, dass kein unbefugter Zugriff erfolgt. Ein unbefugter Zugriff entsteht unter anderem dann, wenn mehrere Prozesse einen Wert gleichzeitig bearbeiten. Die Änderungen von einem der Prozesse gehen dabei verloren. Dies kann durch den Einsatz einer oder mehrerer *Semaphore*[1] verhindert werden. Die Daten für ein paralleles System können zentral oder dezentral gelagert werden.

2.1.3.2 Theoretischer maximaler Geschwindigkeitsgewinn durch eine verteilte Softwarearchitektur nach dem Amdahlschen Gesetz

Bei der Kommunikation zwischen Programmen auf entfernten Computern oder auf mehreren Kernen fällt in der Regel ein *Kommunikationsoverhead* an, welcher die Abarbeitung von Problemen verlangsamt. Kommunikationsoverhead entsteht dadurch, dass die einzelnen Elemente einer verteilten Software Daten austauschen müssen und somit von einander abhängig sind. Diese Abhängigkeiten resultieren in regelmäßigen Wartezyklen der einzelnen Komponenten. Zudem müssen Daten unter Umständen auf allen beteiligten Rechnern synchronisiert werden. Aus diesen Gründen kann selten ein linearer Geschwindigkeitsgewinn beim Verteilen eines Problems auf mehrere Rechner und Prozessoren erreicht werden. Nach dem Amdahlschen Gesetz wird der Geschwindigkeitszuwachs vor allem durch den sequentiellen Anteil des Problems beschränkt, da dieser nicht parallelisierbar ist [1]. Die Steigerung ist also nicht nur abhängig von der Anzahl der CPUs, sondern auch von der Parallelisierbarkeit des Programmcodes. Der theoretische Geschwindigkeitszuwachs S lässt sich wie folgt berechnen:

[1] Ein Semaphor ist eine Datenstruktur zur Verwaltung von kooperierendem oder konkurrierendem Zugriff auf eine gemeinsam genutzte Ressource.

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

Wobei P der parallelisierbare Programmanteil in Prozent ist und N die Prozessorzahl beschreibt. Abbildung 2.1 zeigt deutlich, wie stark der Geschwindigkeitszuwachs von der Eignung des Programmcodes abhängig ist. Ein Wert von $100\%(P = 1)$ für die Parallelisierbarkeit würde einen linearen Geschwindigkeitszuwachs bedeuten. Bei

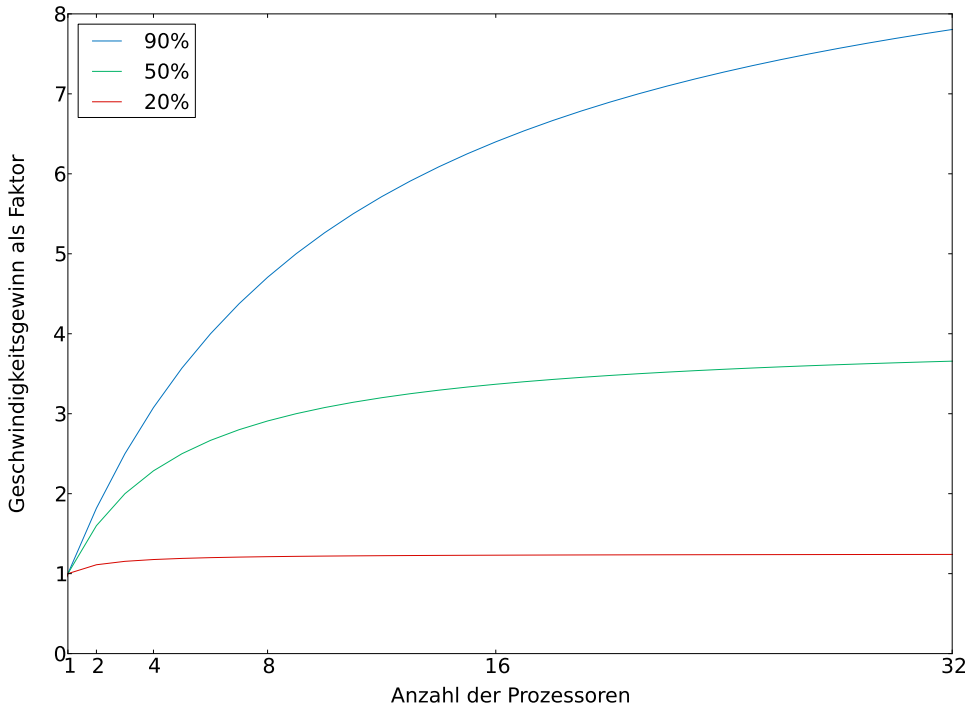


Abbildung 2.1: Theoretischer, maximaler Geschwindigkeitszuwachs nach dem Amdahlschen Gesetz. Die Farben codieren die Parallelisierbarkeit eines Programms. Das blaue Programm kann zu 90% parallelisiert werden, das grüne zu 50% und das rote nur zu 20%.

$90\%(P = 0.9)$ Parallelisierbarkeit erreicht man mit 32 Kernen jedoch noch nicht einmal die achtfache Steigerung. Es ist nicht einfach den genauen Prozentsatz der parallelen und sequentiellen Programmabschnitte zu berechnen, da viele unbekannte Faktoren wie Kommunikation, Speicherzugriff oder Ein- und Ausgabe nicht genau bestimmt werden können [32]. Oft verhalten sich die einzelnen Programmabschnitte zudem nicht linear. Dennoch ist das Amdahlsche Gesetz ein guter Schätzwert für die maximal notwendige Prozessorzahl.

2.2 Stand der Forschung

Eine grundlegende Aufgabe der Robotik ist es, die Bedienung von robotischen Maschinen so einfach wie möglich zu gestalten [26]. Bestenfalls soll der Roboter nur mittels einer abstrakten Kontrollschnittstelle, wie zum Beispiel einer Spracheingabe, gesteuert werden. Die Befehle beschränken sich dabei auf einfache als Aufforderung gestellte Aufgaben. Der Befehl „serve tea“ wird beispielsweise eindeutig durch den Roboter interpretiert und verarbeitet. Das Lösen dieser Aufgabe sollte dann vollkommen autonom geschehen. Hierzu ist es wichtig, dass der Roboter sowohl den gesamten Ablauf, als auch die in Unteraufgaben benötigten Bewegungen der einzelnen Vorgänge selbstständig plant. Zumindest in der Pfadplanung werden bereits seit einigen Jahren Ansätze der Autonomie verfolgt.

2.2.1 Algorithmen zur Pfadplanung

Der folgende Abschnitt beschäftigt sich mit einigen Algorithmen zur Pfadplanung, die in der Vergangenheit erfolgreich eingesetzt werden konnten.

2.2.1.1 Potential Field Guided Path Planning

Einer der ersten Ansätze zur Pfadplanung basiert auf der Idee von Potentialfeldern [24]. Bei *Potential Field-Guided Path Planning* wird jedes Objekt als Hindernis mit einem künstlichen Kraftfeld gesehen. Der Roboter ist dabei eine Punktmasse, die sich unter dem Einfluss der Potentialfelder durch den kartesischen Raum bewegt. Die Startkonfiguration q_{init} bildet eine Potentialquelle und wirkt abstoßend. Die Zielkonfiguration q_{goal} wird als Potentialsenke definiert und wirkt anziehend. Hindernisse erzeugen eine abstoßende Kraft. In jeder Konfiguration q wirkt die Gesamtkraft $F(q)$ eine Beschleunigung auf die Masse des Roboters aus. Daraus kann zu jedem Zeitpunkt die Kraft und das Drehmoment der einzelnen Motoren im Roboter bestimmt werden. Die erhaltenen Parameter dienen dann als Kommandos für dessen Motoren. Das Verhalten des Roboters ähnelt dann einer Verschiebung durch die Potentialfelder. Dies wird in Abbildung 2.2 noch einmal verdeutlicht. Diese Art der Planung kann zur Onlinepfadplanung verwendet werden. Bei günstigen Konstellationen der Umgebung mit wenigen bis gar keinen Hindernissen können so durchaus schnell Lösungen gefunden werden. Speziell für mobile Roboter ist dies ein erfolgversprechender Ansatz. Allerdings ist die Chance groß, dass der Planer lokale Minima der Potentialfunktion nicht überwinden kann. Im Allgemeinen ist die Potential-Feld-Methode zudem ungeeignet für Roboter mit einer hohen Anzahl von Freiheitsgraden. Die Abbildung der Potentiale im kartesischen Raum auf den n-dimensionalen Konfigurationsraum ist sehr kompliziert. In engen Passagen tendiert der Roboter zudem zunehmend zum Oszillieren, da er von allen Seiten abgestoßen wird [21]. Um dies zu umgehen, muss großer Aufwand beim Erstellen der Potentialfelder betrieben werden.

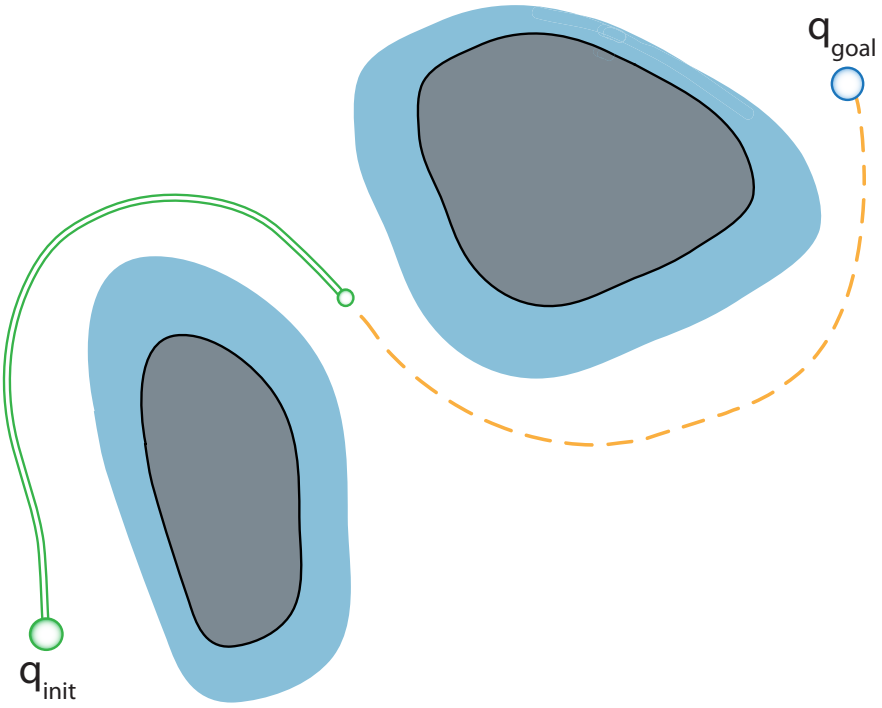


Abbildung 2.2: Roboter als Punktmasse unter dem Einfluss von Potentialfeldern.

2.2.1.2 Sampling-basierte Verfahren

Eine eigene Unterklasse der Pfadplanungsalgorithmen bilden sampling-basierte Verfahren. Dabei werden zufällig Konfigurationen aus dem kompletten C_{space} gezogen. Die Exploration des kompletten Konfigurationsraums liegt bei diesen globalen Verfahren im Vordergrund. Samplingstrategien sind darauf ausgelegt, den Konfigurationsraum so schnell wie möglich nach bestimmten Kriterien abzudecken. Ein optimales Verfahren würde in möglichst kurzer Zeit den gesamten C_{space} erkunden. Da dies bei einem Konfigurationsraum mit einer unendlichen Größe nicht realisierbar ist, muss versucht werden, an jeden Wert mit einer gewissen Abweichung heranzukommen. Erhält der Planer unbegrenzt Zeit wird immer eine Verbindung zur Zielkonfiguration gefunden sofern eine existiert. Planer mit dieser Eigenschaft werden als *Probabilistically Complete* bezeichnet. Durch Heuristiken können Samplingverfahren beschleunigt werden. Obwohl randomisierte Algorithmen nicht deterministisch sind, sind sie dennoch sehr gut geeignet für die Bahnplanung. Nach relativ kurzer Zeit kann eine Lösung gefunden werden. Die reine Planung für ein einfaches Problem kann durchaus innerhalb weniger Sekunden stattfinden. Jedoch benötigt eine solche Trajektorie unbedingt einen Optimierungsschritt um Schleifen und Umwege zu entfernen. Die Optimierungszeit kann je nach Verfahren ein Vielfaches der Planungszeit betragen. Im Folgenden werden zwei der Pfadplanungsverfahren näher betrachtet. Die Optimierungsverfahren werden gesondert in Kapitel 2.2.2 aufgeführt.

2.2.1.2.1 Probabilistic Roadmaps

Eine Variante der zufallsbasierten Bahnplanung ist das Verfahren der *Probabilistic Roadmaps* (PRM) [11]. PRMs liegt die Idee von Samplingstrategien zu Grunde. Der Pfad im C_{space} wird in zwei Schritten bestimmt. Der erste Schritt besteht darin, die sogenannte *Roadmap*, also eine Straßenkarte, zu erstellen. Diese Phase wird als Konstruktionsphase bezeichnet. Dabei werden zufällig kollisionsfreie Konfigurationen erstellt und anschließend durch einen lokalen Planer miteinander verbunden. Hierbei werden jeweils die nächsten Nachbarn untereinander verknüpft, sodass ein dichtes Netz entsteht. Das komplette Netz bildet eine kollisionsfreie Karte in welcher der Konfigurationsraum auf verschiedenen Wegen durchlaufen werden kann. Die zweite Phase ist die Anfrageauflösung. In dieser wird versucht, die Startkonfiguration q_{init} mit der Zielkonfiguration q_{goal} mittels einer Graphensuche miteinander zu verbinden. Ist dies erfolgreich, werden die einzelnen Knoten zu einem Pfad P verbunden und danach in einem Glättungsschritt in einen realisierbaren Pfad umgewandelt (Siehe Abbildung 2.3).

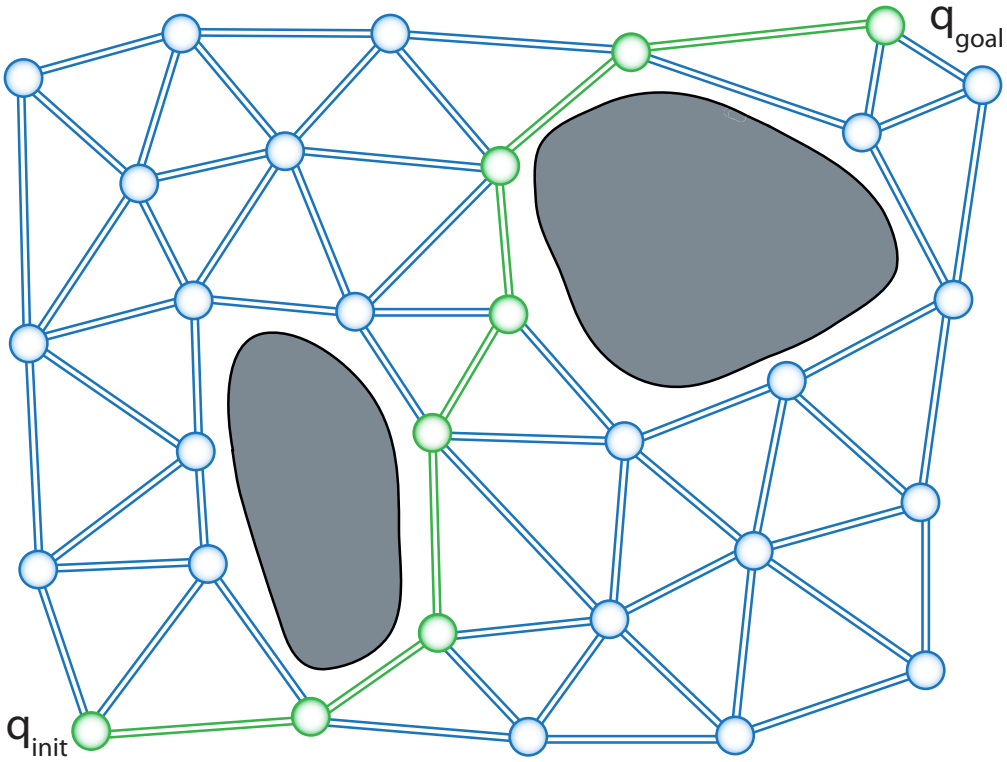


Abbildung 2.3: Erstellen einer Probabilistic Roadmap. Das blaue Netz stellt die Roadmap dar und der grüne Weg ist die kürzest mögliche Trajektorie.

Einmal erstellte Roadmaps können immer wieder zur Pfadplanung verwendet werden, solange sich die Szene nicht ändert. Sollte während der Erkundungsphase mehrmals kein Ergebnis gefunden werden, ist es möglich, die Roadmap zu erweitern. Dabei werden üblicherweise *Bias-Algorithmen*[2] verwendet, die voreingenommen gegenüber unerkundeten Bereichen agieren und diesen bei der Erkundung vorziehen. Gibt es zum Beispiel einen großen, noch unerkundeten Bereich, so wird die Suche zunächst dort fortgesetzt. Verändert sich die Umgebung in der sich der Roboter bewegt, so kann der Algorithmus um eine erneute Kollisionsabfrage zur Laufzeit erweitert werden. So kann in einem gewissen Maße auf Veränderungen der Szene eingegangen werden [5].

Probabilistic Roadmaps eignen sich hervorragend für Industrieszenarien bei denen die gleiche Aufgabe immer wieder durchgeführt werden muss. Dafür ist vor allem die detailierte Roadmap verantwortlich. Der detaillierte Aufbau der Karte nimmt sehr viel Zeit in Anspruch. Dafür können im Nachhinein umso schneller realisierbare Bahnen gefunden werden. Umgebungen, bei denen sich Elemente oft bewegen, sind hingegen nicht mit diesem Verfahren zu bewältigen, da die gesamte Karte ständig überprüft werden müsste. Das nichtdeterministische Verhalten des Kartenaufbaus erschwert zudem die Reproduzierbarkeit der Pfade und ist speziell mit Hinsicht auf Demonstrationen im Labor ein wichtiger Kritikpunkt.

2.2.1.2.2 Rapidly Exploring Random Trees

Die Klasse der *Rapidly-Exploring Random Trees* (RRT) ist eine weitere Algorithmenfamilie zur zufallsbasierten Erkundung des C_{space} [25]. Auch hier wird der Konfigurationsraum mittels Samplingalgorithmen durchsucht. Anders als bei PRMs wird bei RRTs direkt eine Verbindung zwischen den einzelnen Knoten erstellt. Alle Konfigurationen im RRT haben, ausgenommen der Wurzel, genau einen Vorgänger. Somit entsteht die typische Struktur eines Suchbaumes. Der entstehende Pfad ist im Gegensatz zu den Netzen der PRMs direkt zielführend. Erstellte Routen führen also direkt zur Zielkonfiguration q_{goal}. Der Pseudocode in Abbildung 2.4 erläutert das Vorgehen näher: In einem Iterationsschritt wird je Schritt eine neue Konfiguration q_{rand} gezogen. Die Extend-Funktion wählt dann den nächstliegenden Nachbarknoten (q_{near}) im bestehenden Baum aus und wählt einen Punkt q_{new} auf der Geraden zwischen q_{rand} und q_{near} aus. In der NewConfig Funktion überprüft der lokale Planer dabei eventuelle Kollisionen. Es können drei Situationen eintreten: *Reached*, die signalisiert, dass q_{goal} mit diesem Knoten erreicht wurde. *Advanced*, mit der ein kollisionsfreier, neuer Knoten q_{new} an den Baum angefügt wird. Und drittens $Trapped$, in dem der neue Punkt in einem kollidierenden Bereich liegt. Um eine noch schnellere Exploration des C_{space} zu erreichen, setzen viele RRT-Varianten dabei auf Bias-Algorithmen.

[2] Bias ist englisch und bedeutet Tendenz, Vorurteil oder Neigung. Ein Bias-Algorithmus handelt daher immer in Richtung einer vorgegebenen Neigung.

Auch bei Rapidly-Exploring Random Trees wird ein Optimierungsschritt benötigt, um eine Trajektorie von redundanten Bewegungen zu befreien. Die Effizienz des RRT-Algorithmus kann gesteigert werden, indem, wie in Abbildung 2.5 gezeigt, jeweils von der Start- und der Zielkonfiguration ein Suchbaum gestartet wird. Dabei wechseln sich die Bäume beim Erkunden des C_{space} stets ab, sobald der trapped Status einsetzt. Können die beiden Bäume eine Verbindung herstellen, so ist ein Weg gefunden [22].

Algorithmus 1 BuildRRT(q_{init})

```
1  T.Init(q_init)
2  for k ← 1 to K
3  do
4      q_rand ← RandomConfig()
5      Extend(T, q_rand)
```

Algorithmus 2 Extend(T, q)

```
 1  q_near ← NearestNeighbor(q, T)
 2  if NewConfig(q, q_near, q_new)
 3    then
 4          T.AddVertex(q_new)
 5          T.AddEdge(q_near, q_new)
 6          if q_new = q
 7            then
 8                 return reached
 9            else
10                 return advanced
11  return trapped
```

Abbildung 2.4: Pseudocode des RRT Algorithmus [13].

Für einmalige Anfragen, bei denen sich die Umgebung anschließend ändert, sind RRTs gut geeignet, da sie direkt zielführend sind. Ein unerkundeter Raum kann schnell exploriert werden, wodurch zeitnah ein realisierbarer Pfad gefunden wird. Anders als Probabilistic Roadmaps hinterlässt das RRT Verfahren kein ungültiges Netz nach einer Manipulation von Objekten. Bei Zielkonfigurationen in schwer erreichbaren Gebieten findet zudem der zweite Suchbaum schnell einen Weg in Richtung offenem Raum. Nichtsdestotrotz ist auch dieses Verfahren nicht im regulären Einsatz benutzbar, da die Planungszeit immer noch mehrere Sekunden dauern kann.

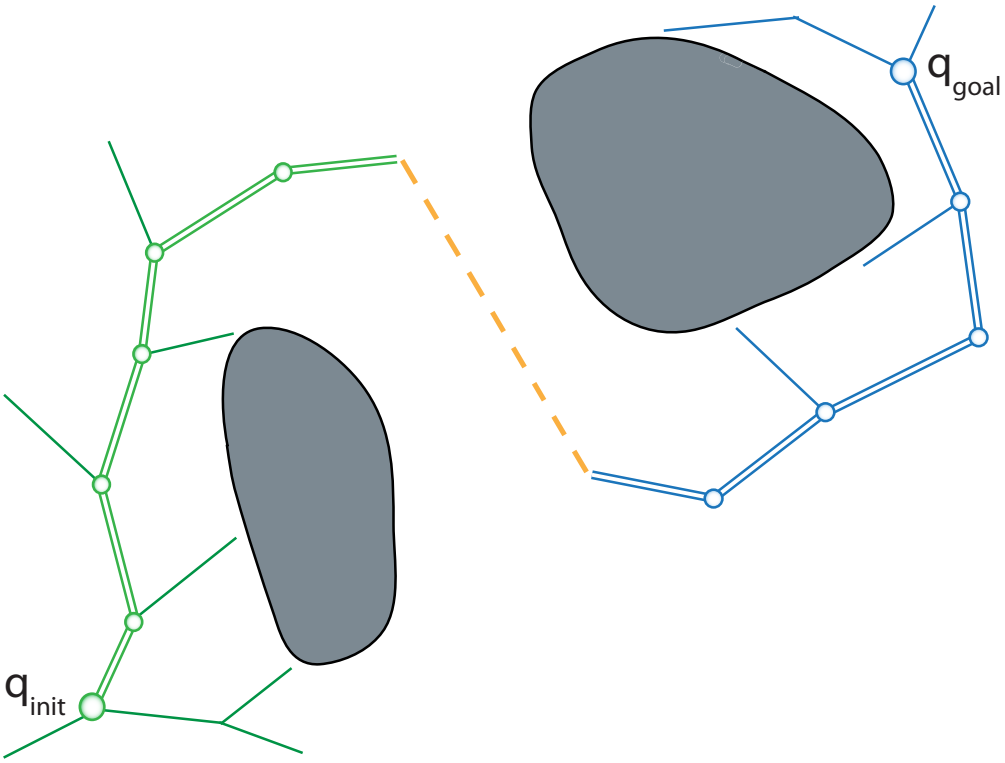


Abbildung 2.5: Erstellung eines BiRRT Suchbaumes. Der blaue Teilbaum ist der Vorwärtszweig und der grüne Teilbaum der Rückwertszweig.

2.2.2 Algorithmen zur Pfadoptimierung

Trotz ihres nicht deterministischen Verhaltens, erfreuen sich samplingbasierte Planungsverfahren enormer Beliebtheit. Dies liegt vor allem daran, dass auch für komplexe Roboter in einer relativ kurzen Zeit ein Pfad generiert werden kann. Dennoch gibt es einige negative Effekte, die ein solcher Planer mit sich bringt: Aufgrund der Samplingverfahren generieren die Planer oft Pfade niederer Qualität. Darunter fallen solche Pfade, die redundante Bewegungsabläufe beinhalten oder ruckartige Richtungsänderungen aufweisen. Dadurch werden die resultierenden Trajektorien oft unnötig lang. Um die Bewegungen zu glätten und zu kürzen wird nach der eigentlichen Planung ein weiter Schritt eingeführt: Der *Optimierungsschritt*. Ein Optimierungsschritt oder auch Glättungsschritt ist somit zwingend notwendig, braucht aber unter Umständen ein Vielfaches der Rechenzeit der eigentlichen Planung. Die Optimierung kann sich auf die bereits erwähnte Eigenschaft der Länge beziehen, oder auf andere Merkmale sowie Energieeffizienz[4] oder Natürlichkeit[41]. Das Qualitätskriterium dieser Arbeit und die im Folgenden betrachteten Verfahren beziehen sich ausschließlich auf die Länge.

Viele Verfahren setzen auf die Modifikation von gegebenen Konfigurationen oder ganzen Pfadabschnitten. Es werden also keine wesentlichen neuen Konfigurationen hinzugefügt. Dabei werden die vorhandenen Punkte so verschoben, dass die gesamte Trajektorie kürzer wird. So kann unter anderem versucht werden den *Gradientenabstieg* zu verringern, um so enger an Hindernissen vorbeizufahren [36]. Betrachtet man ein zweidimensionales Beispiel, kann der Gradient als Tangente in einem Punkt der Kurve dargestellt werden. Durch die Verringerung des Gradienten der Tangente, wird also der Pfad mehr an eine Gerade angenähert. Abbildung 2.6 verdeutlicht dieses Vorgehen. Die modifizierenden Verfahren haben den großen Nachteil, dass sie keine redundanten Bewegungen aus einem Pfad entfernen können und somit zur Reduktion der Gesamtlänge nur wenig beisteuern. Das Gradientenabstiegsverfahren wird oft für zwei-dimensionale Navigation in der mobilen Robotik verwendet. Für den Einsatz in der Bewegungsplanung eines Manipulators ist es weniger geeignet, da sich der Gradientenabstieg nur mit großem Aufwand in den Konfigurationsraum überführen lässt.

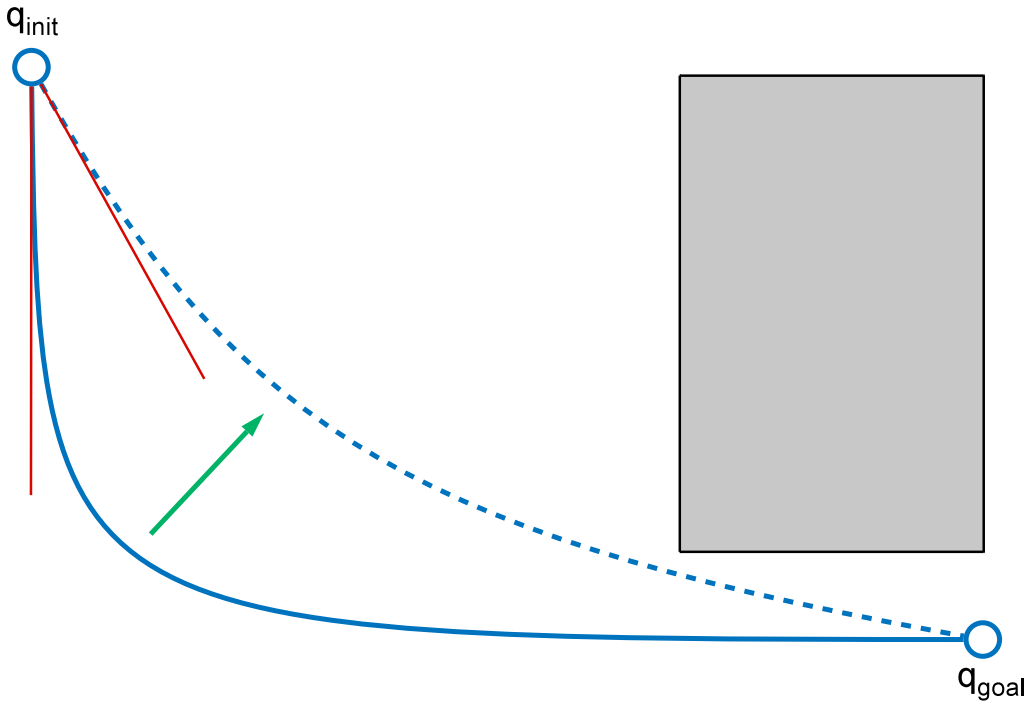


Abbildung 2.6: Optimierung mit dem Gradientenabstiegsverfahren. Der grüne Pfeil visualisiert die Änderung des Pfades. Der Anstieg der roten Tangente hat sich erheblich reduziert.

Das am häufigsten verwendete Optimierungsverfahren für die Reduzierung der Pfadlänge ist der *Linear Shortcut Optimizer*. Dieses hat gute Erfolge in der Praxis gezeigt und ist einfach zu implementieren. Der Algorithmus wird nach der eigentlichen Planungsphase ausgeführt und verläuft nach einem einfachen Schema: Zunächst werden

zwei Gelenkwinkelkonfigurationen q_A und q_B aus dem bereits bestehenden Pfad gewählt. Die Konfigurationen können dabei zufällig oder deterministisch gewählt werden [17]. Anschließend werden die zwei Knoten mit einer Linie im C_{space} verbunden. Ist der neue Pfad kollisionsfrei, so ersetzt er die bisherige Verbindung zwischen q_A und q_B. Abbildung 2.7 verdeutlicht diesen Vorgang. Eine mögliche Variation des Algorithmus verändert je Optimierungsschritt nur einen Freiheitsgrad [18], wobei hier nur Verbesserungen beim Einsatz von mobilen Robotern festgestellt wurden. Wie bei allen Optimierungsverfahren ist auch bei der Linear Shortcut Methode nicht sichergestellt, dass das tatsächliche Optimum gefunden wird. Jedoch kann nach vergleichsweise kurzer Zeit bereits ein sehr gutes Ergebnis erreicht werden.

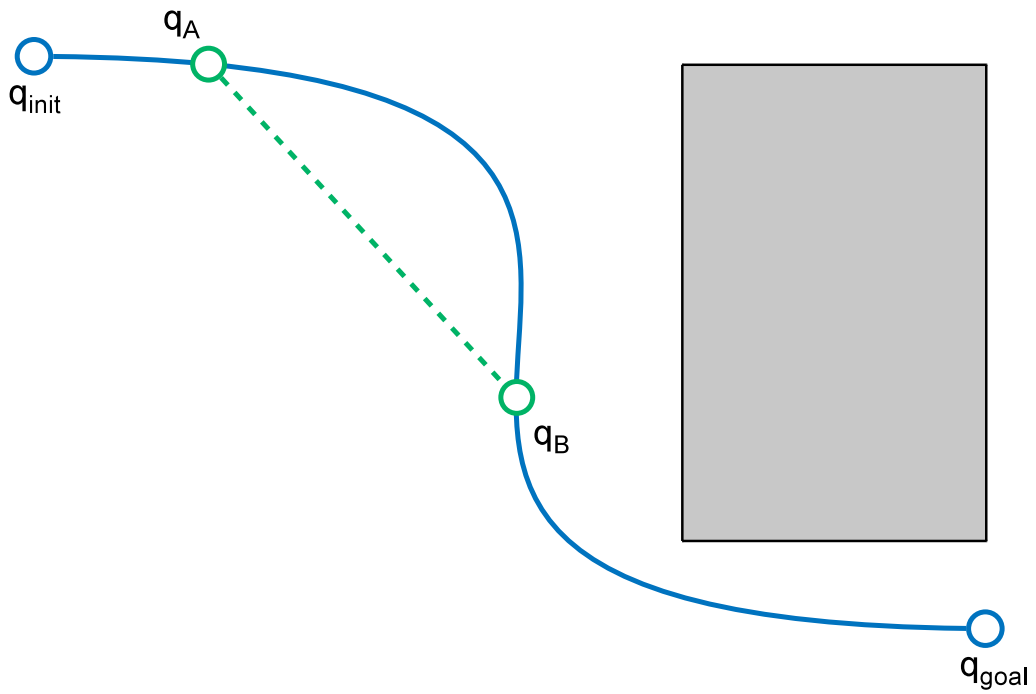


Abbildung 2.7: Der Linear Shortcut Optimization Algorithmus. Die grüne Linie verbindet die zufällig gewählten Konfigurationen q_A und q_B.

2.2.3 Verteilte Ansätze für Pfadplanung und -optimierung

Ein oft verwendeter Ansatz zur Verkürzung der Rechenzeit ist die Parallelisierung eines Algorithmus. Parallelisierung kann auch für die Algorithmen der Pfadplanung und -optimierung verwendet werden. So ist das zufallsbasierte Erstellen einer Baumstruktur für die Aufteilung auf mehrere Rechner oder Prozessoren sehr gut geeignet. Auch die Optimierung kann von einer geeigneten Parallelisierungsmethode profitieren.

2.2.3.1 Parallelisierung eines RRT Verfahrens

Bei einem zufallsbasierten Verfahren können zwei Ansätze zur Parallelisierung verfolgt werden. Die Methoden können als grob- und feinkörnige Parallelisierung betrachtet werden. Bei der grobkörnigen Methode wird das *ODER-Paradigma (OR paradigm)* ausgenutzt [9]. Dabei werden mehrere RRT Algorithmen mit der gleichen Zielkonfiguration und der gleichen Startkonfiguration gestartet. Aufgrund der stochastischen Natur des Algorithmus, wird eine der Instanzen früher eine Lösung finden. Ist dies der Fall, sendet diese Instanz eine Nachricht an die verbleibenden RRT Instanzen, damit diese ihre Suche einstellen. Dazu muss der Ablauf lediglich um eine kleine Kommunikationsroutine erweitert werden. Der erweiterte Algorithmus erreicht mit steigender Prozessorzahl nicht nur eine geringere Ausführungszeit, auch die Knotenzahl der erhaltenen Trajektorie wird zeitgleich geringer.

Algorithmus 3 BuildParallelRRT(q_{init})

```
1  T.Init(q_init)
2  while done = FALSE
3  do
4      q_rand ← RandomConfig()
5      if ParallelExtend(T_a, q_rand) ≠ Trapped
6        then
7              if ParallelExtend(T_b, q_rand) = Reached
8                then
9                      Lock(T)
10                     done ← TRUE
11                     Unlock(T)
12     Swap(T_a, T_b)
```

Algorithmus 4 ParallelExtend(T, q)

```
1  q_near ← NearestNeighbor(q, T)
2  if NewConfig(q, q_near, q_new)
3    then
4          Lock(T)
5          T.AddVertex(q_new)
6          T.AddEdge(q_near, q_new)
7          Unlock(T)
8          if q_new = q
9            then
10                 return reached
11           else
12                 return advanced
13 return trapped
```

Abbildung 2.8: Pseudocode des feinkörnigen BiRRT Algorithmus [8].

Bei der feinkörnigen Methode muss der RRT Algorithmus angepasst werden. Hierbei wird eine einzige Aufgabe auf mehrere Prozessoren verteilt [8]. Dies kann erreicht werden indem die Befehle *AddVertex* und *AddEdge* aus dem Standardalgorithmus (siehe Abbildung 2.4) als Thread neu implementiert werden. Anstelle eines einmaligen Aufrufs in der Hauptfunktion durchlaufen dann mehrere Threads eine Endlosschleife, in der jeweils ein neuer Knotenpunkt angefügt wird (siehe Abbildung 2.8). Die Threads suchen die einzelnen Knotenpunkte zwar unabhängig voneinander, fügen diese aber in einen gemeinsamen Baum ein. Um Inkonsistenz zu vermeiden muss in der *BuildRRT* Routine der Extend-Befehl dementsprechend abgewandelt werden. Eine Semaphore regelt den Zugriff auf den Baum. Eine Hilfsvariable signalisiert wenn der Algorithmus ein Ergebnis gefunden hat. Sobald dieses Signal von einem der Threads gesetzt wird, beenden sich die restlichen Threads. Da die einzelnen Threads völlig voneinander unabhängig sind, kann eine theoretisch maximale Verbesserung von annähernd 100% erreicht werden [8]. Es besteht damit nach Amdahl ein linearer Geschwindigkeitsanstieg. Bei doppelter Prozessorzahl bedeutet das eine Halbierung der Rechenzeit.

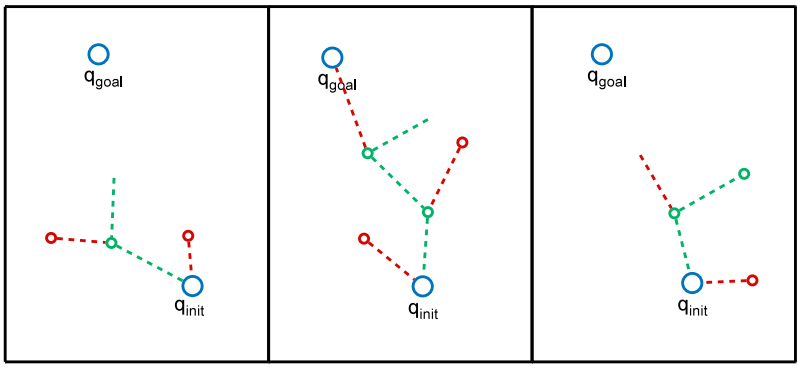


Abbildung 2.9: Paralleler Aufbau eines RRT Baumes. Drei unabhängige Instanzen des Algorithmus versuchen parallel das selbe Ziel zu erreichen. Dabei wird jeder Baum von zwei unabhängigen Prozessen (rot und grün) erweitert. Der mittlere Baum hat als erste Instanz das Ziel gefunden.

Des Weiteren ist es möglich beide Verfahren zur Parallelisierung des RRT Verfahrens zu kombinieren [8]. Zum Einen wird das ODER-Paradigma verwendet, um eine Suchanfrage mit demselben Algorithmus mehrfach simultan zu lösen. Zusätzlich arbeiten mehrere Prozessoren gleichzeitig an der Erstellung eines Suchbaumes. In Abbildung 2.9 ist ein derart kombinierter RRT Algorithmus illustriert. Mit den kombinierten Algorithmen ist ein Planungsproblem dann gelöst, wenn einer der simultan wachsenden und zugleich parallel entstehenden Bäume, das Ziel gefunden hat. Sobald beispielsweise der rote Zweig des mittleren Baumes das Ziel erreicht hat, wird der kooperative grüne Prozess beendet. Anschließend werden die übrigen beiden Suchbäume gestoppt. Das Ergebnis des mittleren Baumes dient als Grundlage für die Bewegung des Roboters. Die Teilergebnisse der beiden anderen Instanzen werden verworfen.

2.2.3.2 Parallelisierte Optimierung

Parallele Strategien zur Optimierung von Trajektorien eines Roboterarms sind nicht weit verbreitet. Lediglich zur Wegfindung mobiler Roboter werden einige verteilte Ansätze verfolgt. Die meisten dieser Algorithmen setzen dabei auf *verteilte Intelligenz*. So auch die *Ant Colony Optimization*, welche das Verhalten einer Ameisenkolonie simuliert, die einen Weg von einer Futterquelle zu ihrem Bau sucht. Hierbei erkennen die jeweiligen Agenten[3], hier „Ameisen", die *Pheromonspur* der zuvor auf einem Weg gelaufenen Agenten. Jede Ameise, die einer Pheromonspur folgt, verstärkt diese zusätzlich. Je stärker die Pheromonspur, desto anziehender ist der jeweilige Weg. So verschwinden nach und nach auf natürliche Weise die längeren Wege [12]. Dieser Algorithmus ist vor allem für die wiederholte Ausführung einer Aufgabe sinnvoll. Die zweite Methode ist die *Particle Swarm Optimization* und lässt sich auf Schwarmverhalten zurückführen. Mehrere Partikel werden auf einer Ebene verteilt und daraufhin deren Verbesserung im Bezug auf einen bekannten Punkt berechnet. Das Partikel mit dem besten Wert dient dann als Ausgangspunkt für die nächste Iteration [10]. Für Planungen im Gelenkwinkelraum sind diese Verfahren jedoch weniger geeignet, da die Umsetzung für mehrere Dimensionen sehr komplex ist. In dieser Arbeit wird stattdessen in Kapitel 5 das bewährte Linear Shortcut Verfahren auf mehrere Prozesse aufgeteilt.

2.2.4 Die Planungs- und Simulationsumgebung OpenRAVE

Eine große Herausforderung bei der Entwicklung realer Roboter ist das Testen von komplexen Modulen für verschiedenste Anwendungen. Darunter solche, die für die Bewegungsplanung und Aufgabenplanung zuständig sind. Für diesen Zweck wird eine flexible, anpassbare Entwicklungs- und Simulationsumgebung benötigt. Eine derartige Umgebung bietet OpenRAVE, ***Open** Robotics and **A**nimation **V**irtual **E**nvironment*. OpenRAVE ist eine Cross-Plattform Softwarearchitektur, die es ermöglicht, Pfadplanung für jeden erdenklichen Roboter unter Berücksichtigung der Umgebung zu berechnen und zu visualisieren.

Eine Plugin-Architektur erlaubt dem Nutzer einfach und schnell neue Komponenten wie Planer, Optimierer, Inverskinematiken oder Sensormodule zu entwerfen [14]. Die Kernapplikation, sowie die zahlreichen Plugins, sind in C++ implementiert. Unter den bereits implementierten Planern finden sich hauptsächlich RRT Verfahren, auf welche in Abschnitt 2.2.1 eingegangen wurde. Daneben finden sich noch weitere nützliche Erweiterungen, wobei der Inverskinematik Generator *ikFast* besonders hervorgehoben werden muss. Mit ikFast kann aus der vorhandenen Vorwärtskinematik eines beliebigen Roboters, die passende Inverskinematik analytisch bestimmt werden. Auch für Manipulatoren mit redundanten Freiheitsgraden kann eine Inverskinematik erstellt werden. Dazu werden die als redundant deklarierten Gelenke als

[3] Ein Agent ist ein Computerprogramm, welches zu einem gewissen selbstständigen Handeln fähig ist. Agenten treten oft in Gruppen auf.

freie Parameter bei der Berechnung außen vor gelassen und später beim Suchen einer Lösung iterativ ausprobiert.

Der Einsatz einer Planungsumgebung kann die Autonomie eines Roboters zwar erweitern, aber dadurch erhöht sich nur bedingt dessen selbstständiges Verhalten. Die einzelnen Schritte einer größeren Manipulationsaufgabe werden weiterhin in einem Skript definiert. Um einen Ablauf der Planungsschritte für ein Szenario festzulegen, wird wahlweise Python oder Matlab als Skriptsprache angeboten. Ein Skript bietet gegenüber einem Programm den Vorteil, dass es nicht kompiliert werden muss. So ist es auch denkbar, dass ein Skript von einer höheren Instanz, beispielsweise einem Aufgabenplaner, automatisch generiert wird.

Die Struktur der Roboter oder Objekte wird im XML Format modelliert. Dabei können bereits erstellte Objekte im Inventor oder Collada Format eingebunden werden. Die graphische Oberfläche von OpenRAVE besteht nur aus einem Coin3D Viewer. Die angezeigten Objekte und Roboter können dabei nicht nur per Skript gesteuert werden, sondern auch in der GUI verschoben oder im Falle eines Roboters in ihrer Konfiguration modifiziert werden (siehe Abbildung 2.10). Objekte besitzen zweierlei Modelle: Zum Einen das Renderingmodell, welches für die bloße Visualisierung

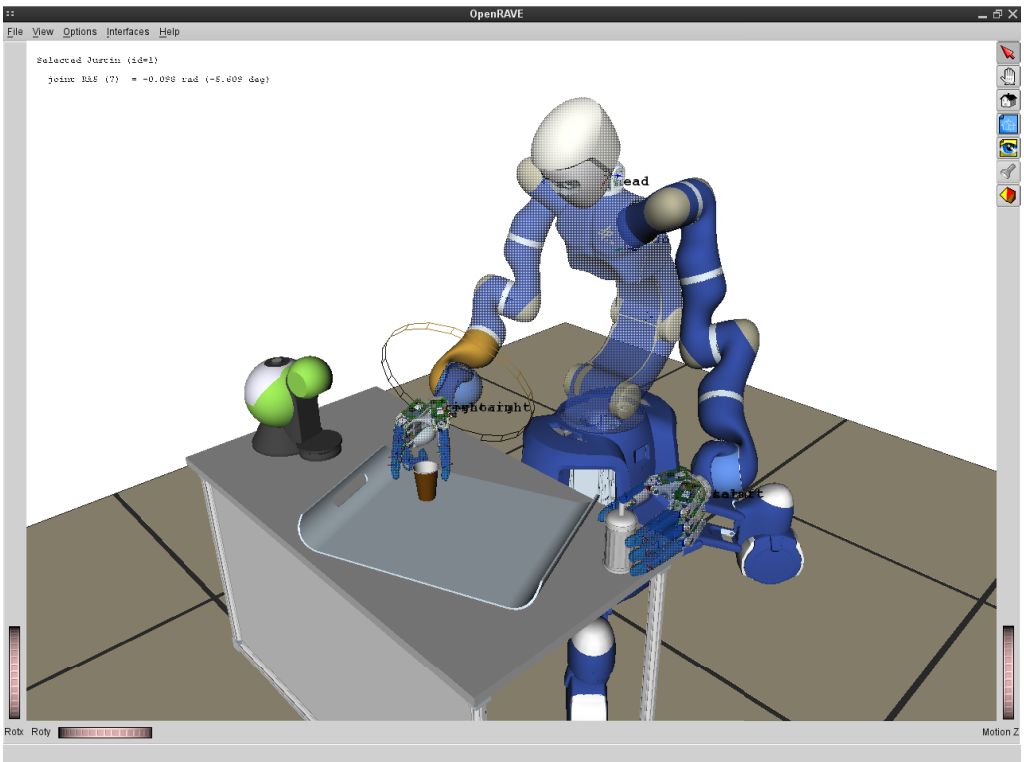


Abbildung 2.10: OpenRAVE Umgebung während der Modifikation eines Gelenkwinkels. Der orangefarbene Ring kann zur Veränderung der Gelenkwinkelstellung gedreht werden.

zuständig ist und zum Anderen das Datamodell, das zur Kollisionsabfrage verwendet wird. Dabei sollte das Kollisionsmodell mit einer möglichst geringen Anzahl von Polygonen erstellt werden. Je genauer das Kollisionsmodell, desto aufwändiger gestalten sich die Kollisionsüberprüfungen während der Laufzeit und damit auch die Pfadplanung selbst. Sind die Modelle hingegen zu ungenau, so wird die Umgebung nicht korrekt widergespiegelt und Aufgaben die hohe Präzision erfordern können nicht erfüllt werden. Einen guten Kompromiss erzielen *konvexe Hüllen*. Konvexe Hüllen umfassen ein Objekt vollständig, aber sparen detailreiche Strukturen aus. Objekte können so mit weniger Dreiecken modelliert werden.

2.2.5 Pfadplanung im Einsatz auf realen Robotersystemen

Die Integration von Algorithmen auf echten Systemen ist ein wichtiger Schritt. Speziell im Bereich der künstlichen Intelligenz wurden schon früh die theoretischen Grundprinzipien für diverse Autonomieprobleme entwickelt, aber nur wenige umgesetzt. Dies liegt vor allem daran, dass damals die nötige Hardware fehlte [31]. Mit dem Aufkommen komplexer Roboter können diese theoretischen Ansätze nun auf echten Systemen ausgeführt werden. Dabei treten häufig Probleme auf, die in einer simulierten Umgebung nicht berücksichtigt werden [40]. Insbesondere Leichtbauroboter sind konstruktionsbedingt stärker von Gravitationseinflüssen betroffen, wodurch ein in der Simulation erstellter Bewegungsablauf von der Realität abweichen kann. Zusätzlich ist die Planung mit unter sehr rechenaufwändig. Zwar hat sich die Rechenzeit im Vergleich zu den Anfängen der Pfadplanung auf wenige Sekunden reduziert, benötigt aber für komplexe Szenarien immer noch zu lange für die Generierung eines gültigen Pfades. Aus diesen Gründen ist die autonome Pfadplanung, insbesondere für humanoide Roboter, bis heute noch nicht weit verbreitet. Neben einigen bekannten Instituten im Forschungsbereich der Robotik verwenden nur vereinzelte Projekte Pfadplanung. In der Industrie kann Pfadplanung zur einmaligen Berechnung eines optimalen Pfades für Montageschritte verwendet werden [15]. In Summe tritt die autonome Pfadplanung jedoch verschwindend gering auf. In dieser Arbeit sollen daher die Planungszeiten sowie die Optimierungszeit deutlich verringert werden. Ebenfalls sollen die auf dem echten Roboter auftretenden Ungenauigkeiten kompensiert werden.

2.3 Justin - der mobile, humanoide Roboter des DLR

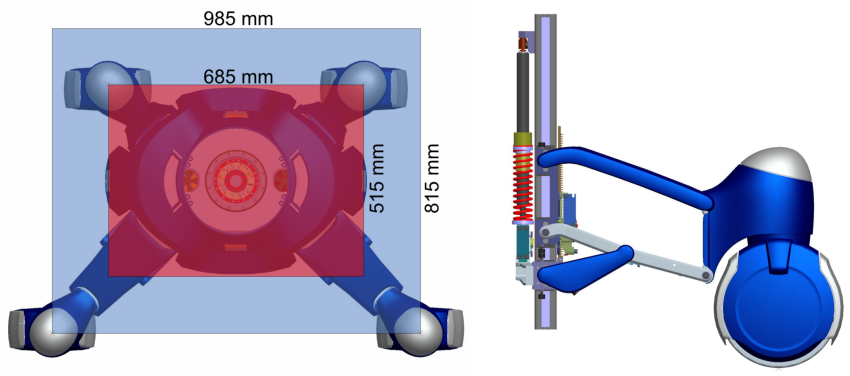
In diesem Abschnitt wird der mobile, humanoide Roboter *Rollin' Justin* vorgestellt [29]. Er wird seit einigen Jahren am Institut für Robotik und Mechatronik des Deutschen Zentrums für Luft- und Raumfahrt in Oberpfaffenhofen entwickelt. Die menschenähnliche Gestalt und Anordnung seiner Arme, sowie die Erweiterung des Systems um eine mobile Plattform, machen ihn zu einem optimalen Forschungswerkzeug für alltägliche Manipulationsaufgaben.

2.3.1 Aufbau von Justin

Basierend auf vorangegangen Forschungsergebnissen des DLR ist der Roboter völlig modular aufgebaut. So besteht Justin in erster Linie aus zwei DLR Leichtbaurobotern (*light weight robot, lwr*) [19], die als Arme dienen und jeweils einer DLR Hand als Endeffektor [7]. Verbunden sind die Arme über einen Torso, welcher ebenfalls in der Leichtbauweise entwickelt wurde. Als Kopf des Systems wird der DLR 3D-Modelierer verwendet, um die Umgebung wahrzunehmen. Die neueste Version von Justin, der so genannte *Rollin' Justin*, verfügt zudem über eine mobile Plattform und erhällt somit einen stark erweiterten Arbeitsraum. Insgesamt hat der Oberkörper 43 kontrollierbare Freiheitsgrade.

2.3.2 Mobile Plattform

Für komplexe Aufgaben ist die Fähigkeit, einen Manipulator beliebig zu platzieren, von großem Vorteil. Dies kann durch eine mobile Plattform geleistet werden. So können Suchalgorithmen für die Pfadplanung des Gesamtsystems weit über den eigentlichen Konfigurationsraum der Arme hinaus planen [13]. Außerdem kann der Roboter derart platziert werden, dass der Arbeitsraum der Arme optimal eingesetzt werden kann [39]. Desweiteren kann eine mobile Plattform weitere Vorteile beim Ausführen von Manipulationsaufgaben bringen. Beim Einsatz der Plattform während der Manipulation können zum Beispiel zusätzliche Kräfte aufgebracht werden. Ebenfalls möglich ist die aufgabenspezifische Positionierung der Manipulatoren mittels der Plattform [20]. Bei der Konstruktion der mobilen Plattform für Justin wurde auf Räder gesetzt. Im Gegensatz zu einem humanoiden Laufroboter mit zwei Beinen ist ein sicherer Stand garantiert. Um trotzdem mehr Flexibilität zu erlangen, sind die Räder an adaptiven Beinelementen angebracht. Wird ein sicherer Stand benötigt, können die Beine ausgefahren werden. Beim Durchqueren von Türen hingegen ist der Platz beschränkt, sodass die Beine eingefahren werden müssen (siehe Abbildung 2.11). Kleine Hindernisse, wie Teppiche und Schwellen, können durch zuschaltbare Dämpfer überwunden werden, ohne das die Stabilität des Roboters gefährdet wird.



(a) Variable Standfläche (b) Dämpfersystem

Abbildung 2.11: Variable Standfläche, und Dämpfersystem der mobilen Plattform [16].

2.3.3 Oberkörper und Kopf

Der Torso von Justin basiert auf der gleichen Leichtbautechnik wie die Arme [6]. Durch den beweglichen Oberkörper (siehe Abbildung 2.12) ist Justin in der Lage, Objekte vom Boden gleichermaßen zu erreichen, wie Objekte in einem Regal. Die einzelnen Gelenke sind derart konstruiert, dass das Brustsegment keinen hohen Lasten ausgesetzt ist. Eine Seilkonstruktion leitet die hohen Drehmomente, die auftauchen wenn Justin seine Arme ausstreckt in die mobile Plattform. Auf dem Torso sitzt ein 3D-Modellierer, dessen Kameras als Justins Augen dienen.

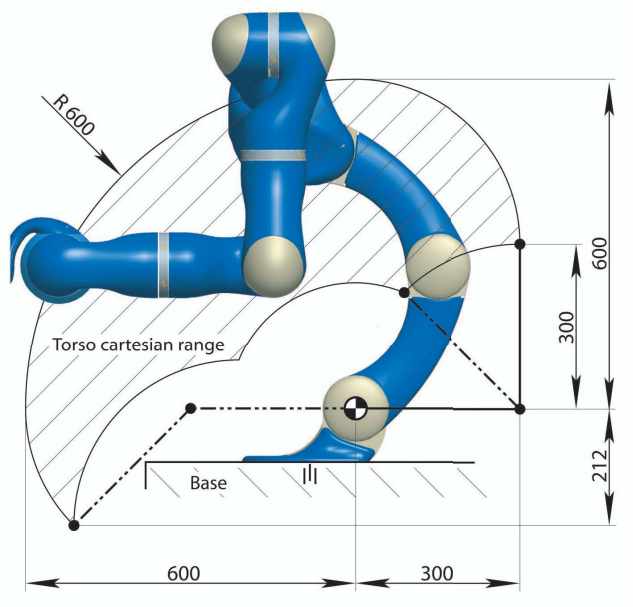


Abbildung 2.12: Aktionsbereich des Torsos [29].

2.3.4 DLR Leichtbauarme

Der Leichtbauroboter des DLR befindet sich mittlerweile in der dritten Generation [19]. Mit einem Gewicht von 13,5 kg ist er in der Lage eine Last entsprechend seinem Eigengewicht zu heben. Anders als bei herkömmlichen Industrierobotern befindet sich beim LWR III die gesamte Steuerungslogik im Gehäuse des Armes. Ebenso wie der menschliche Arm hat der DLR-Arm sieben Freiheitsgrade, wodurch eine dem Menschen ähnliche Flexibilität erreicht wird. Durch die integrierten Regelungen erhält der Arm zudem eine gewisse Nachgiebigkeit, was den Umgang mit dem Roboter im Ganzen sicherer macht.

2.3.5 DLR Hand

Die Hand ist das wichtigste Werkzeug des Menschen und das gilt auch für die Serviceroboter der Zukunft. Mit der Hand ist es dem Menschen möglich Objekte zu greifen und Werkzeuge zu benutzen. Die DLR-Hand versucht diese Vielseitigkeit zu reproduzieren [7]. Die aktuelle Generation der Hand, die an Justin montiert ist, besteht aus vier gleichwertigen Fingern mit je drei Freiheitsgraden. Sie ist weitgehend der menschlichen Hand nachempfunden, muss allerdings mangels Miniaturisierung einen Finger einbüßen. Die DLR Hand ist sehr gut geeignet für eine Vielzahl von Manipulationsaufgaben. Die Manipulationsfertigkeiten werden umso größer, wenn beide Hände parallel eingesetzt werden.

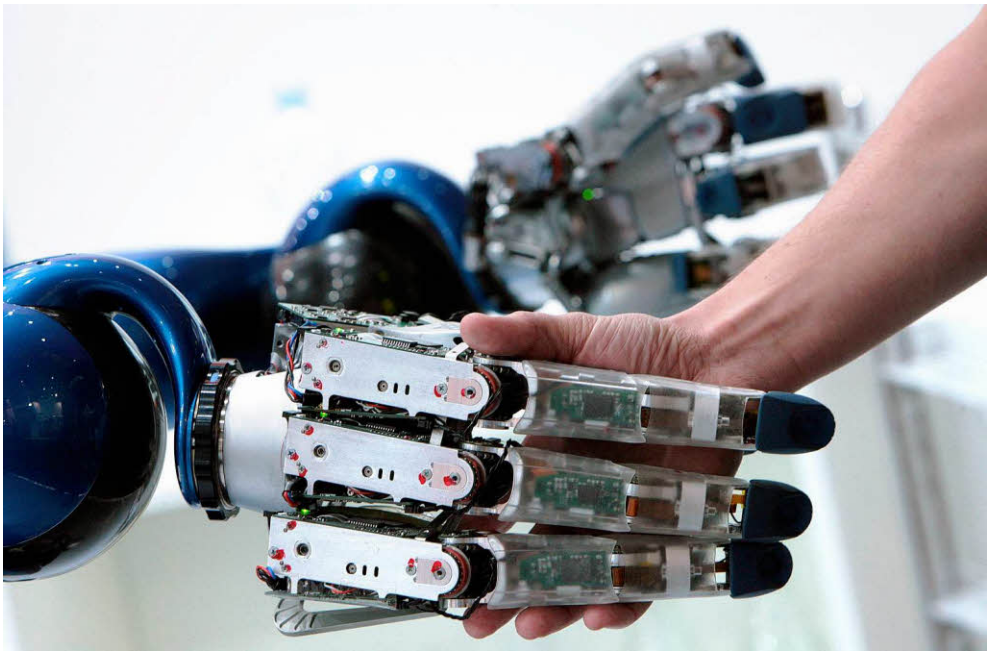


Abbildung 2.13: Justins antropomorphe Hand in Aktion.

3 Problemanalyse

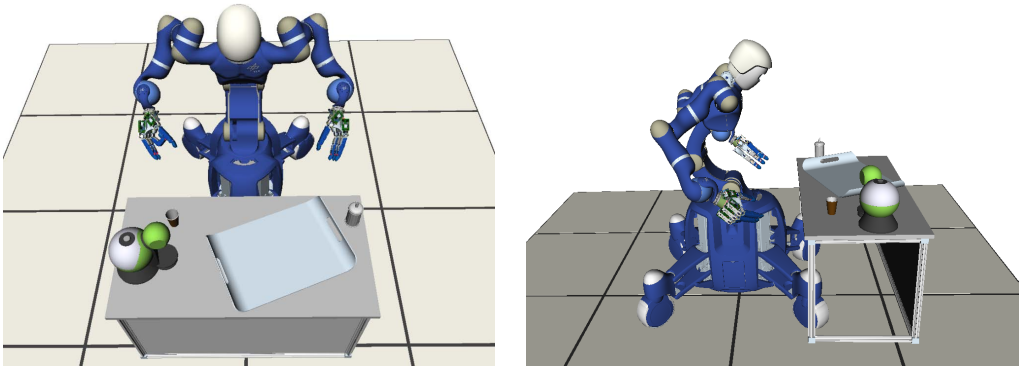
Dieses Kapitel beschäftigt sich mit den Problemen des Pfadplanungsvorganges. Dabei werden zweierlei Fragen verfolgt: Zuerst wird der Frage nachgegangen, welche Elemente des Planungs- und Optimierungsalgorithmus am zeitaufwändigsten sind. Dazu wird eine Versuchsreihe durchgeführt, bei denen der Roboter verschiedene Trajektorien planen muss. Die Zeiterfassung erfolgt auf der untersten Ebene des Algorithmus. Die Planung und die Optimierung werden dabei gesondert betrachtet. Die zweite Fragestellung beschäftigt sich mit der Ausführung der geplanten Trajektorie auf dem realen Roboter. Frühere Arbeiten haben gezeigt, dass Justin als Leichtbauroboter konstruktionsbedingt stärker durch externe und interne Beschleunigungen beeinflusst wird als herkömmliche Industrieroboter [40]. Dabei spielen sowohl Gravitation, als auch Beschleunigungen entlang der Trajektorie eine Rolle.

3.1 Zeitanalyse für einen RRT Planer

Zur Zeitanalyse der Pfadplanung und der dazugehörigen Optimierungsstrategie wird zunächst ein Versuch in der Simulation durchgeführt. Die verwendeten Algorithmen sind dabei der BiRRT Algorithmus zur Pfadplanung und das Standardverfahren zur Optimierung in OpenRAVE, das Linear Shortcut Verfahren. Um repräsentative Werte zu erhalten, wird hierbei bereits auf ein alltägliches Szenario gesetzt. Dieser erste Versuch beschreibt ein Aufräumszenario in einer Küchenumgebung. Dieses Szenario wird im Verlauf dieser Arbeit öfter als Referenzexperiment verwendet um die Ergebnisse der entwickelten Algorithmen besser miteinander vergleichen zu können. Auch die Versuche auf dem echten Roboter werden später diesen Aufbau verwenden.

Bei dem verwendeten Computer handelt es sich um einen Standardarbeitsplatzrechner der Dell Precision T3400 Serie. Diese bestehen aus einem Intel®Core™2 Duo CPU E8400 3.00GHz Zweikernprozessor mit 6144 KB Cache und einem Hauptspeicher von 2GB. Als Betriebssystem wird OpenSUSE 11.x eingesetzt. Die Versuche werden mit der SVN Revision 1984 von OpenRAVE durchgeführt. Auch alle späteren Versuche dieser Arbeit werden mit dieser Rechnerklasse ausgeführt. Bei den Versuchen zur Parallelisierung kommen dabei bis zu vier identische Rechner zum Einsatz.

3.1.1 Versuchsaufbau



(a) Draufsicht (b) Seitenansicht

Abbildung 3.1: Draufsicht und Seitenansicht des Aufräumszenarios.

Der Aufbau dieses Szenarios lehnt sich an die Laborumgebung von Justin an. Die verwendeten Modelle basieren auf den realen Objekten in der Laborumgebung. Die Umgebung besteht aus einem Tisch mit einigen darauf befindlichen Gegenständen. Der Rest des Labors befindet sich nicht in Reichweite des Roboters und ist deswegen nicht modelliert. Da sich Justin nicht von seiner Ausgangsposition weg bewegt und lediglich mit den Armen hantiert, reicht dies völlig aus. Auf dem Tisch befinden sich zwei Hindernisse und zwei Zielobjekte. Ein Kaffeebecher und eine Zuckerdose sind dabei die zu greifenden Objekte, wohingegen der Kaffeemaschine und dem Tablett ausgewichen werden muss. Auch der Tisch und der Roboter selbst sind als Hindernisse zu betrachten.

Körperteil	Gelenkwinkelparameter in Grad
Torso	0,00 -48,85 77,17
Rechter Arm	-24,37 -89,88 5,01 90,00 35,00 -9,96 39,98
Rechte Hand	-12,20 0,00 0,00 0,00 34,37 0,00 0,00 34,37 0,00 0,00 55,11 81,82
Linker Arm	-24,37 -89,88 5,01 90,00 35,00 -9,96 39,98
Linke Hand	7,79 15,35 0,00 0,00 21,20 0,00 0,00 21,20 0,00 0,00 21,20 0,00
Kopf	0,00 13,00

Tabelle 3.1: Startkonfiguration für das Optimierungsszenario.

Konkret steht Justin im Ursprung des Weltkoordinatensystems und damit auch zentriert vor dem Tisch, der nur in der X-Achse verschoben ist. Justin steht also in einem relativen Abstand von 0,34 m von der vorderen Tischkante entfernt. Die Objekte auf dem Tisch wurden mit Bezug auf die Reichweite der Arme zufällig verteilt. Die Koordinaten der Objekte sind im Weltkoordinatensystem angegeben. Die Höhe beträgt dabei immer 0,766 m. Die Kaffeemaschine steht am rechten hinteren Rand des Arbeitsplatzes auf der kartesischen Position X = 0,850 m und Y = -0,520 m und ist um 13° auf dem Tisch rotiert. Das Tablett ist mit einer Position von X = 0,788 und Y = 0,184 eher auf der linken Seite des Tisches orientiert. Es ist um 22° gedreht. Die zu greifenden Objekte befinden sich am vorderen linken und rechten Rand des Tisches. Der Kaffeebecher steht auf der Position X = 0,611 und Y = -0,364. Die Zuckerdose befindet sich auf der gegenüber liegenden Seite auf dem Punkt X = 0,607 und Y = 0,573. Justin steht zunächst in der Ausgangsstellung für das Greifen des Bechers mit drei Fingern. Die gesamte Stellung ist in Tabelle 3.1 aufgelistet. Der detaillierte Aufbau ist im Anhang A.1 angefügt.

3.1.2 Versuchsablauf

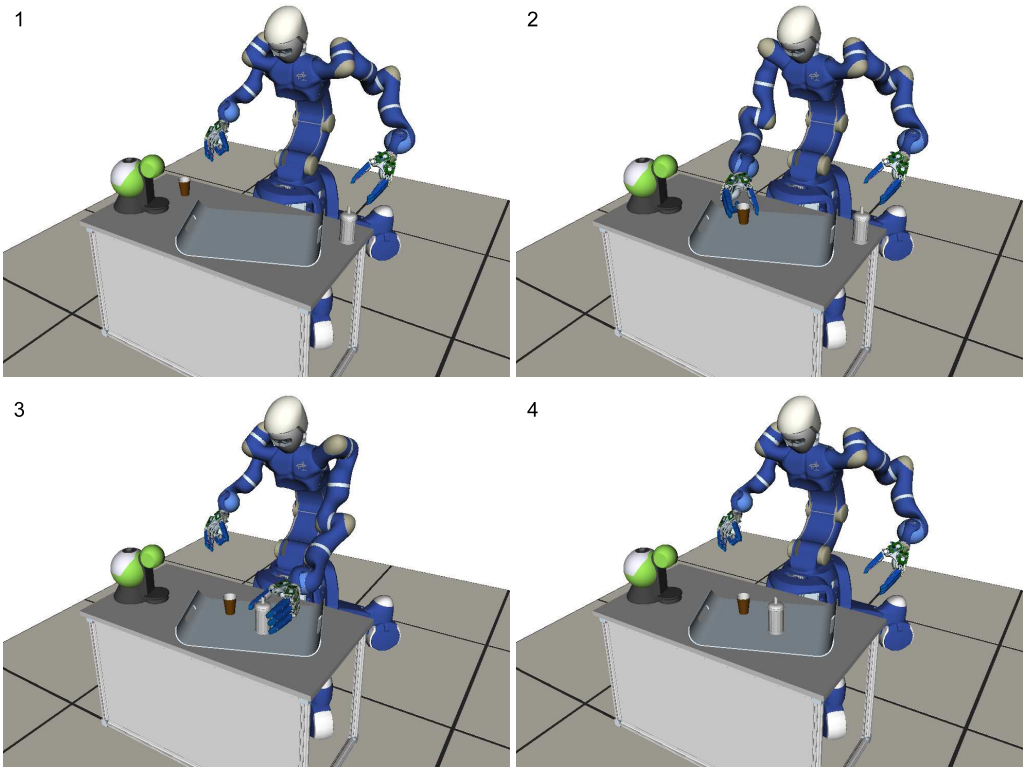


Abbildung 3.2: Ablauf des Aufräumszenarios.

Für Justin stellt sich das Szenario als Aufräumszenario dar. Der Becher und die Zuckerdose müssen dabei für einen möglichen Abtransport auf dem Tablett positioniert werden. Der Roboter verfährt dabei sequentiell. Eine Parallelisierung in der Abarbeitung der Handlungsabschnitte, wie in [40] gezeigt, steht nicht im Fokus dieser Arbeit. Die Trajektorien werden zunächst geplant und anschließend optimiert. Beginnend mit dem Becher wird dieser zunächst mit dem rechten Manipulator gepackt und auf dem Tablett abgelegt. Anschließend wird der Arm wieder in seine Ausgangsposition gebracht. Das Gleiche wird danach mit dem Zucker und dem linken Manipulator wiederholt. Die Positionen zum Absetzen sind für den Becher der Punkt p1 an der Stelle X = 0,759 m, Y = 0,042 m und für die Zuckerdose der Punkt p2 mit den Koordinaten X = 0,832 m, Y = 0,252 m. Der komplette Ablauf wird 20 Mal wiederholt. Die gemessenen Zeiten werden anschließend gemittelt. Der Ablauf ist in Abbildung 3.2 skizziert.

3.1.3 Ergebnisse

Die gemessenen und gemittelten Zeiten in Abbildung 3.3 decken sich mit früheren Arbeiten [41]. Die blauen Balken stehen für die Pfadplanung und die grünen Balken für die Optimierung. Diese sind wiederum geteilt in algorithmenspezifische Abläufe und Kollisionsüberprüfungen. Der Algorithmen-Abschnitt beinhaltet die Initialisierung, den Aufbau des Baums und die Speicherverwaltung. Die Kollisionsüberprüfung beinhaltet eine Überprüfung auf Selbstkollisionen und eine Überprüfung auf Kollisionen mit Objekten in der Umgebung. Die gemessenen Zeiten zeigen, dass die Kollisionsüberprüfungen den zeitaufwändigsten Teil der Verfahren ausmachen. Dies gilt sowohl für die Planung als auch für die Optimierung. Der Glättungsschritt beansprucht selbst bei diesen einfachen Trajektorien mehr als die Hälfte der Gesamtrechendauer.

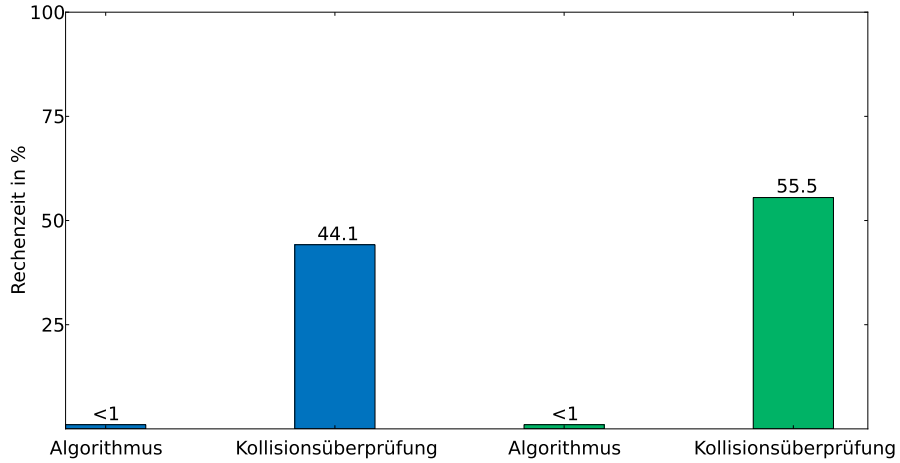


Abbildung 3.3: Zeitliche Aufteilung der Algorithmen. Die blauen Balken stehen für die Pfadplanung und die grünen Balken für die Optimierung.

Das Histogramm aus Abbildung 3.4 zeigt das 75% aller Kollisionsüberprüfungen etwa 3,7 Millisekunden Berechnungszeit benötigen. Selbst bei einer relative kurzen Trajektorie, zum Beispiel dem Greifen eines frei stehenden Glases, werden schon über 4000 Kollisionsüberprüfungen für Planung und Optimierung benötigt. Damit liegt die Gesamtdauer der Berechnung bereits bei fast 15 Sekunden. Oft sind die Trajektorien aber um einiges länger als in dem hier verwendeten Szenario und die Kollisionsüberprüfungen in komplexeren Umgebungen weit teurer.

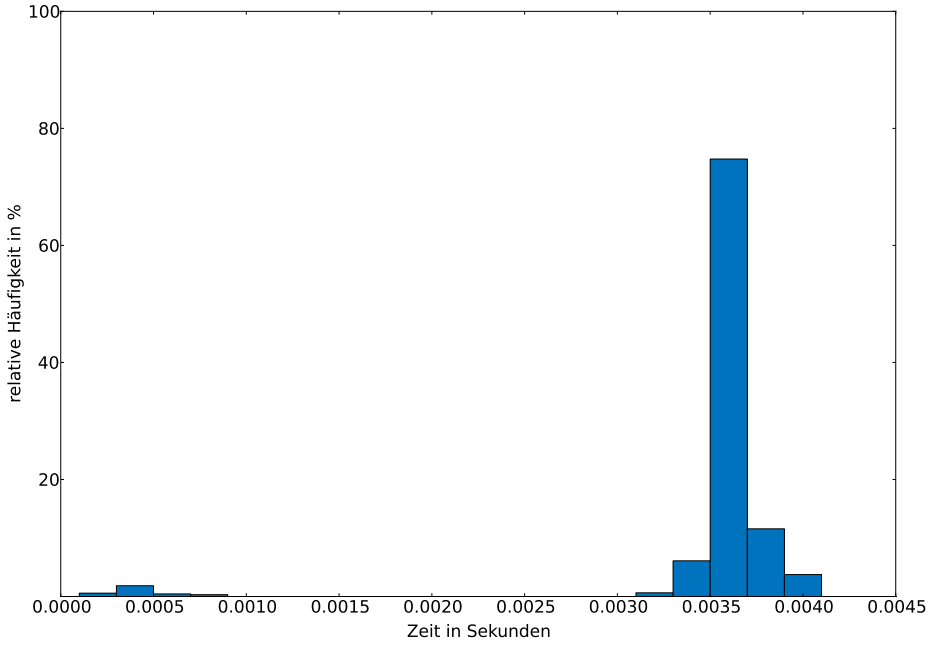


Abbildung 3.4: Histogramm über den Zeitverbrauch der Kollisionsüberprüfungen.

Zur beschleuningung der Pfadplanung, muss also letztendlich die Kollisionsüberprüfung beschleunigt werden. Um die Kollisionsüberprüfung zu beschleunigen, kann diese mehrfach parallel ausgeführt werden. Die bereits im Stand der Forschung erwähnten parallelen Planungsmethoden verfolgen dieses Prinzip. In dieser Arbeit wird zusätzlich das Optimierungsverfahren mit diesem Hintergedanken parallelisiert. Die entsprechende Softwarearchitektur dazu wird in Kapitel 4 beschrieben. Die Integration der einzelnen Planungselemente wird in Kapitel 5 vorgestellt.

3.2 Ausführung auf dem realen Roboter

Die meisten Pfadplanungsalgorithmen gehen von starren Körpern für die Kinematik eines Roboters aus. Justin besitzt jedoch eine Leichtbauroboterstruktur und ist damit nachgiebig. Die Annahme eines starren Körpers ist also nicht uneingeschränkt gültig [40]. Die Struktur des Roboters verformt sich leicht unter den herrschenden

Beschleunigungen entlang einer Trajektorie oder der Gravitation. Die Pose des Manipulators und damit die Pose des Endeffektors in der Simulation können dabei stark von der Realität abweichen. Eine denkbare Lösung, die sich mit OpenRAVE realisieren ließe, ist ein Dynamikmodel mit in die Planung aufzunehmen. Jedoch benötigt die Berechnung der Dynamik des Leichtbauroboters viele Ressourcen und ist damit rechnerisch aufwändig. Zudem sind solche Modelle oft zu ungenau um verwertbare Ergebnisse zu erzielen. Der minimale Abstand c zwischen dem Roboter und einem Hindernis in der Umgebung weicht damit ebenfalls von der Realität ab.

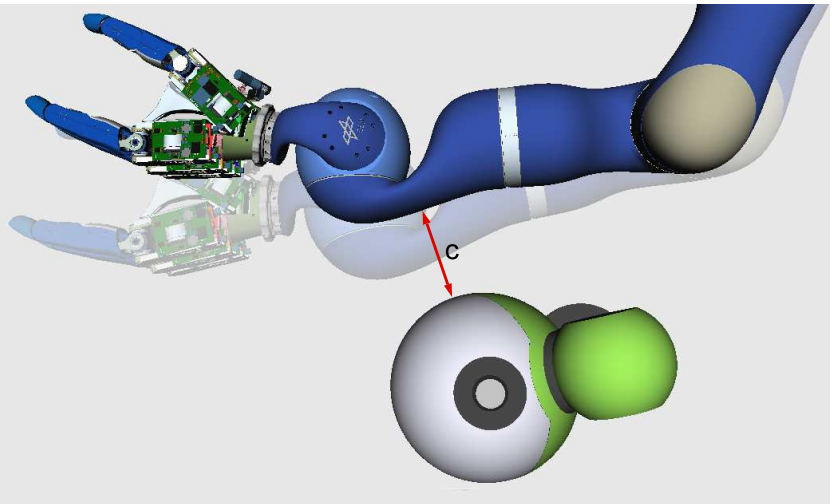


Abbildung 3.5: Schematische Darstellung der Bewegung eines Manipulators mit Überschwingen. Der transparente Arm ist hier stellvertretend für die reale Position des Roboters. Der Abstand zu möglichen Hindernissen in der Umgebung wird in der simulierten Welt berechnet und stimmt damit ebenfalls nicht mit der Realität überein.

Zur kompensierung der gravitationsbedingten Abweichungen, sind bereits zwei Möglichkeiten in einer früheren Arbeit angedacht worden [40]. Beide Methoden setzten auf ein Gravitationsmodell des Roboters und arbeiten vom Prinzip her identisch. Die erste Methode verwendet das Modell zur Korrektur der Lage des Oberkörpers während der Planung, wohingegen die zweite Variante das Gravitationsmodell zur Nachbearbeitung der Trajektorie der Arme verwendet. Das Gravitationsmodell stellt Informationen über die Abweichung der Position der Robotergelenke, von der durch die direkte Kinematik vorhergesagten Pose, als Offset bereit. Unter der Berücksichtigung dieses Offsets, kann eine geplante Trajektorie in die entsprechenden Motorparameter zur Ansteuerung des realen Roboters umgerechnet werden. Jedoch ist dieses Modell nicht exakt und verlangsamt den Planungsprozess. Im Planungsprozess werden viele Konfigurationen ausprobiert, die für das Erreichen des Zieles nicht von Bedeutung sind. Daher ist es ratsam das Gravitationsmodell möglichst nur zur Nachbearbeitung zu verwenden.

Die Beschleunigungen, die während einer Trajektorie auftreten, sind indessen weit schwieriger zu kompensieren. Dazu wäre es nötig, ein vollständiges Dynamikmodell des Roboters bei jedem Schritt der Pfadplanung zu berücksichtigen. Der dadurch entstehende Rechenaufwand verhindert den Einsatz der Planung im täglichen Betrieb. Daher wird in dieser Arbeit ein alternativer Weg gegangen, um dieses Problem zu beseitigen. Die Beschleunigungen entlang einer Trajektorie führen zu Überschwingern, die wiederum ungewollte Kollisionen auslösen (siehe Abbildung 3.5). Eine aktive Handhabung der Beschleunigungen und Geschwindigkeiten bei der Ausführung einer Trajektorie kann dies vermeiden. Informationen über den minimalen Abstand zwischen dem Roboter und den umstehenden Hindernissen können dazu verwendet werden den Interpolator des Roboters zu parametrisieren. Der resultierende Pfad besteht dann aus Konfigurationen mit variablen Geschwindigkeits- und Beschleunigungsprofilen. Diese zusätzliche Information kann Überschwinger vermeiden und damit Kollisionen verhindern. Die Berechnung der Abstandsinformation wird innerhalb der verteilten Softwarearchitektur bereitgestellt und in Kapitel 5 untersucht.

4 Eine verteilte Softwarearchitektur zur effizienten Pfadplanung für Rollin' Justin

Ein mobiler, humanoider Serviceroboter wie Rollin' Justin, ist heutzutage in der Lage komplexe Aufgaben auszuführen. Dazu gehört zum Beispiel das Servieren einer Eistee-Mischung (siehe Abbildung 4.1). Meistens sind die Bewegungsabläufe zum Bewältigen solcher Aufgaben von Menschenhand vorgegeben. Um in einer komplexen Umgebung einen solchen Bewegungsablauf autonom zu planen, ohne dabei mit der Umwelt zu kollidieren, benötigt der Roboter einen effizienten Pfadplanungsmechanismus. Dabei darf die Zeit zwischen Aufkommen der Aufgabe und deren Ausführung nicht zu lange dauern, damit der Beobachter nicht ungeduldig wird.

Abbildung 4.1: Justin beim Öffnen einer Teedose [33].

Die in dieser Arbeit vorgestellte Pfadplanungsumgebung versucht diese Zeit zu minimieren, indem verteilte und parallele Strukturen verwendet werden. Um alle Berechnungszeiten des Planungszykluses möglichst effizient zu verringern, bietet sich eine allgemeine Softwarearchitektur zur Parallelisierung der einzelnen Abläufe an. Nach Balzert ist eine Softwarearchitektur *„eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen“* [3]. Das folgende Kapitel behandelt daher nicht nur den strukturellen Aufbau des Gesamtsystems sowie dessen einzelne Komponenten, sondern auch die Kommunikation zwischen den Modulen und die Aufteilung der Hardwareressourcen.

4.1 Anforderungen an die Softwarearchitektur

Die in diesem Kapitel vorgestellte Softwarearchitektur ist dazu ausgelegt die folgenden beiden Ziele zu erfüllen:

- Die Performanz des gesamten Planungszykluses soll hinsichtlich der Planungszeit signifikant verbessert werden, damit die Zeit zwischen der Anforderung einer Aufgabe und deren Ausführung auf ein Minimum zu reduziert werden kann.

- Die geplanten Pfade sollen mit zusätzlichen Informationen über den minimalen Abstand zwischen dem Roboter und dessen Umgebung erweitert werden. Dadurch soll es möglich sein, die Modelierungsdiskrepanzen und Fehler eines Dynamikmodells, die zu Ungenauigkeiten in der Ausführung einer Trajektorie des Leichtbauroboters führen, zu relativieren.

Beide Ziele sind wichtig und notwendig für die Integration eines Pfadplaners in ein System zur autonomen Aufgabenplanung. Letztendlich soll damit ein komplexer Roboter wie Rollin' Justin in der Lage sein, effizient Aufgaben der realen Welt zu bewältigen. Auf der einen Seite soll also die Rechenzeit minimiert werden und auf der anderen Seite soll der Pfad mit zusätzlichen Informationen angereichert werden, damit eine gewisse Sicherheit bei der Ausführung garantiert werden kann. Da die Erweiterung des Pfades mit den Abstandsinformationen allerdings zusätzliche Rechenzeit beansprucht, sind diese Ziele gegenläufig. Nichtsdestotrotz können beide Anforderungen in einem Framework vereint werden. Um die Berechnungszeit dabei so gering wie möglich zu halten, müssen neben der Parallelisierung des Planungsschrittes und des Optimierungsschrittes zusätzlich noch die Abstandsabfragen parallelisiert werden. Aus der vorangegangenen Problemanalyse und den Anforderungen geht also hervor, dass die Softwarearchitektur diese drei Elemente der Pfadplanung in sich vereinen muss. Die Integration der einzelnen Algorithmen in die Softwarearchitektur wird in Kapitel 5 untersucht. Als Grundlage dafür wird in diesem Kapitel die dafür notwendige Softwarearchitektur vorgestellt.

4.2 Hierarchische Architektur der Softwareelemente

Pfadplanung stellt einen wesentlichen Teil beim Lösen von Aufgaben dar. Zur effektiven Nutzung der Pfadplanung, in der Aufgabenplanung, wird daher ein einfaches und dennoch wirkungsvolles Framework benötigt.

Die Softwarearchitektur zur Parallelisierung der Planungselemente besteht sowohl aus mehreren Prozessen auf einem einzelnen Rechner, als auch aus mehreren physikalisch getrennten Computern, um eine weitere Verteilung zu ermöglichen. Mit einem Server und mehreren Clients arbeitet das System nach dem *Remote Procedure Call* Verfahren in umgekehrter Hierarchie. Nicht der Server verrichtet die Arbeit, sondern die vielen Clients lösen ein Problem gemeinsam. Die Aufgabenverteilung erfolgt also hierarchisch. Es gibt drei hierarchische Stufen. An oberster Stelle steht eine Kontrollinstanz zur Verteilung der einzelnen Teilaufgaben. Diese Kontrollinstanz wird hierbei als *Commander* bezeichnet (siehe Abschnitt 4.2.1). Der Commander wird als einziges Unterprogramm auf dem Server gestartet und ist als Plugin für OpenRAVE implementiert. Dies ist besonders vorteilhaft, da somit kein zusätzlicher Aufwand für den Benutzer beim Erstellen von Planungsskripten entsteht. Der Benutzer kann wie gewohnt OpenRAVE zur Pfadplanung einsetzen. Im Hintergrund wird die aufwändige Berechnung jedoch ausgelagert.

Zur eigentlichen Berechnung der Planungsaufgaben sind Serviceprogramme erforderlich, welche einmalig beim Plugin-Aufruf auf den Clients gestartet werden. Per SSH verbindet sich der Server mit den gewünschten Hostrechnern. Dazu werden die Rechnernamen und deren maximale Prozesszahl aus einer Umgebungsvariable gelesen. Der SSH-Aufruf führt das Vermittlungsprogramm, hier *Broker* genannt (siehe Abschnitt 4.2.2), auf dem Client aus und überliefert zusätzlich Informationen für die spätere Kommunikation über das TCP Protokoll. Der Broker ist zuständig für die Verteilung der einzelnen Aufgaben an mehrere Arbeitsprozesse, die sogenannten *Agenten* (ebenfalls Abschnitt 4.2.2). Diese werden nicht per SSH gestartet, sondern aus dem Broker mit dem Fork-Befehl abgespalten. Sobald alle Unterprogramme gestartet sind, meldet sich der Clientrechner am zuständigen TCP-Socket des Servers an. Die weitere Kommunikation erfolgt ab sofort über diese Verbindung. Auf der Clientseite kommunizieren die Serviceprogramme untereinander über *Shared Memory*. So können die Unteraufgaben nahezu ohne Zeitverlust durch Kommunikation auf mehreren Prozessoren abgearbeitet werden. Die gesamte Struktur wird als *Distributed Planning Cycle Environment (kurz DISPLACE[4])* bezeichnet, da sie die Verteilung des kompletten Planungszykluses in sich vereint. Der Planungszyklus besteht dabei aus der Planung und der Optimierung. Zusätzlich sind noch die Abstandsabfragen integriert. Die Integration der Planungselemente in die Softwarearchitektur wird gesondert betrachtet. Zunächst wird die Architektur im Detail vorgestellt. Abbildung 4.2 visualisiert den Aufbau als Blockschaltbild.

[4] Displace steht im Englischen für verdrängen oder verlagern und spielt damit auf die Verlagerung der rechenaufwändigen Kollisionsabfragen auf mehrere Computer und Prozessoren an.

4.2.1 Aufbau des Servers

An oberster Stelle der Hierarchie steht der Server, mit dem darin beinhalteten *Commander*. Der Commander ist verantwortlich für den Aufbau der Kommunikationswege und für die Verteilung der Unteraufgaben auf die zahlreichen Clients. Der Server besitzt ebenso wie die auf den Client befindlichen Ebenen eine OpenRAVE Instanz. Die OpenRAVE Instanz des Servers dient im Wesentlichen zur Verwaltung der in OpenRAVE üblichen Datenstrukturen. Es werden keine Planungen oder ähnliche Aktionen ausgeführt. Ferner besitzt diese Instanz die einzige Visualisierung für die geplanten Bewegungen.

Der Commander ist als Plugin für OpenRAVE konstruiert und besteht daher wie alle Planungsplugins im Wesentlichen aus einer Klasse mit den beiden Methoden *InitPlan()* und *PlanPath()*: Üblicherweise werden Plugins in OpenRAVE bei jedem Aufruf neu initialisiert. Da dies aber bei der verteilten Softwarearchitektur den wiederholten Aufbau der kompletten Infrastruktur bedeuten würde, ist der Commanders als *static* deklariert. Der Commander, und damit der eröffnete Socket File-Descriptor, bleiben so über den Gültigkeitsbereich des Plugins bestehen. Außerdem wird so sichergestellt, dass nur ein Commander in der gesamten Planungsumgebung existiert. Bei der Initialisierung wird zunächst ein TCP Socket auf einem freien Port geöffnet. Dieser dient späterhin als Kommunikationsweg mit den Clients. Gestartet werden diese jedoch, wie bereits beschrieben, über eine SSH Verbindung. Sobald die Verbindung steht und sich alle Clients zurückgemeldet haben, ist die Initialisierung abgeschlossen. Die TCP Verbindung kann nun bei jeder Planungsphase wiederverwendet werden. Beim Aufruf der InitPlan() Funktion wird das Verhalten der Planung mit einem Keyword spezifiziert. Die Keywords „*plan*“, „*optimize*“ oder „*distance*“ stehen zur Verfügung. Mit erneutem Aufruf der Initialisierung mit Angabe eines Keywords wird der Planungsmodus umgeschaltet. Wird kein Keyword bei der Initialisierung angegeben, erfolgt beim Aufruf der PlanPath() Funktion eine Planung mit anschließender Optimierung.

Die PlanPath() Funktion verteilt die entsprechenden Aufgaben an die Clients. Im eigentlichen Sinne plant diese Funktion keinen Pfad. Stattdessen wird über die TCP Verbindungen die Aufgabe kommandiert. Die drei zuvor erwähnten Methoden stehen zur Verfügung: Pfadplanung, Pfadoptimierung und Abstandsabfragen. Ein Netzwerkpaket für eine Aufgabe besteht dabei immer aus einem *Header* und einem variablen *Datenfeld*. Der Header beinhaltet den Nachrichtentyp, eine fortlaufende Identifikationsnummer und ein Servicefeld, indem je nach Bedarf eine Integerzahl hinterlegt werden kann. Abhängig der Aufgabe beinhaltet das Servicefeld etwa die Anzahl der aktiven Freiheitsgrade oder die Pfadlänge einer berechneten Trajektorie. Alle anfallenden Aufgaben werden asynchron ausgeführt. Das heißt, sobald eine Aufgabe versendet wurde, wird nicht auf die passende Antwort gewartet, sondern es werden gleich weitere Aufgaben verteilt. Der Server blockiert nicht und kann somit auch gleich wieder Antworten anderer Clients empfangen. Erst wenn alle Clients mit Aufgaben versorgt sind, wartet der Server auf eine Antwort.

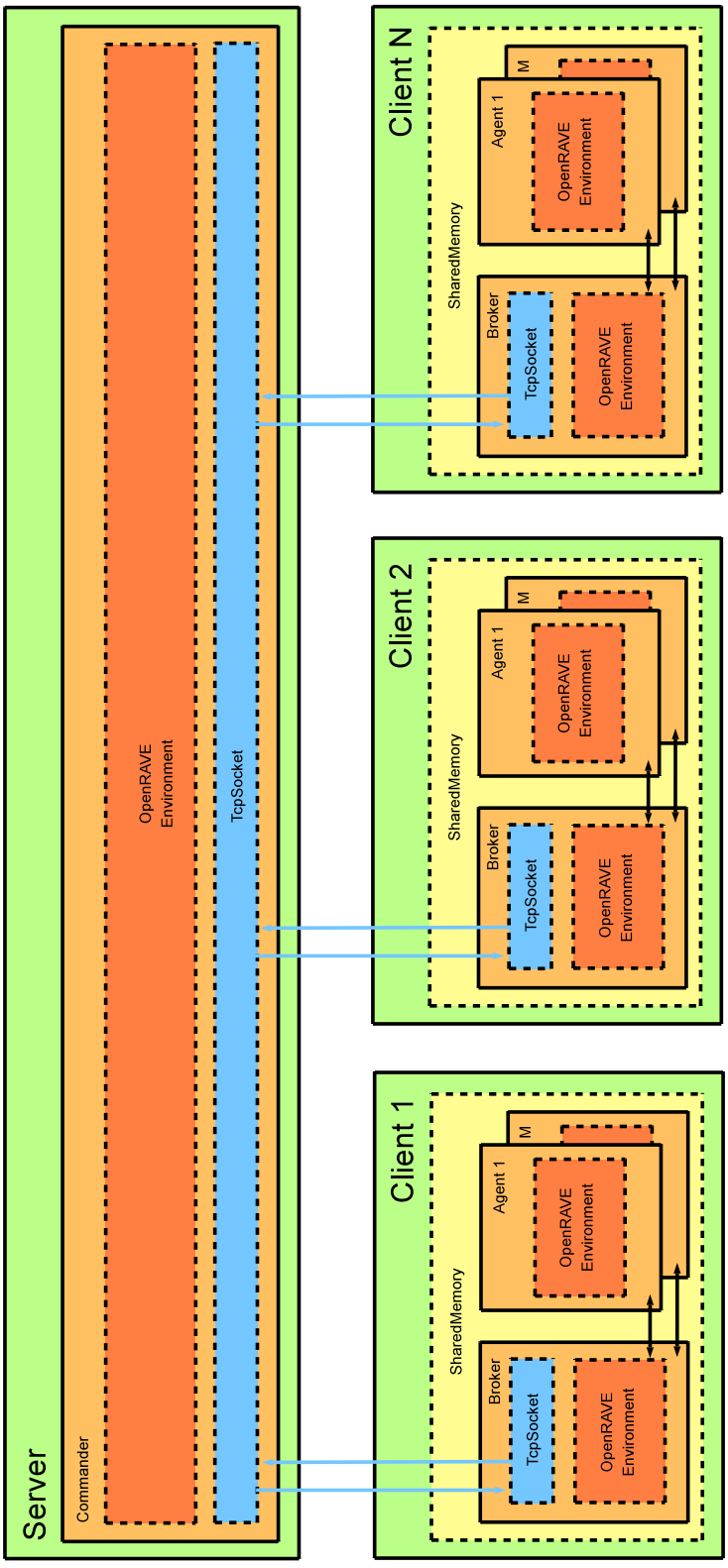


Abbildung 4.2: Hierarchische Softwarearchitektur zur Verteilung der Bewegungsplanung. Grüne Blöcke stellen physikalische Rechner dar. Orangefarbene Blöcke sind eigenständige Prozesse. Jeder Prozess hat seine eigene Repräsentation der OpenRAVE Umgebung (hier als roter Block dargestellt). Die Kommunikation erfolgt über die im Bild blau gezeichneten TCP Verbindungen. Untereinander kommunizieren Prozesse auf dem gleichen physikalischen Computer über Shared Memory, welches hier gelb angedeutet ist.

4.2.2 Aufbau des Clients

Die Clients beinhalten gleich zwei Hierarchiestufen der Softwarearchitektur. Jeder Client besteht aus einem Broker und mehreren Agenten. Pro physikalischem Kern ist ein Agent vorgesehen. Abbildung 4.3 zeigt den Client im Detail.

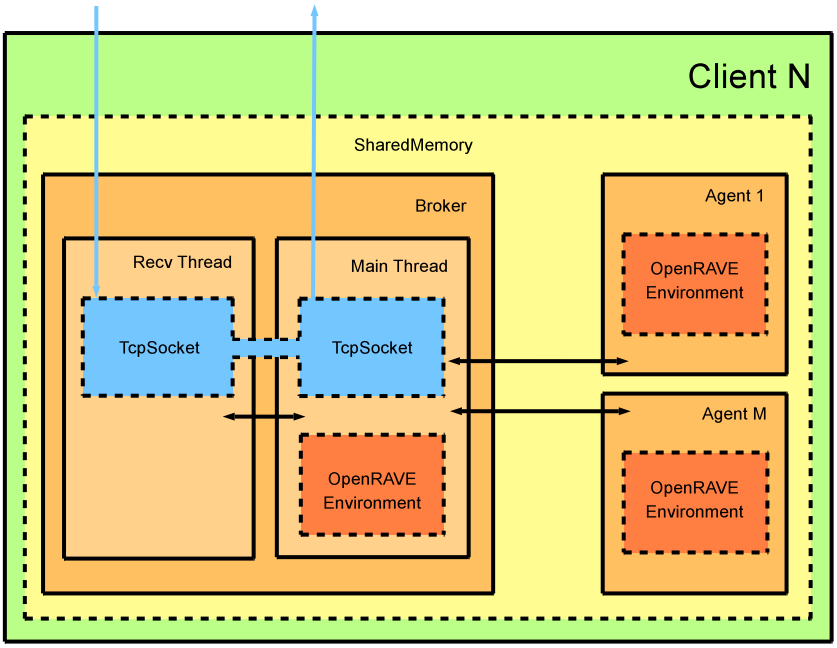


Abbildung 4.3: Die interne Architektur des Clients im Detail. Die Farbkodierung entspricht der Übersichtsdarstellung. Zusätzlich sind die Threads des Brokers noch in einem hellen Orange dargestellt. Die Verbindung des TCP Sockets symbolisiert die gemeinsame Nutzung des Sockets durch die beiden Threads.

Der Broker ist das Gegenstück zum serverseitigen Commander. Dieser bildet die Vermittlungsschicht, welche eigentlich aus zwei Threads besteht. Ein Thread zum Empfangen der vom Server gesendeten Aufgaben und ein Hauptthread, der die empfangenen Nachrichten an die Agenten weiterleitet. Der Empfängerthread ist abgekapselt, um blockierend auf externe Nachrichten warten zu können. Der Hauptthread verteilt die Unteraufgaben nicht-blockierend und wartet anschließend bis alle Agenten ihre Arbeit verrichtet haben. Kommt eine Nachricht am Empfängerthread an, wird zunächst der Header ausgelesen, um zu entscheiden wie die Nachricht weiterverarbeitet wird. Die beiden Threads sind Teil einer gemeinsamen Klasse und können damit Informationen über Membervariablen austauschen. Je nach eingegangener Nachricht reagiert der Hauptthread entsprechend. Abhängig vom Nachrichtentyp werden dabei die Agenten unterschiedlich bedient. Das Verhalten bei den verschiedenen Nachrichtentypen wird im Detail im Zusammenhang mit den Algorithmen in Kapitel 5 behandelt. Der Rückwärtszweig der Kommunikation findet ebenfalls über den Broker statt. Sobald alle Agenten ihre Berechnungen abgeschlossen haben, in-

formiert der Hauptthread den Server über das Ergebnis. Da sich die beiden Threads den TCP Socket teilen, muss nicht zuerst der Empfängerthread informiert werden. Die Nutzung des Sockets ist durch wechselseitigen Ausschluss geschützt, das heißt ein Semaphor erwirkt, dass immer nur ein Thread gleichzeitig den Socket nutzen kann.

Die Agenten sind der ausführende Teil des Frameworks. Die aufwändigen Kollisionsabfragen werden auf dieser untersten Ebene durchgeführt. Somit benötigen die Agenten die meiste Rechenleistung. Aus diesem Grund ist pro CPU-Kern des Clients nur ein Agent vorgesehen. Die Agenten erhalten ihre Aufgaben nicht direkt vom Server, sondern vom Hauptthread des Brokers. Der Broker besitzt zweierlei Kollisionsüberprüfungsinstanzen. Es können sowohl einfache Kollisionsüberprüfungen ausgeführt werden, die nur wahr oder falsch liefern, als auch Abstandsabfragen, die die minimale Entfernung zwischen dem Roboter und der Umgebung zurück liefern. Zwar kann auch die Abstandsabfrage eine Kollision erkennen, sie ist aber wesentlich rechenaufwändiger als eine herkömmliche Kollisionsüberprüfung und bietet sich daher nicht immer an. Zur Kommunikation zwischen dem Broker und den Agenten wird ein Shared Memory Segment verwendet. Alle Agenten und der Broker greifen über eine eindeutig definierte ID auf den Speicher zu und können diesen auslesen und beschreiben. Die Schreib- und Lesezugriffe erfolgen nach dem Erzeuger-Verbraucher Muster:

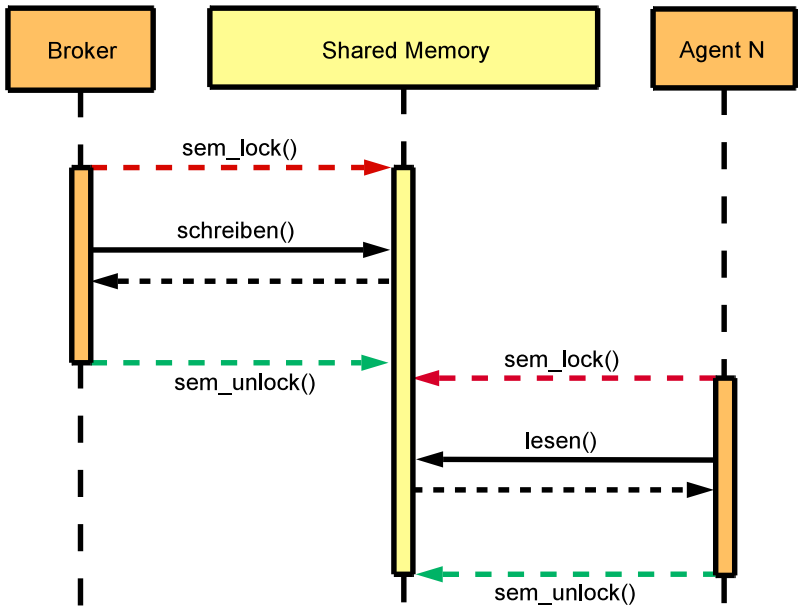


Abbildung 4.4: Kommunikation zwischen Broker und Agent im Erzeuger-Verbraucher Muster. Sobald der Broker den Shared Memory beschrieben hat, entsperrt dieser die Semaphore für den Zugriffsschutz (grüner Pfeil). Kurz darauf sperrt einer der wartenden Agenten die Sempahore zum Lesen (roter Pfeil).

Der Vorgang in Abbildung 4.4 beschreibt das typische Erzeuger-Verbraucher Muster. Der gemeinsam genutzte Speicherbereich kann immer nur von einem der Prozesse verwendet werden. Dabei macht es keinen Unterschied, ob der Prozess lesend oder schreibend zugreift. Dies ermöglicht eine definierte Zugriffskontrolle für den Inhalt des Speichers. So ist es in diesem Beispiel dem Agent nur erlaubt den Speicher auszulesen, sobald dieser vollständig beschrieben ist. Dies gilt im Umkehrschluss auch für den Broker. Der Broker darf erst wieder neuen Inhalt in den Speicher schreiben, sobald der Agent seinen Teil vollständig ausgelesen hat. Unvollständige oder fehlerhafte Daten können so nicht entstehen.

Die Startreihenfolge der Clients ist genau vorgegeben. Der Broker wird mit dem zuvor beschriebenen SSH Aufruf vom Server aus gestartet. Dabei erhält er den Hostnamen und den Port des Servers. Bei der Initialisierung wird damit die Verbindung zum Server hergestellt. Zusätzlich wird ein Shared Memory Segment angelegt, welches die Kommunikation mit den in der Hierarchie darunter liegenden Agenten bereitstellt. Um die Bewegungsplanung ausführen zu können, wird eine OpenRAVE Umgebung initialisiert. Mit dem Fork-Befehl werden anschließend Prozesse abgekapselt, die mittels Exec-Befehl die Agenten starten. Nachdem sich diese mit dem Shared Memory verbunden haben, können die Clients die drei verschiedenen Aufgabentypen entgegen nehmen.

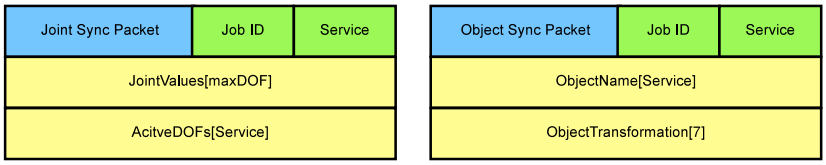
4.2.3 Synchronisation der OpenRAVE Umgebungen

Eine einheitliche Parametrisierung der Planungsumgebung auf jeder Ebene der Softwarearchitektur ist die Voraussetzung für parallele Planungsverfahren. Damit dies erreicht werden kann, müssen die einzelnen Umgebungsrepräsentationen auf Server- und Clientrechnern synchron gehalten werden. Dazu müssen vor jedem Planungsaufruf Synchronisationsnachrichten ausgetauscht werden. Der Commander spiegelt immer die aktuelle Situation des realen Roboters wieder und dient daher als Referenzpunkt für alle anderen OpenRAVE Instanzen. Der Broker erhält seine Synchronisationsnachrichten direkt vom Commander über das Netzwerk und verbreitet diese dann über das Shared Memory an die Agenten. Bei jedem Aufruf des Frameworks müssen die entscheidenden Merkmale der Umgebung synchronisiert werden. Dazu gehören die folgenden Punkte:

- Gelenkwinkelstellungen des gesamten Roboters,
- Aktive Gelenke des Roboters,
- Positionen und Orientierung aller Objekte in der Umgebung,
- und die vom Roboter gegriffenen Objekte.

Der erste und zweite Punkt, sowie der dritte und vierte Punkt können dabei zusammen gefasst werden, sodass insgesamt zwei verschiedene Synchronisationsnachrichten benötigt werden: Zum Einen die Synchronisation der Objekte und zum Anderen die Synchronisation des Roboters. Bei der Synchronisation der Umgebung werden die einzelnen Objekte unabhängig behandelt. Die Synchronisation der Robotergelenkstellungen erfolgt für den ganzen Roboter mit einem Mal, anstatt für jedes Gelenk einzeln. Im Folgenden werden diese zwei Mechanismen beschrieben.

Zur Synchronisation des Roboters ist es wichtig, dass alle Gelenkwinkel erfasst werden. Auch wenn die Gelenke gerade nicht aktiv sind, kann es sein, dass diese extern verändert wurden. Das Paket zur Gelenkwinkel-Synchronisation besteht daher aus zwei Arrays. Das Array, das die Gelenkwinkel beinhaltet, ist immer so groß wie die maximale Anzahl der Gelenkwinkel des entsprechenden Roboters (bei Justin sind dies 43 Werte). Das Array für die aktiven Gelenke ist dynamisch. Die Größe dazu wird im Servicefeld des Headers als Integerzahl mitgeliefert und ist in Abbildung 4.5 in Arrayschreibweise dargestellt (*ActiveDOFs[Service]*). Bei der Synchronisation der Objekte in der Umgebung muss für jedes Objekt ein Paket versendet werden. Dieses beinhaltet den Namen des Objektes, über den das Objekt eindeutig identifizierbar sein muss und dessen Position im Raum. Da der Name unterschiedlich lang sein kann, ist dessen Länge im Servicefeld hinterlegt (*ObjectName[Service]*). Ist ein Objekt als gegriffen deklariert, so wird der Zusatz „*@<Manipulator Name>*“ an den Namen angehängt. Der Broker und die Agenten fügen in diesem Fall das Objekt an den entsprechenden Endeffektor an. Aus diesem Grund muss immer erst der Roboter vor den Objekten synchronisiert werden, da sich sonst die Objekte möglicherweise nicht in der Nähe der Hand befinden, aber trotzdem bewegt werden.



(a) Roboter Synchronisationspaket

(b) Objekt Synchronisationspaket

Abbildung 4.5: Pakete zur Synchronisation des Roboters und der Umgebung. Die oberste Reihe bildet jeweils den Header. Die Daten befinden sich in den Reihen darunter.

Neben diesen offensichtlichen Parametern müssen noch weitere Parameter bei der Synchronisation beachtet werden. Da die Planungsumgebung verteilt agiert, müssen die Parameter zum Konfigurieren des Planers übergeben werden. Im aktuellen Stand des Frameworks wird dies allerdings noch nicht berücksichtigt. Zusätzliche Parameter, wie zum Beispiel Constraint-Funktionen zum Einschränken von gewissen Bewegungsrichtungen während der Planung oder modifizierte Schrittweiten, können daher bislang noch nicht verwendet werden. Außerdem können keine neuen Objekte in die Umgebung eingefügt werden.

5 Parallele Algorithmen für die verteilte Softwarearchitektur

Eine wichtige Fähigkeit des Menschen ist die Manipulation verschiedenster Objekte mit seinen Armen und Händen. Dabei wirken die Interaktionen mit der Umgebung zu jedem Zeitpunkt sehr durchdacht und gleichermaßen geschickt. Obwohl die Bewegungen eines Menschen verhältnismäßig komplex sein können, benötigen wir nur Bruchteile einer Sekunde um diese zu koordinieren, ganz im Gegensatz zu den heutigen Servicerobotern. Selbst die Berechnung einer simplen Trajektorie ist oft eine große Herausforderung. Zur Berechnung eines kollisionsfreien Pfad von einer Startkonfiguration zu einer Zielkonfiguration, benötigen Roboter oft mehrere Sekunden. Eine komplexe Aufgabe, wie in Abbildung 5.1 dargestellt, die gleichzeitig Genauigkeit und eine schnelle Anpassung an eine sich ändernde Umgebung verlangt, ist somit nur schwer lösbar.

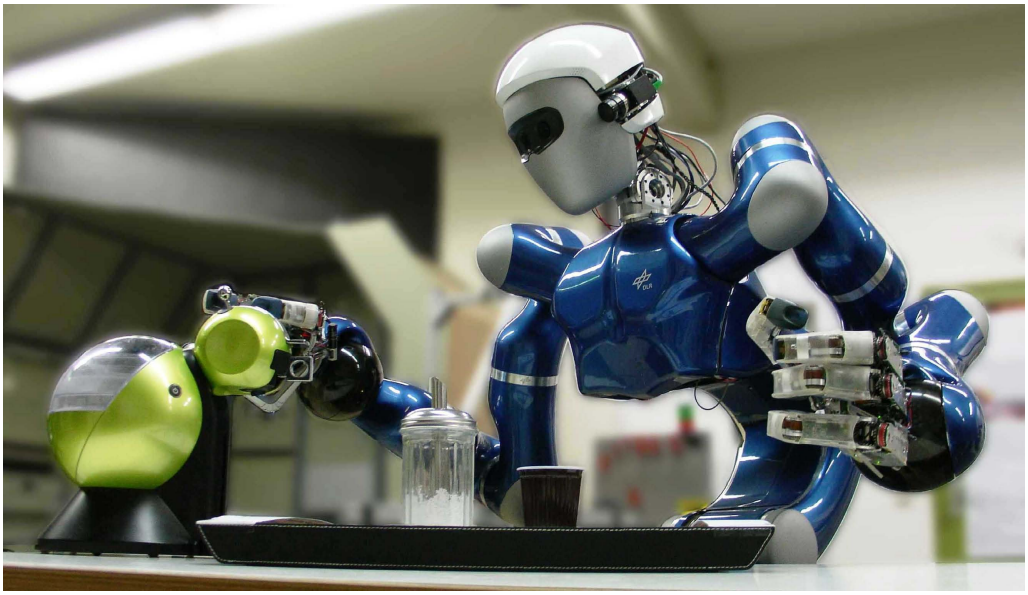


Abbildung 5.1: Kaffee kochen: Eine schwere Aufgabe für Justin.

Zur Minimierung der Gesamtrechendauer müssen die einzelnen Elemente der Pfadplanung beschleunigt werden. Dabei darf die Qualität der Trajektorie nicht unter der verkürzten Rechenzeit leiden. Ein möglicher Weg dies zu erreichen, ist die Parallelisierung eines Algorithmus auf mehrere Prozesse.

5.1 Verteilung des Optimierungsalgorithmus

Die Problemanalyse hat gezeigt, dass die meiste Rechenzeit in der Optimierungsphase benötigt wird. An dieser Stelle werden daher die Verfahren zur Parallelisierung der Optimierung vor der Planung betrachtet. Das Linear Shortcut Verfahren hat sich gut in der Praxis bewährt und wird oft verwendet. In diesem Abschnitt wird daher gezeigt, wie dieser Optimierungsalgorithmus bei gleich bleibender Qualität im Bezug auf die Länge der Trajektorie, effizient beschleunigt werden kann.

5.1.1 Ansätze zur Verteilung des Linear Shortcut Algorithmus

Zur Verteilung des Linear Shortcut Algorithmus auf mehrere Prozessoren und Computer werden zwei Ansätze untersucht. Bei der ersten Methode wird die Abarbeitung eines einzelnen Trajektorienabschnittes parallelisiert. Dabei erhalten mehrere Prozessoren jeweils einen Teilabschnitt des Gesamtsegments. Die zweite Methode beschreibt eine nebenläufige Bearbeitung mehrere unabhängiger Trajektorienabschnitte zur gleichen Zeit. Beide Verfahren bewirken indirekt die parallele Ausführung mehrerer Kollisionsüberprüfungen.

5.1.1.1 Kooperative Überprüfung eines Trajektorienabschnittes

Bei der kooperativen Optimierung eines Trajektorienabschnittes wird ein Teilabschnitt simultan bearbeitet. Das bedeutet, mehrere Prozesse arbeiten an dem selben Trajektoriensegment. Dabei werden, wie beim sequentiellen Linear Shortcut Algorithmus, zunächst zwei zufällige Konfigurationen entlang der Trajektorie gewählt. Ziel ist es, diese Konfigurationen mit einer Gerade zu verbinden, um die Trajektorie zu verkürzen. Allerdings wird bei dieser Methode die Linie nicht von einer einzigen Instanz zur Kollisionsüberprüfung auf Kollision überprüft, sondern gleich von mehreren.

Die Aufteilung auf N Prozesse erfolgt durch eine Teilung der Linie in N Liniensegmente. Dazu müssen $N + 1$ Konfigurationen berechnet und auf die jeweiligen Prozesse verteilt werden. Die Zielkonfiguration der ersten Instanz wird als Startkonfiguration der nächsten Instanz verwendet. Im konkreten Fall des DISPLACE-Frameworks erfolgt die Verteilung der Linien sogar auf zwei Ebenen. Zunächst wird die Linie durch die Anzahl der verfügbaren Clients geteilt. Jeder Client erhält also sein eigenes Liniensegment. In Abbildung 5.2 sind die großen Ringe stellvertretend für diese Aufteilung. Zusätzlich wird die Linie für jeden Agent auf den jeweiligen Clients nochmals aufgeteilt. Diese Aufteilung ist hier mit den kleinen Ringen dargestellt. Alle im System befindlichen Agenten arbeiten also an einem Untersegment einer Linie. Die Agenten A_{11} und A_{12} gehören zum Client 1 und die Agenten A_{21} und A_{22} zu Client 2.

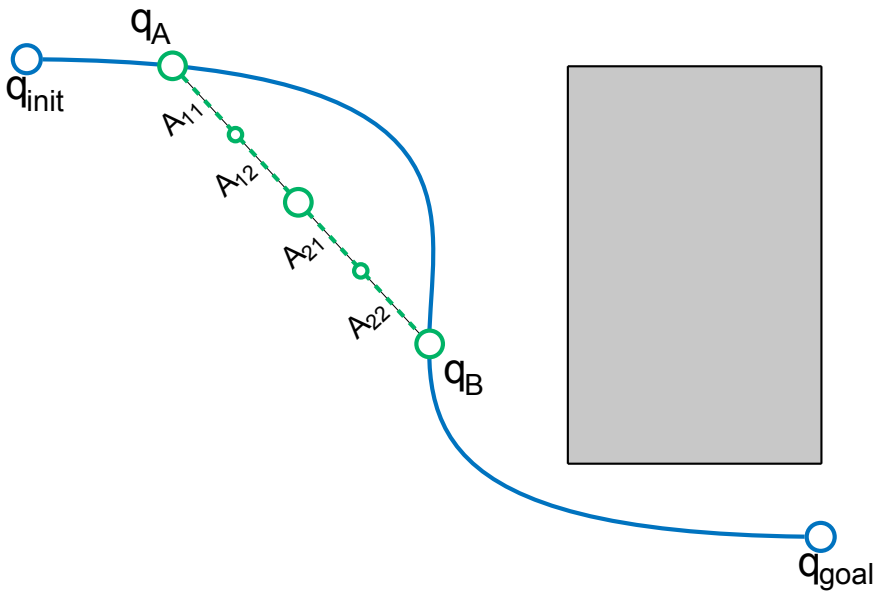


Abbildung 5.2: Verteilte Optimierung eines Trajektorienabschnittes.

Man würde nun erwarten, dass bei der Verwendung der doppelten Anzahl an CPUs auch immer die Berechnungszeit auf die Hälfte absinkt. Leider kann ein solcher linearer Geschwindigkeitsgewinn nicht erreicht werden. Generell erstellt das Linear Shortcut Verfahren nur relativ kurze Linien die auf Kollision überprüft werden müssen. Da die zufällig generierten Pfade des RRT-Planers in den gewählten Szenarien mit einer Länge von durchschnittlich 124 Konfigurationen nur relativ kurz sind, bestehen auch zu überprüfend Linien nur selten aus mehr als 50 Konfigurationen. Die Anzahl an notwendigen Kollisionsüberprüfungen ist also stark limitiert. Der Einsatz von mehreren Agenten kann die Berechnungszeit also nicht beliebig verkürzen. Bei einer beispielhaften Konstellation von vier Clients mit je zwei Agenten würde eine Linie aus 50 Konfigurationen in sechs Kollisionsüberprüfungen pro Agent resultieren. Die Zeit, die sechs Kollisionsüberprüfungen beanspruchen, ist allerdings nahe an der Zeit, die allein die Verteilung der Linie und die Kommunikation beansprucht. Sobald also das Verteilen einer Linie mehr Zeit beansprucht als die Kollisionsabfragen selbst, wird die Parallelisierung unwirtschaftlich.

5.1.1.2 Nebenläufige Überprüfung mehrerer unabhängiger Trajektorienabschnitte

Eine weitere Möglichkeit zur Parallelisierung des Algorithmus ist die nebenläufige Verarbeitung mehrerer Trajektorienabschnitte. Das bedeutet, es werden mehrere Teilabschnitte auf einmal auf Kollisionen überprüft. Die Prozesse arbeiten dabei völlig unabhängig. Mehrere Instanzen der eigentlichen Funktion zur Kollisionsüberprü-

fung sind dazu nötig. Jede dieser Instanzen erhält einen unabhängigen Abschnitt der Trajektorie zur Bearbeitung. Diese dürfen sich unter keinen Umständen kreuzen oder einschließen, da sonst die Teilergebnisse ungültig wären. Damit dies sichergestellt werden kann, müssen die einzelnen Start- und Zielkonfigurationen der jeweiligen Abschnitte nacheinander ausgewählt und an die noch freien Instanzen verteilt werden. So erhalten die einzelnen Kollisionsüberprüfer nach und nach einen Abschnitt der Trajektorie zur Überprüfung. Dabei wird zwischen der Start- und Zielkonfiguration eine Linie gebildet und in diskreten Abschnitten auf Kollision überprüft. Die Linie ist dabei eine Linie im Konfigurationsraum. Meldet sich eine der Instanzen zurück, wird bei einem positiven Ergebnis ohne Kollision der alte Teilabschnitt durch die neue Linie ersetzt. Abbildung 5.3 illustriert die nebenläufige Abarbeitung der Trajektorie noch einmal in einem 2-dimensionalen Beispiel.

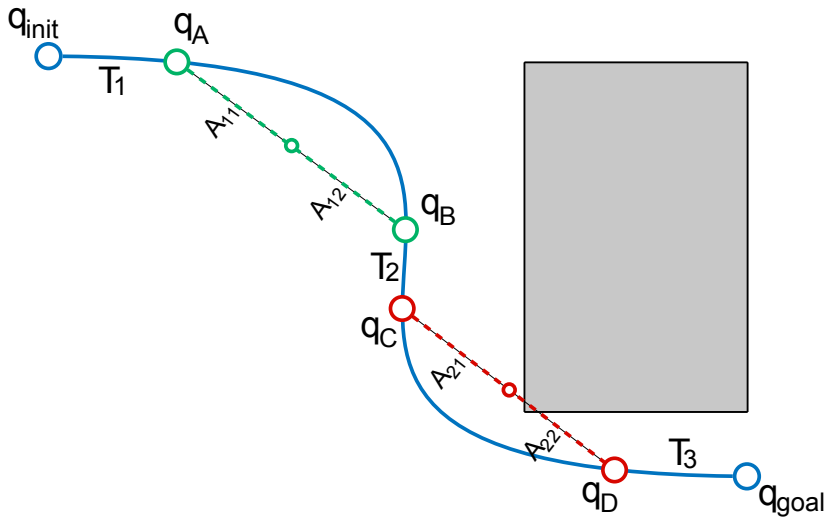


Abbildung 5.3: Nebenläufige Optimierung mehrerer Teilabschnitte einer Trajektorie.

Dabei stellt die grüne Verbindung eine erfolgreiche Verkürzung des Weges durch eine Linie dar, die zwischen den Punkten q_A und q_B auf Kollisionen überprüft wurde. Der rote Pfad deutet einen gescheiterten Versuch an, q_C mit q_D zu verbinden. Konkret wird in diesem Fall der Parallelisierung pro Client in der Softwarearchitektur ein Liniensegment bearbeitet. Auf dem Client teilt der Broker entsprechend der CPU Zahl die Linie erneut auf. Die kleineren Kreise stellen dabei die internen Grenzen für die Agenten dar. In diesem Beispiel wurden also zwei Agenten auf zwei unabhängigen Clients verwendet. Auch hier gehören die Agenten A_{11} und A_{12} zu Client 1 und die Agenten A_{21} und A_{22} zu Client 2. Das graue Hindernis liegt dabei zwischen den beiden ausgewählten Konfigurationen des roten Pfades, wodurch keine direkte Verbindung möglich ist. Beide Anfragen können ohne Wechselwirkungen parallel abgearbeitet werden.

5.1.1.3 Vergleich der parallelen Überprüfungsverfahren

Das nebenläufige Verfahren verspricht eine schnelle Optimierung. Leider kann es das Optimierungsziel nicht einhalten. Die daraus resultierenden Linien sind kaum kürzer. Man stelle sich einen 3. Client vor, der einen weiteren Trajektorienabschnitt optimieren soll. Abbildung 5.3 verdeutlicht, dass die möglichen Abschnitte, die zur Auswahl für die Optimierung stehen, begrenzt sind. Es steht nur noch eine Untermenge des möglichen Pfades zur Verfügung. Bei mehreren simultan arbeiteten Prozessen ist der Pfad stark fragmentiert: Es können nur noch die Zwischensegmente T_1, T_2 und T_3 optimiert werden. Versucht man einen dieser Abschnitte zu optimieren, findet sich zwar in jedem Fall eine gültige Linie, jedoch bringt die Optimierung dieses Abschnittes kaum Vorteile bezüglich der Gesamtlänge. Dieser Ansatz zur Parallelisierung führt zwar zu sehr kurzen Berechnungszeiten, allerdings ohne relevante Optimierungsergebnisse. Insbesondere Schleifen werden aus Pfaden schlechter entfernt. Der normale Linear Shortcut Algorithmus ist sehr effektiv, da er redundante Bewegungen vor allem im kollisionsfreien Raum sehr gut wegglätten kann. Dies liegt daran, dass die zufällig gewählten Start- und Zielkonfigurationen zuweilen lange Wege umschließen und dementsprechend ein hohes Potential an Verbesserung bieten. Zwei Konfigurationen abzukürzen, die bereits nahezu durch eine Gerade im C_{space} verbunden sind, bringt hingegen wenig Gewinn.

Von den parallelen Algorithmen zeigt nur die erste kooperative Methode dieses Verhalten. Die kooperative Variante zur Parallelisierung wird daher im späteren Verlauf zur Optimierung verwendet.

5.1.2 Zusätzliche Anpassungen des Optimierungsverfahrens

Zur weiteren verringerung der Optimierungszeit, wird ein Phänomen ausgenutzt, welches bei der Entwicklung der parallelen Optimierungsverfahren beobachtet wurde: Bei beiden Methoden zur Verteilung des Linear Shortcut Algorithmus ist ein interessanter Effekt zu beobachten. Da die verschiedenen Instanzen an mehreren Stellen gleichzeitig mit der Überprüfung auf Kollisionen beginnen, ist die Wahrscheinlichkeit größer relativ schnell auf ein Hindernis zu stoßen, falls ein solches existiert. In Abbildung 5.3 kann beobachtet werden, dass bei der Abarbeitung des roten Liniensegments frühzeitig eine Kollision erkannt werden kann. Der zweite der beiden Agenten erreicht schon nach kurzer Zeit das Hindernis und meldet eine Kollision an den Broker. Die anderen Agenten werden dann über einen frühzeitigen Abbruch informiert und beenden ebenfalls ihren Dienst. Somit kann zeitnah ein neuer Trajektorienabschnitt bearbeitet werden. In diesem Fall ist der Geschwindigkeitsanstieg für einen einzelnen Optimierungsschritt *superlinear*[5].

[5]Superlinearität liegt dann vor, wenn eine Folge schneller als linear konvergiert. Im Falle der Abarbeitung eines Liniensegments bedeutet dies, ein Ergebnis steht fest (Kollision) bevor die Agenten die Linie vollständig abgearbeitet haben.

Die Verteilung auf N unabhängige Prozesse lässt eine eventuell vorhandene Kollision im schlechtesten Falle N-mal schneller finden, wenn erst das letzte Element des Liniensegments eine Kollision aufweist. Im besten Fall kann eine Kollision bereits nach einer einzelnen Kollisionsüberprüfung entdeckt werden, falls das erste Element des Liniensegments kollisionsbehaftet ist. Da die Hindernisse, die sich in der realen Welt befinden, als Volumen beschrieben werden können, sind nicht nur einzelne Konfigurationen von einer möglichen Kollision betroffen, sondern gleich ganze Trajektorienabschnitte. Diese Eigenschaft kann ausgenutzt werden, um mögliche Hindernisse möglichst schnell zu erkennen. Ähnlich der binären Suche für sortierte Listen, kann auch im Fall einer zu optimierenden Trajektorie ein effektiverer Suchalgorithmus eingesetzt werden. Ziel ist es, eine bessere Reihenfolge für die zu überprüfenden Konfigurationen zu finden. Abbildung 5.4 zeigt den Vorgang in Einzelschritten.

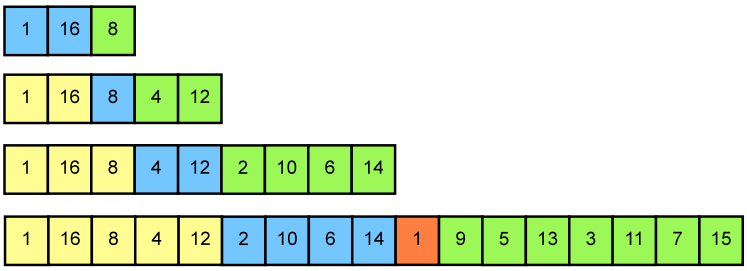


Abbildung 5.4: Schematisches Beispiel der erschöpfenden Suche. Blaue Zahlen sind neue Elemente in der Liste. Grüne Elemente berechnen sich aus den blauen Zahlen. Gelbe Zahlen befinden sich schon länger in der Liste. Rote Zahlen befinden sich bereits in der Liste und werden daher nicht noch einmal benötigt.

Beginnend mit dem ersten und dem letzten Element des Liniensegments verfolgt die Suche das Ziel eine Kollision mit einem Hindernis möglichst schnell zu detektieren. Diese beiden Konfigurationen bilden den Anfang für die Suche. Die Indizes der beiden Elemente werden in eine Liste eingetragen. In Abbildung 5.4 sind dies die blauen Zahlen in der ersten Reihe. Diese beiden Werte werden dann aufaddiert und durch zwei dividiert. Die erhaltene Zahl wird in die Liste angehängt (grün) und als Ausgang für die nächste Berechnung verwendet. Alle neu angehängten Zahlen in der zweiten Reihe (blau) werden erneut einmal mit dem ersten Index und einmal mit dem letzten Index addiert und wieder jeweils durch zwei geteilt. Wiederum werden alle neu berechneten Zahlen an die Reihe angehängt (grün), es sei denn die Zahl ist bereits in der Liste vorhanden (rot). So entsteht nach und nach die Reihenfolge für die Abarbeitung der einzelnen Konfigurationen. Anders als eine binäre Suche ist die Suche nach einem Hindernis in einem Trajektorienabschnitt eine *exhaustive search*. Eine exhaustive search (dt.: erschöpfende Suche) ist eine Suche, bei der im ungünstigsten Fall jedes Objekt der Reihe untersucht werden muss. Bei der Optimierung ist dies jedoch nicht negativ zu sehen, da sowieso alle Konfigurationen auf Kollision überprüft werden müssen.

Eine zusätzliche Heuristik basiert auf einer Beobachtung während der Ausführung verschiedener Trajektorien. Mit dem Wissen über eine optimale Bahn als Darstellung einer Geraden von Startkonfiguration zu Zielkonfiguration im C_{space} kann berechnet werden, wie groß der entsprechende Gelenkwinkelweg in *rad* ausfallen würde, falls eine lineare Verbindung möglich ist. Dieser Wert kann als Referenzmaß für eine optimale Trajektorie verwendet werden. Manche Pfade können als Gerade betrachtet werden. Es ist daher zunächst sinnvoll, die Bahn auf eine einfache Gerade hin zu untersuchen. Jedoch befindet sich oft am Ende ein zu greifendes Objekt, welches den Weg blockiert. Dennoch beschreiben auch solche Pfade meist annähernd eine Gerade. Das Optimum der Gerade wird allerdings nie erreicht. In diesem Fall kann die Optimierung frühzeitig abgebrochen werden. Nähert sich das Verfafren also nach einigen Iterationen bereits dem Optimum bis auf eine gewisse Toleranzgrenze, kann das Verfahren frühzeitig abgebrochen werden. Beobachtungen haben gezeigt, dass bereits eine Annäherung an 30% des optimalen Werts eine zufriedenstellende Bahn generiert.

5.1.3 Validierung des verteilten Optimierungsverfahrens

Dieser Abschnitt befasst sich mit der Validierung der entwickelten kooperativen Strategie zur parallelen Pfadoptimierung. Dazu werden drei unterschiedliche Szenarien verwendet.

- Szenario 1: Ein leeres Szenario ohne Hindernisse,
- Szenario 2: Ein Szenario mit nur einem Hindernis,
- Szenario 3: Das Küchenszenario als komplexes Beispiel.

Aufbau und Ablauf aller Szenarien werden im Folgenden beschrieben und ausgewertet. Anhand der Gegenüberstellung der Ergebnisse soll herausgefunden werden, wie stark die Verringerung der Rechenzeit durch das verteilte Optimierungsverfahren abhängig von der Szene und der Pfadlänge ist.

5.1.3.1 Versuchsaufbau

In allen drei Szenarien hat Justin die gleiche Ausgangsposition. Die Konfiguration des Roboters weicht nicht von der des Referenzszenarios aus Kapitel 3.1 ab und ist in Tabelle 3.1 einzusehen. Im ersten Szenario befindet sich Justin in einer leeren Umgebung. Kein Hindernis blockiert den Weg des Roboters. Ziel dieses Szenarios ist es, herauszufinden wie lange eine Optimierung ohne den Einfluss fremder Objekte dauert. Der Aufbau ist im linken Teil von Abbildung 5.5 zu sehen.

Das zweite Szenario beinhaltet den bereits bekannten Tisch aus dem Referenzszenario der Problemanalyse, sowie einen zusätzlichen Pfeiler auf dem Tisch als Hindernis. Der Pfeiler befindet sich an der Position X = 0,70 m und Y = -0,32 m und ist vom Tisch ab 0,8 Meter hoch. Der Aufbau ist im rechten Teil von Abbildung 5.5 zu sehen.

Als drittes Szenario wird das Referenzszenario aus der Problemanalyse herangezogen. Der Aufbau ist der gleiche. Das Küchenszenario bestehend aus dem Tisch, der Kaffeemaschine und dem Tablett als Hindernisse und dem Kaffeebecher und dem Zuckerstreuer als Zielobjekte. Die genaue Anordnung aller Szenarien kann in Anhang A.1 eingesehen werden.

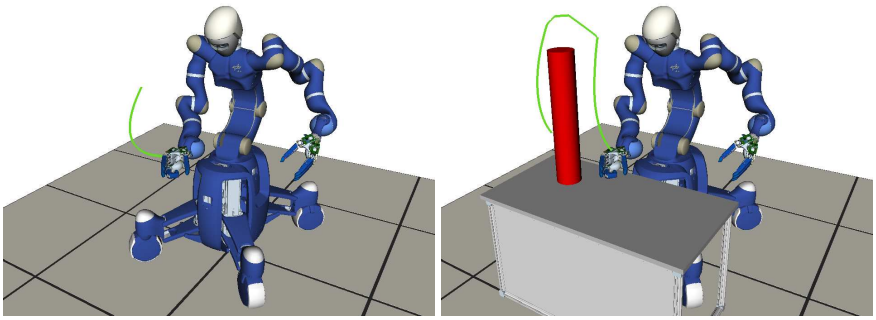


Abbildung 5.5: Szenarien zur Validierung der parallelen Algorithmen. Links das leere Szenario 1 und rechts Szenario 2 mit dem Pfeiler auf dem Tisch.

5.1.3.2 Versuchsablauf

Bei den hier untersuchten Szenarien ist nur der Optimierungsvorgang von Bedeutung. Da die Optimierung wie auch die Pfadplanung jedoch zufallsbasiert arbeiten, werden die Ausgangstrajektorien zur Optimierung bei jedem Schritt neu geplant.

Im ersten Szenario muss Justin lediglich seinen rechten Arm von einer Konfiguration q_{init} zu einer zweiten Konfiguration q_{goal} bewegen. Da sich keine Hindernisse im Raum befinden, ist zu erwarten, dass die Planung relativ schnell erfolgen kann und die Bahn in einer Geraden im Gelenkwinkelraum resultiert. Im zweiten Szenario muss Justin die gleiche Bewegung durchführen. Dabei ist jedoch der Pfeiler im Weg. Da dieser auf dem Tisch steht und ziemlich nahe bei Justin positioniert ist, muss dieser nach oben hin umfahren werden. Die erwarteten Trajektorien sind in Abbildung 5.5 eingezeichnet. Im dritten Szenario muss Justin, wie schon beim Referenzszenario, die Küchenumgebung aufräumen.

Der komplette Ablauf jedes Szenarios wird 20 mal wiederholt. Im Vergleich stehen das sequentielle Linear Shortcut Verfahren und das kooperative Verfahren. Das kooperative Verfahren wird mit mehreren Konstellationen der Softwarearchitektur ausgeführt.

5.1.3.3 Ergebnisse

Die gemessenen Zeiten in Tabelle 5.1 zeigen, dass die Optimierung durchaus unter der Verwendung von parallelen und verteilten Architekturen beschleunigt werden kann. Allerdings ist der Geschwindigkeitsgewinn nicht linear. Die Zeiten aus Szenario 2 und 3 zeigen, dass mit acht Agenten etwa ein Geschwindigkeitsanstieg um Faktor vier erreicht werden kann. Im Umkehrschluss beträgt die Parallelisierungsrate nach Amdahl etwa 90%. Für Szenario 2 ist keine große Verbesserung zu erkennen, da die Pfade sehr kurz sind.

	sequentiell	parallel 1x2	parallel 2x2	parallel 4x2
Szenario 1	1,36	1,22	1,16	1,13
Szenario 2	7,20	4,32	3,70	2,42
Szenario 3	6,78	3,12	2,42	1,76

Tabelle 5.1: Gemittelte Zeiten in Sekunden für die reine Optimierung der Szenarien. Der parallele Algorithmus wurde auf 1 Client mit 2 Agenten, auf 2 Clients mit je 2 Agenten und auf 4 Clients mit je 2 Agenten aufgeteilt.

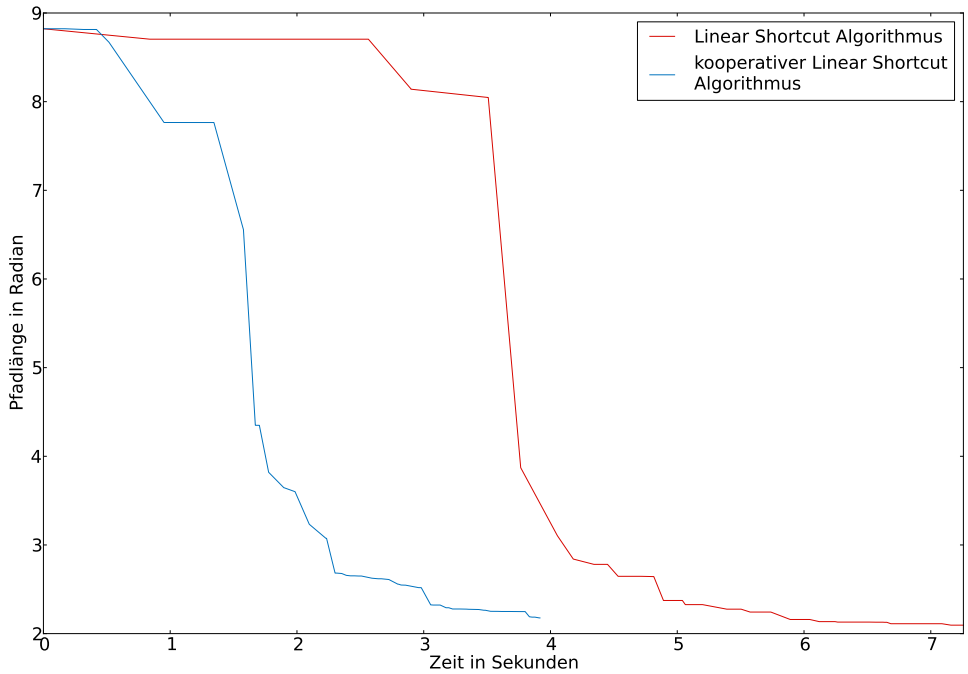


Abbildung 5.6: Paralleler und sequentieller Linear Shortcut Algorithmus im Vergleich beim 3. Szenario. Der parallele Algorithmus (blau) ist mit zwei Agenten auf einem Client annähernd doppelt so schnell wie der sequentielle Algorithmus (rot).

Die kooperative Variante des Linear Shortcut Verfahrens unterscheidet sich nur in der Parallelisierung von der Originalversion. Der eigentliche Ablauf bleibt der Gleiche. Somit bleiben die Vorteile beider Methoden erhalten. Das Verfahren konvergiert schneller und die Trajektorienlänge erreicht ein ähnliches Niveau wie beim sequentiellen Algorithmus. Abbildung 5.6 zeigt, dass bei einer Aufteilung auf zwei Agenten auf einem Client nahezu die Hälfte der Zeit eingespart werden kann, ohne dabei das Ergebnis der Optimierung zu beeinflussen. Allerdings können aufgrund von Kommunikationsoverheads und einer begrenzten Größe der Trajektoriensegmente nicht beliebig viele Instanzen verwendet werden.

5.2 Verteilung der Abstandsabfragen

Die Abstandsabfrage, als letzter Schritt des Planungszykluses, kann ebenfalls in die Softwarearchitektur zur Parallelisierung integriert werden. Ziel der Abstandsabfrage ist die Anreicherung der Trajektorie mit zusätzlichen Abstandsinformationen über den minimalen Abstand des Roboters zur Umgebung. Dazu muss der Abstand für jede Konfiguration der Trajektorie berechnet werden. Im Normalfall müsste dazu die gesamte Trajektorie sequentiell abgearbeitet werden. Unglücklicherweise ist der Algorithmus zur Abstandsabfrage sehr zeitaufwändig und benötigt für die Berechnung eines Abstandes bereits bis zu 21 Millisekunden. Eine normale Kollisiononsabfrage benötigt nur etwa 4 Millisekunden. Prinzipiell werden dabei rekursiv alle Dreiecke der verschiedenen Objekte in der Umgebung mit den Dreiecken des Roboters verglichen. Abhängig von der Pose des Roboters kann berechnet werden, wie groß der euklidische Abstand zwischen zwei Dreiecken der verschiedenen Objekte ist [23]. Zwar können Hierarchieebenen und Caching-Methoden verwendet werden, um die Teilberechnungen zu beschleunigen, dennoch benötigt die gesamte Berechnung des minimalen Abstandes für eine gesamte Trajektorie je nach Szenario immer noch mehrere Sekunden. Die Berechnungsdauer steigt dabei, unter der Vernachlässigung des Caches, mit jedem weiteren Dreieck linear [23].

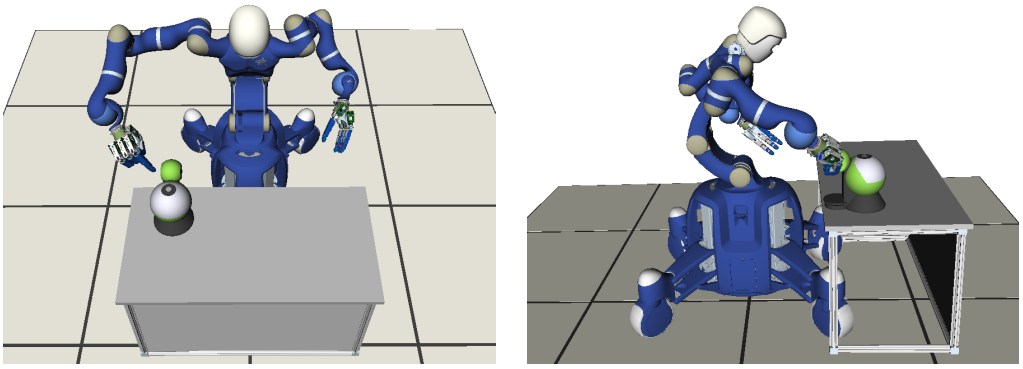
5.2.1 Parallelisierung der Abstandsabfragen

Die Parallelisierung der Abstandsabfrage verfolgt ein einfaches Prinzip: Schon eine Berechnung des minimalen Abstandes zwischen Roboter und Umgebung ist sehr aufwändig. Bereits eine einzige Berechnung kann ohne merkliche kommunikationsbedingte Zeitverluste in einem eigenen Prozess berechnet werden. Unter der Verwendung des DISPLACE-Frameworks lässt sich dies besonders effizient erreichen. Jeder Agent im Framework wird mit der Berechnung eines Abstandes zu jeweils einer Konfiguration des Pfades beauftragt. Der Broker dient dabei lediglich als Vermittler. Eingehende Abstandsabfragen werden an freie Agenten verteilt. Liefert ein Agent einen Abstand zu einer bestimmten Konfiguration, so wird diese vom Broker zurückgeschickt. Der Commander schreibt den Abstand an die korrekte Stelle in eine Liste. Die Agenten werden solange bedient, bis alle Abstände berechnet wurden.

5.2.2 Validierung des Verfahrens zur verteilten Abstandsabfrage

Das Ziel bei der Validierung der Methoden zur Abstandsabfrage ist es herauszufinden, wie abhängig der Zeitgewinn von der Entfernung zwischen Roboter und Objekten ist. Zudem soll das Verhalten bei zunehmender Parallelisierung untersucht werden. Dazu wird nicht das Referenzbeispiel verwendet. Beim Referenzbeispiel variieren die Abstände zu stark und die Trajektorien sind nicht genau definiert. Die Länge kann sich je nach Ausführung unterscheiden. Stattdessen wurde ein Ablauf gewählt, bei dem der minimale Abstand zwischen Roboter und den Hindernissen nicht sonderlich stark variiert. Die Längen der Trajektorien sind zudem genau bestimmt. Der verwendete Algorithmus nach Larsen et. al [23] wird innerhalb des PQP-Plugins von OpenRAVE benutzt, um die minimalen Abstände zu bestimmen.

5.2.2.1 Versuchsaufbau



(a) Draufsicht (b) Seitenansicht

Abbildung 5.7: Ansichten des Szenarios zur Validierung der Abstandsabfragen.

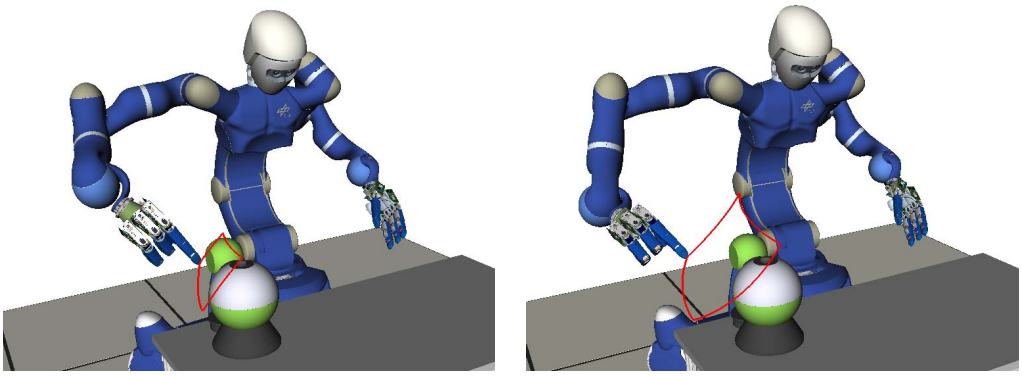
Der Aufbau ähnelt dem des Referenzbeispiels, die meisten Objekte wurden jedoch entfernt. Einzig der Tisch und die Kaffeemaschine befinden sich in der Umgebung des Roboters. Die Kaffeemaschine bildet das Hindernis, zu dem der Abstand am geringsten ist. Ähnlich wie beim Referenzszenario steht Justin wieder in einer Ausgangsstellung vor dem Tisch. Dabei ist die rechte Hand in einer Zeigegeste konfiguriert. Der Zeigefinger der Hand ist damit zu jedem Zeitpunkt der Trajektorie das nächste Objekt zum Hindernis. Abbildung 5.7 zeigt den Aufbau in der Draufsicht und in der Seitenansicht. Die Kaffeemaschine ist auf dem Tisch in Richtung Justin verschoben. Die Koordinaten der Maschine liegen bei X = 0,600 m, Y = -0,400 m und Z = 0,766 m. Die Kaffeemaschine ist 90° um die X-Achse gedreht. Sie zeigt also genau in die Richtung des Roboters. Die Konfiguration des Roboters selbst kann in Tabelle 5.2 eingesehen werden. Der vollständige Aufbau ist im Anhang A.1 aufgestellt.

Körperteil	Gelenkwinkelparameter in Grad
Torso	0,0 -48,9 77,2
Rechter Arm	-2,4 -36,1 -22,5 77,2 18,8 -39,3 38,9
Rechte Hand	-12,2 55,1 81,8 0,0 34,3 0,0 0,0 55,1 81,8 0,0 55,1 81,8
Linker Arm	-24,4 -89,9 5,0 90,0 35,0 -9,9 39,9
Linke Hand	7,7 15,3 0,0 0,0 21,2 0,0 0,0 21,2 0,0 0,0 21,2 0,0
Kopf	0,0 13,0

Tabelle 5.2: Startkonfiguration für das Szenario zur Validierung der Abstandsabfragen.

5.2.2.2 Versuchsablauf

Da gezeigt werden soll, welche Rolle der Abstand beim Ermitteln des minimalen Abstands spielt, ist dieser entlang der gesamten Trajektorie stets zu einem gewissen Grad konstant. Das Experiment besteht aus zwei Durchgängen. Bei beiden Durchgängen soll der Manipulator in einem definierten Abstand die Kontur der Kaffeemaschine abfahren. Im ersten Durchgang fährt der Manipulator sehr nahe an der Kaffeemaschine vorbei. Der mittlere Abstand entlang der Trajektorie entspricht 0,0162 m (siehe Abbildung 5.8a). Die Länge beträgt 100 Konfigurationen. Die Länge des Pfades im zweiten Durchgang beträgt 200 Konfigurationen bei denen sich der Mani-



(a) enge Ausführung (b) weite Ausführung

Abbildung 5.8: Ablauf des Szenarios zur Validierung der Abstandsabfragen. Im ersten Durchgang bewegt sich der Manipulator eng an der Kaffeemaschine vorbei (0,0162 m) und im zweiten Durchgang relativ weit (0,129 m). Die rote Linie steht dabei jeweils für die Position des Zeigefingers der rechten Hand. Diese Punkte sind beim Verfahren entlang der Trajektorie die Punkte mit dem kürzesten Abstand zur Kaffeemaschine.

pulator in einem mittleren Abstand von 0,129 m entlang der Kaffeemaschine bewegt (siehe Abbildung 5.8b). Die rote Linie entspricht dabei der Folge aus Punkten des geringsten Abstandes zwischen Roboterfingerspitze und Kaffeemaschine entlang der Trajektorie. Entlang der gesamten Trajektorie ist dieser Punkt an der Spitze des Zeigefingers. Während des Experiments wird der minimale Abstand zwischen Roboter und Hindernis bei jeder Konfiguration der Trajektorie gemessen. Die verstrichene Zeit wird dabei aufgezeichnet. Am Ende wird die Gesamtdauer berechnet und über 20 Versuche gemittelt. Wie auch bei den anderen Versuchen wird der Versuch mit einer sequentiellen Ausführung und mehreren parallelen Konstellationen der Rechnerarchitektur wiederholt.

5.2.2.3 Ergebnisse

Damit überprüft werden kann, wie abhängig die Abstandsabfrage von der Distanz zu den Hindernissen ist, wurden zuerst die Zeiten für die Berechnung eines einzelnen minimalen Abstandes an einem beliebigen Punkt der Trajektorie bestimmt. Dazu wurden alle Zeiten für die Berechnung des Abstandes einzeln aufgenommen. Die Ergebnisse wurden aufsummiert und durch die Anzahl der Konfigurationen geteilt. Interessant ist hierbei, dass die Zeiten für den ersten Durchgang mit dem engen Verfahrweg, beinahe nicht von den Zeiten für den weiten Verfahrweg abweichen. Im ersten Durchlauf betrug die gemittelte Zeit 0,0218 Sekunden und im zweiten Durchlauf 0,0213 Sekunden. Die Berechnungszeit ist also nicht abhängig von der Entfernung der Objekte. Dies liegt unter anderem daran, dass bei der Abstandsabfrage jedes Dreieck untersucht werden muss, um den minimalen Abstand zwischen Roboter und Umgebung zu ermitteln. Dabei ist es egal, ob sich die Objekte nahe am Roboter befinden oder nicht, da keine hierarchischen Kollisionsmodelle[6] verwendet werden.

Der zweite Teil der Auswertung befasst sich mit der Länge der Trajektorie. Es soll festgestellt werden, ob sich die Berechnung der Abstandsabfragen tatsächlich mit der Länge der Trajektorie linear verändert. Dazu wurden die beiden bereits vorgestellten Durchläufe so erstellt, dass die kurze Trajektorie 100 Konfigurationen beinhaltet und die lange Trajektorie genau doppelt so viele. Gleichzeitig wird beobachtet, ob sich auch bei der Verwendung mehrerer Prozesse die benötigte Zeit zur Berechnung linear senken lässt.

Die Zeiten in Tabelle 5.3 verdeutlichen, dass zumindest für die zwei verwendeten Trajektorien eine lineare Abhängigkeit im Bezug auf deren Länge besteht. Der sequentielle Algorithmus benötigt doppelt so lange zur Berechnung der weiten Trajektorie, wie zur Berechnung der engen und damit kürzeren Trajektorie. Außerdem

[6] Hierarchische Kollisionsmodelle besitzen mehrere Auflösungsstufen. Je nach Entfernung zu einander variiert die Genauigkeit der Modelle die auf Kollision hin überprüft werden. Im besten Falle müssen so nur sehr grobe Modelle gegeneinander geprüft werden, um festzustellen, dass keine Kollision besteht.

kann beobachtet werden, dass auch bei der Verwendung paralleler Techniken im Ansatz ein linearer Verlauf erkennbar ist. Der Kommunikationsoverhead bremst die Parallelisierung jedoch besonders bei kürzeren Trajektorien zunehmend ein.

	sequentiell	parallel 1x2	parallel 2x2	parallel 4x2
100 Punkte	2,210	1,239	0,706	0,470
200 Punkte	4,525	2,298	1,236	0,779

Tabelle 5.3: Gemittelte Zeiten in Sekunden für die Berechnung des minimalen Abstandes entlang des Pfades. Der parallele Algorithmus wurde auf 2 Agenten, auf 2 Clients mit je 2 Agenten und auf 4 Clients mit je 2 Agenten aufgeteilt.

5.3 Verteilung des Planungsalgorithmus

Da die Parallelisierung der Pfadplanung bereits mehrfach in der Forschung untersucht wurde, wird die dafür vorgesehene Integration an diesem Punkt der Arbeit als letztes erläutert. Die Algorithmen dafür sind bereits in Abschnitt 2.2.3.1 des Standes der Forschung zu finden und dort ausführlich beschrieben.

5.3.1 Parallelisierung des Planungsalgorithmus

Das parallele Verfahren zur Pfadplanung ist im jetzigen Stadium nur teilweise implementiert. Bisher wurde lediglich das ODER-Paradigma verwendet um die Planung zu parallelisieren. Der kooperative Aufbau eines Suchbaumes ist noch nicht realisiert. Das ODER-Paradigma wird lediglich auf Ebene der Broker realisiert. Die Anzahl der Agenten ist damit nicht ausschlaggebend für die Planungszeit. Jeder Client baut einen Baum auf und versucht damit selbstständig das Planungsproblem zu lösen. Der Broker nimmt dabei die Aufgabe vom Commander in Empfang. Auf jedem Client wird die gleiche Anfrage bearbeitet. Die Konfigurationen q_{init} und q_{goal} sind also auf jedem Client gleich.

Sobald einer der Broker einen möglichen Pfad gefunden hat, leitet dieser die Nachricht als Erfolgsmeldung weiter an den Commander, sendet allerdings noch keine konkreten Informationen über den gefundenen Pfad. Der Commander entscheidet zunächst, ob der Pfad angefordert werden soll oder nicht. Dies dient der Synchronisation auf der Clientebene. Falls zwei Clients gleichzeitig fertig werden, darf nur der erste der Clients einen Pfad liefern. Auf die restlichen Clients wird nicht gewartet. Mit dem Erhalt der Erfolgsmeldung des Clients werden die restlichen Clients beendet und der Pfad wird angefordert.

5.3.2 Validierung des Verfahrens zur verteilten Pfadplanung

Zur Validierung der parallelen Methode zur Pfadplanung wurden die gleichen Szenarien verwendet wie zur Validierung der Optimierungsstrategie (Abschnitt 5.1.3.1). In diesem Experiment wird der reine Planungsvorgang untersucht. Die Pfade werden also nur geplant und nicht optimiert. Das Experiment für jedes Szenario wurde dabei jeweils 20 mal wiederholt.

Zusätzlich zur reinen Planungszeit t ist in Tabelle 5.4 noch die Standardabweichung σ der Planungszeiten angegeben. Für das ODER-Paradigma ist die Standardabweichung ein wichtiger Kennwert. Zwar kann die mittlere Zeit nicht signifikant reduziert werden, allerdings sinkt die Standardabweichung enorm. Ausreißer bei der Planungszeit können so auf ein Minimum reduziert werden. Von ehemals 4,24 Sekunden im dritten Szenario sinkt diese auf bis zu 2.30 Sekunden unter der Verwendung von 4 Clients. Dadurch kann die mittlere Planungszeit auf immerhin 4,21 Sekunden reduziert werden, was bereits einer Geschwindigkeitssteigerung von 30% entspricht. Diese Beobachtung kann auch für das zweite Szenario gemacht werden. Im ersten Szenario ist der zu suchende Pfad zwar bereits ziemlich kurz, die parallele Ausführung der Planung bringt jedoch auch hier Vorteile.

		sequentiell	parallel 1x2	parallel 2x2	parallel 4x2
Szenario1	t	3,60	3,49	2,88	2,64
	σ	1,03	0,85	0,49	0,28
Szenario2	t	11,43	10,69	9,06	7,16
	σ	4,25	4,82	3,11	1,61
Szenario3	t	6,35	6,50	5,13	4,21
	σ	4,24	4,37	2,84	2,30

Tabelle 5.4: Gemittelte Planungszeiten t und Standardabweichungen σ in Sekunden für die reine Pfadplanung der Szenarien. Der parallele Algorithmus wurde auf einem Client mit 2 Agenten, auf 2 Clients mit je 2 Agenten und auf 4 Clients mit je 2 Agenten aufgeteilt, wobei die Agenten hier keine Auswirkung auf die Rechendauer haben.

Aus der Arbeit von Carpin et. al [8] ist bekannt, dass die kooperative Bearbeitung eines Planungsproblems durch mehrere Prozesse einen linearen Geschwindigkeitsanstieg erzielt. Mit den in diesem Experiment erhaltenen Zahlen kann also davon ausgegangen werden, dass eine zusätzliche Parallelisierung der Pfadplanung auf Ebene der Agenten zu einem größeren Geschwindigkeitsgewinn führt. Hierbei würden die Agenten den Baum kooperativ aufbauen. Unter der Annahme, dass bis zu vier Agenten pro Client gestartet werden können, kann die eigentliche Pfadplanung bereits annähernd in einer Sekunde erfolgen.

5.4 Auswertung der parallelen Algorithmen

Die Integration der Algorithmen zur parallelen Pfadplanung und Pfadoptimierung hat gezeigt, dass die Verteilung der Planungsaufgabe auf mehrere Rechner und Kerne durchaus große Auswirkungen auf die Gesamtdauer der Berechnung hat. Auch in Kombination mit der parallelen Berechnung der Abstandsinformation lassen sich so schnell kollisionsfreie Pfade mit Zusatzinformationen über den minimalen Abstand des Roboters zu dessen Umgebung erzeugen. Abbildung 5.9 zeigt die Ergebnisse der einzelnen Planungselemente für die zuvor verwendeten Szenarien zur Evaluierung der verteilten Planung und Optimierung. Es ist zu erkennen, dass insbesondere komplexere Pfade stark von der Parallelisierung profitieren. Das Umfahren eines Hindernisses oder die Bewegung innerhalb eines komplexen Szenarios wie dem Küchenszenario, kann enorm beschleunigt werden. Die zeitaufwändigste Operation ist nun die Planung selbst, da diese bisher nur mit dem ODER-Pradigma parallelisiert wurde. Dennoch kann für das Küchenszenario bereits eine Halbierung der Gesamtrechendauer unter der Verwendung von 4 Clients mit jeweils 2 Agenten erreicht werden. Zusätzlich erhält der Pfad zu jeder Konfiguration die Information über den Abstand des Roboters zur Umgebung, ohne dabei zu viel Rechenzeit zu investieren. Abschließend kann also zusammengefasst werden, dass bei einer Verschränkung von Planung und Ausführung eine Onlinepfadplanung für Labordemonstrationen erreicht werden kann.

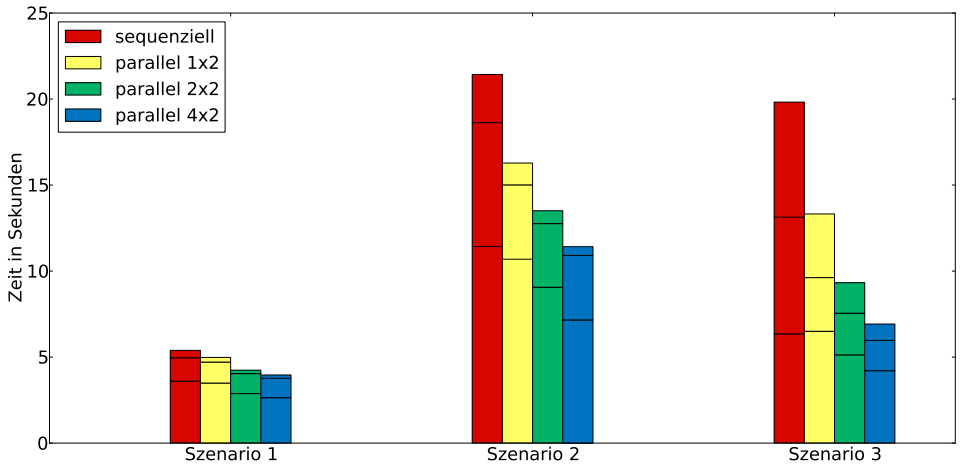


Abbildung 5.9: Gemittelte Zeiten der Experimente zur Integration aller Methoden des Planungszykluses. Die Zeiten sind nach Szenario gruppiert. Der parallele Algorithmus wurde auf einem Client mit 2 Agenten (gelb), auf 2 Clients mit je 2 Agenten (grün) und auf 4 Clients mit je 2 Agenten (blau) aufgeteilt. Die sequentiellen Referenzalgorithmen sind rot dargestellt.

6 Überführung der Softwarearchitektur auf den realen Roboter

Die Integration der bisher nur in der Simulation getesteten Verfahren auf dem realen Roboter ist ein entscheidender Schritt dieser Arbeit. Für Verfahren und Architekturen, die für den Einsatz in einer realen Umgebung entwickelt werden, ist es unabdingbar deren Ergebnisse im realen Betrieb zu testen. Da die Simulation zu einem gewissen Grad von der Realität abweicht, muss untersucht werden, wie gut sich die Verfahren im echten Umfeld des Roboters verhalten. Bei der Überführung auf den realen Roboter soll daher untersucht werden, ob die entwickelte Architektur zur verteilten Pfadplanung und die darin enthaltenen Verfahren auch unter realen Bedingungen einsetzbar sind. Dafür wird im ersten Teil des Kapitels beschrieben, wie die Abstandsinformation genutzt werden kann, um die Ausführung der Trajektorien auf dem echten Roboter sicherer zu machen. Der zweite Teil des Kapitels beschäftigt sich dann mit der tatsächlichen Ausführung auf dem realen Roboter. Dabei werden die Planungszeiten und der Einfluss der Abstandsinformation beobachtet.

6.1 Verwendung der Abstandsinformation

Da die hier verwendeten Planungsverfahren sehr oft Pfade erzeugen, die sehr nahe an Hindernissen vorbeiführen, neigt der reale Roboter zu Kollisionen mit solchen Objekten. Zur Vermeidung dieser, kann man versuchen den Pfad so zu modifizieren, dass diese Engstellen reduziert werden. Da dies sehr rechenaufwändig ist, wird in diesem Abschnitt erläutert, wie Abstandsinformationen verwendet werden können, damit die Ausführung auf dem realen Robotersystem sicherer gestaltet werden kann. Hierbei wird die Rechenzeit zur Trajektoriengenerierung nicht signifikant erhöht, da die Abstandsinformation, wie in Kapitel 5.2 beschrieben, verteilt auf mehreren Computern und Kernen erfolgt.

Die Abstandsinformation wird konkret verwendet, um die Geschwindigkeiten und Beschleunigungen, die bei der Bewegung der Manipulatoren des echten Roboters entstehen, zu limitieren. Ziel der Limitierung ist die Reduzierung von ungewollten Überschwingern der Roboterarme bei zu schnellen Beschleunigungen und Beschleunigungsänderungen. Dies ist vor allem für das egne Umfahren von Hindernissen bedeutend, da hier bereits kleinere Überschwinger zu Kollisionen führen können. Dazu wird der Abstand c einer jeden Konfiguration verwendet, um einen Faktor f zu berechnen, der die maximale Gelenkwinkelgeschwindigkeit des Roboters limitiert. Der Faktor wird abhängig des Abstandes c nach Formel 6.1 berechnet. Dabei

ist c_{max} der maximal zulässige Abstand in Metern zwischen Roboter und Hindernissen, über welchem die Beschleunigungen und Geschwindigkeiten des Roboters nicht reguliert werden. c_{min} ist der Abstand in Metern unterhalb dem die Bewegungen des Roboters auf ein Minimum verlangsamt werden.

$$f = \begin{cases} 1 & \text{if } c > c_{max} \ , \\ \frac{c}{c_{max}} & \text{if } c \leq c_{max} \ , \\ \frac{c_{min}}{c_{max}} & \text{if } c \leq c_{min} \end{cases} \tag{6.1}$$

Die Grenzabstände c_{min} und c_{max} werden empirisch bestimmt. Setzt man c_{min} auf 0,01 m und c_{max} auf 0,10 m, so bewegen sich die Geschwindigkeiten innerhalb der Grenzwerte zwischen 10% und 100%. Der Roboter kommt jedoch nie ungewollt zum Stillstand. Die daraus resultierenden Beschleunigungen und Geschwindigkeiten können mithilfe eines geeigneten Interpolators in Trajektorien mit dem gewünschten Verhalten überführt werden. Abhängig vom Abstand bewegt sich der Roboter schnell im freien Raum, wohingegen er abbremst, sobald er sich Objekten nähert.

Abbildung 6.1 zeigt dieses Verhalten in der Simulationsumgebung. Die Abbildung ist in zwei Unterabbildungen aufgeteilt. Der obere Teil der Abbildung zeigt den Roboter beim Ausführen einer Trajektorie. Die Trajektorie ist entsprechend der Ausführungsgeschwindigkeit eingefärbt. Das blaue Segment weist auf eine schnelle Bewegung hin, die nicht durch einen Sicherheitsabstand abgebremst wird. Innerhalb der roten Segmente der Trajektorie wird die Geschwindigkeit auf ein Minimum herunter gesetzt. Je mehr sich der Manipulator dem Tisch nähert, desto langsamer wird er. Der untere Teil der Abbildung zeigt den berechneten Abstand c (blau) und den davon abhängigen Faktor f (schwarz) zur Limitierung der Geschwindigkeiten und Beschleunigungen für jede Konfiguration. Die Grenze des maximalen Abstandes c_{max} ist als gestrichelte grüne Linie bei 0,10 m eingezeichnet. Der minimale Abstand c_{min} ist bei 0,01 m rot eingezeichnet. Sobald der Abstand unter c_{max} fällt, sinkt auch der Faktor f und limitiert somit die Ausführungsgeschwindigkeit des Manipulators.

Nach diesem Verfahren können auch die Geschwindigkeiten und Beschleunigungen des realen Roboters limitiert werden. Um die Abstandsinformationen der realen Welt zu erhalten, muss jedoch zuvor die Umgebung des Roboters definiert werden. Dazu werden die Objekte in Justins Arbeitsraum mittels dessen eingebauten Kameras lokalisiert. Hierfür wird das kommerzielle Toolkit HALCON von MVTec verwendet [28], mit dem 6D-Posen für bekannte Objekte innerhalb eines 2D Bildausschnittes erkannt werden können. Diese Posen werden anschließend in die Simulation transferiert. Die simulierte Umgebung wird so auf die reale Umgebung abgeglichen. Die gemessenen Abstände der in der Simulation geplanten Pfade stimmen somit, bis auf einen gewissen Lokalisierungsfehler, mit der Realität überein und können dazu verwendet werden, die Geschwindigkeiten und Beschleunigungen zu regulieren.

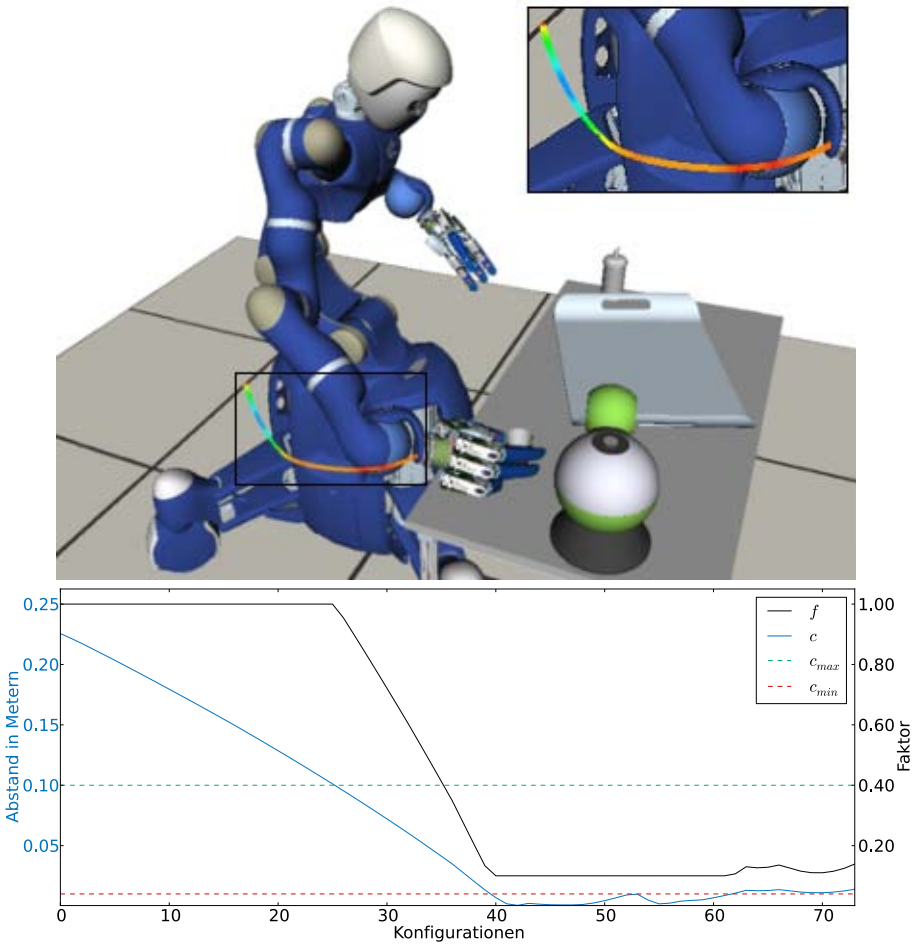


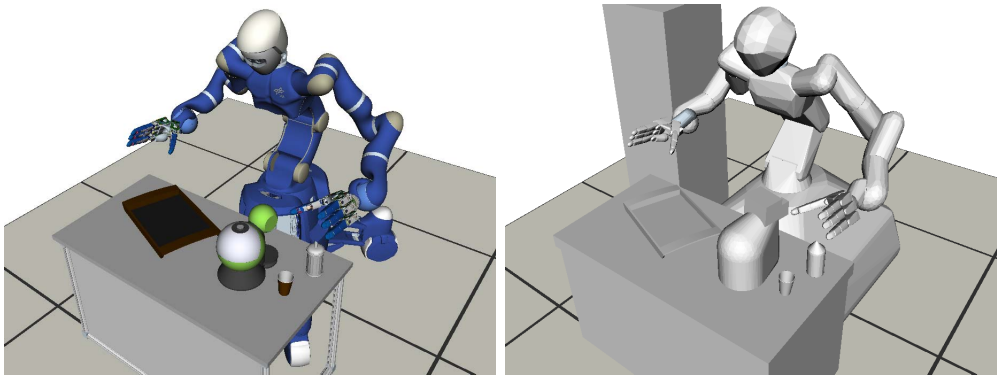
Abbildung 6.1: Geschwindigkeitsprofil für eine Trajektorie des Endeffektors (oben) und die dazugehörigen Abstände für die Annäherung an den Becher (unten). Der Faktor f hat seine Y-Achse auf der rechten Seite und ist in schwarz eingezeichnet. Die berechneten Abstände c haben ihre Skala links und sind als blaue Linie gekennzeichnet.

6.2 Evaluierung der entworfenen Architektur anhand von Rollin' Justin

Zur abschließenden Evaluierung werden alle entwickelten Algorithmen und die verteilte Softwarearchitektur DISPLACE im realen Einsatz mit Rollin' Justin getestet. Dazu wird das Küchenszenario leicht verändert und in die reale Umgebung des Roboters transferiert. Im Gegensatz zu den simulierten Experimenten sind hierbei die Positionen der Objekte sowie die Position des Tisches zu einem gewissen Grad beliebig wählbar.

6.2.1 Versuchsaufbau

Der Aufbau des Experiments weicht etwas von der simulierten Variante des Küchenszenarios ab. Die Kaffeemaschine ist hierbei zwischen das Tablett und die aufzuräumenden Objekte verschoben worden, damit ein größeres Hindernis im Weg steht. Das Tablett steht vom Roboter aus gesehen rechts von der Kaffeemaschine. Der Zucker sowie der Becher stehen links. Die Kaffeemaschine kann so verstellt werden, dass sie wahlweise als direktes Hindernis dient, oder weiter in den Hintergrund verlagert wird. Die Ausgangsstellung des Roboters ist weitestmöglich der Simulation nachempfunden. Abbildung 6.2 zeigt den Aufbau in der Testumgebung in der Simulation. Im rechten Teil der Abbildung ist zu erkennen, dass konvexe Hüllen zur Kollisionsdetektion verwendet wurden. Die Kaffeemaschine ist dabei um die Bedienhebel erweitert. Die graue Box verhindert die Kollision mit einem Infomonitor in der echten Welt.



(a) Renderingmodell (b) Kollisionsmodell

Abbildung 6.2: Simulierte Umgebung des realen Experiments. Der linke Teil zeigt die Anzeige des Renderingmodells, während auf der rechten Seite das Kollisionsmodell angezeigt wird. Das Kollisionsmodell der Kaffeemaschine ist zur Kollisionsdetektion um die Bedienelemente erweitert. Dazu sind kleine Boxen an die relevanten Stellen der Maschine angefügt worden. Die graue Box im Hintergrund dient der Kollisionsvermeidung mit einem Infomonitor in der realen Welt.

6.2.2 Versuchsablauf

Der Ablauf entspricht in den Grundzügen dem Ablauf des simulierten Küchenszenarios. Jedoch wird in diesem Experiment nur der linke Arm zur Manipulation verwendet. Zusätzlich kann das erste Torsogelenk zum Verdrehen des Oberkörpers verwendet werden, sodass sich eine bessere Ausgangsposition zum Abstellen der Objekte ergibt. Die Position der Objekte ist variabel. Somit ist auch der Ablauf variabel. Justin beginnt beim Aufräumen mit dem Objekt, das ihm am nächsten

steht. Befindet sich die Kaffeemaschine weiter hinten, kann der Pfadplanungsalgorithmus durchaus sehr kurze Pfade finden. Ist die Kaffeemaschine näher am Roboter positioniert, findet der Planer tendenziell längere Pfade über, oder um die Kaffeemaschine herum. Zum Aufheben und Absetzen werden definierte Posen über den Objekten angefahren, die zur besseren Annäherung an die Objekte dienen. Von diesen Positionen verfolgt der Manipulator eine kollisionsfreie kartesische Linie, um das Objekt letztendlich zu greifen. Die Griffe sind fest definiert. Nach getaner Arbeit begibt sich Justin wieder auf die Startposition zurück. Abhängig vom Abstand zu den Objekten bewegt sich der Roboter entsprechend schneller oder langsamer. Außerdem wird die Planung kaskadiert. Dies bedeutet, dass während der Roboter einer geplanten Trajektorie folgt, der Planer bereits den nächsten Bewegungsablauf plant. Die Wartezeit zwischen den Bewegungen kann so auf ein Minimum reduziert werden oder sogar ganz entfallen.

Abbildung 6.3: Der reale Roboter beim Aufräumen des Tisches. Rechts ist die Kaffeemaschine nah am Roboter positioniert. Der Becher muss daher über die Maschine hinweg geführt werden. Die Aufräumreihenfolge variiert dabei je nach Position der Objekte.

6.2.3 Ergebnisse

Die Ausführung auf dem realen Roboter hat gezeigt, dass die Planungszeit mit der verteilten Softwarearchitektur so weit reduziert werden kann, dass die Wartezeiten für den Benutzer des Roboters erträglich werden. Durch die Kaskadierung der Bewegungs- und Planungsvorgänge kann die Wartezeit nach der ersten Bewegung sogar komplett vernachlässigt werden. Die Beschleunigungen und Geschwindigkeiten des Roboters werden dabei abhängig des Abstandes zu den Objekten angepasst. Abbildung 6.4 zeigt, wie die Ausführungsgeschwindigkeit gesenkt wird, sobald der Manipulator sich einem Hindernis nähert. Das Hindernis ist symbolisch eingezeichnet.

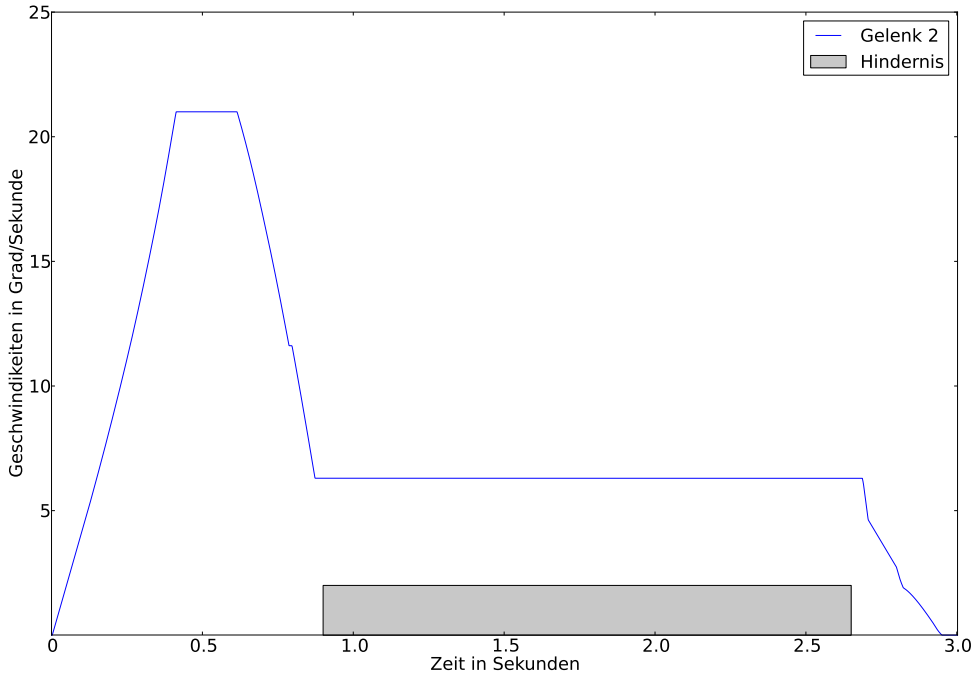


Abbildung 6.4: Gemessene Geschwindigkeiten für das zweite Gelenk des linken Manipulators. Der graue Block symbolisiert das Auftreten eines Objektes in der Nähe des Roboters.

6.3 Auswertung des realen Testszenarios

Das finale Testszenario für den Einsatz auf dem realen Roboter hat gezeigt, dass auch in alltäglichen Situationen schnell kollisionsfreie Pfade für humanoide Serviceroboter erstellt werden können. Dabei leistet die kurze Rechenzeit für die Pfadplanung und -optimierung einen wichtigen Beitrag. Zusätzlich ist die Erweiterung des Pfades um Abstandsinformationen sehr hilfreich, damit Roboter mit Leichtbaustruktur einer Trajektorie ohne ungewollte Kollisionen durch Überschwinger folgen können.

7 Weiterführende Arbeiten und Ausblick

Der Ausblick als letztes Kapitel der Masterarbeit soll einen Überblick über die weiterführenden Arbeiten im Zusammenhang mit der DISPLACE-Architektur verschaffen. Dazu unterteilt sich das Kapitel in die weiterführenden Arbeiten, welche bereits Ansätze zur Weiterentwicklung des Systems beinhalten und den Ausblick, der langfristige Ziele umfasst.

7.1 Weiterführende Arbeiten

Die Integration des DISPLACE-Frameworks in die Laborumgebung des realen Roboters hat gezeigt, dass die Verkürzung der Rechenzeit ein wichtiger Schritt zur autonomen Manipulation für Justin darstellt. Zwar konnten bereits jetzt die Rechenzeiten stark reduziert werden, allerdings steckt noch viel Potential in der eigentlichen Planung. Bisher wurde der Pfadplanungsprozess nur mit Hilfe des ODER-Paradigmas parallelisiert. Dabei konnte ein Zeitgewinn von etwa 30% erreicht werden. Der kooperative Aufbau mehrerer separat entstehender Suchbaume kann die Planungszeit zusätzlich verkürzen. Die DISPLACE-Architektur wurde hinsichtlich dieses Kriteriums entwickelt und bietet daher die optimalen Voraussetzungen für die Integration auch dieses Verfahrens. In theoretischen Überlegungen ist diese Ebene der Parallelisierung bereits angedacht worden. Der nächste konkrete Schritt besteht deswegen darin, die Architektur um diese Funktion zu erweitern.

Auch für die zweiarmige Manipulation bietet sich die verteilte Architektur an. Da die Softwarearchitektur unabhängig von einem Roboter entwickelt wurde, kann die Pfadplanung und -optimierung auch für mehr als nur einen Manipulator verwendet werden. Bei den Versuchen am realen Roboter wurden bereits unterschiedlichste Posen des Roboters mittels der verteilten Pfadplanung angefahren. Dies ist insbesondere für den Laboreinsatz hilfreich, wärenddessen der Roboter durchaus komplexere Posen einnehmen muss, ohne dabei mit der Laboreinrichtung zu kollidieren. Zum Erreichen von weiter entfernten Objekten kann die Bewegung des Torsos effizient mit geplant werden, ohne dabei die Rechenzeit in die Höhe zu treiben.

Als weitere Ebene der Parallelisierung ist zusätzlich angedacht, das Framework zur Aufteilung der Planung auf den linken und den rechten Arm des Roboters zu verwenden [27], welches in einer früheren Arbeit entwickelt wurde um die Planungszeit zu verkürzen. Dazu muss pro Arm des Roboters eine unabhängige Instanz des DISPLACE-Frameworks eingesetzt werden.

7.2 Ausblick

Die Arbeit mit der verteilten Planungsarchitektur und deren Verwendung auf dem realen Roboter haben gezeigt, dass sowohl die in dieser Arbeit entwickelte Softwarearchitektur als auch die Entwicklungsumgebung des Roboters noch einige Erweiterungen benötigen. So kann die DISPLACE-Architektur keine zusätzlichen Objekte handhaben, die nach dem Laden der Umgebung hinzugefügt werden. Dies ist jedoch essentiell für einen humanoiden Roboter, der bei der Exploration eines Raumes ständig auf neue Gegenstände trifft. Außerdem kann bei Verwendung der DISPLACE-Architektur bisher keine Nebenbedingungen bei der Planung eingehalten werden, da dies noch nicht implementiert wurde. Damit Beispielsweise eine Wasserglas aufrecht bewegt werden kann, muss sowohl während des Planungsschrittes als auch während des Optimierungsschrittes darauf geachtet werden, dass nur Konfigurationen berücksichtigt werden, in denen das Wasserglas aufrecht steht. So kann es bei der Bewegung des Zuckerstreuers dazu kommen, dass der Roboter Zucker verschüttet. Die Berücksichtigungen solcher Nebenbedingungen ist nach Stilman et. al [34] bereits in OpenRAVE für die sequentielle Pfadplanung implementiert.

Auf lange Sicht gesehen, soll die in dieser Arbeit realisierte schnelle Pfadplanung als Grundlage für die Aufgabenplanung eingesetzt werden. Bei der Aufgabenplanung muss der Roboter selbstständig entscheiden, in welcher Reihenfolge eine Aufgabe zu lösen ist. Dabei sind zahlreiche Unterabläufe notwendig, um verschiedenste Manipulationen der Umgebung durchzuführen. In Kombination mit einem Griffplaner kann die Pfadplanung dazu verwendet werden, diese Abläufe zu generieren. Pfadplanung ist also ein wichtiger Schritt zur Autonomie eines Roboters und soll daher in den kommenden Jahren weiter ausgebaut werden.

8 Zusammenfassung

In dieser Arbeit wurde gezeigt, dass die verteilte und parallele Softwarearchitektur DISPLACE nicht nur die Pfadplanung, sondern auch den dazugehörigen Optimierungsschritt signifikant beschleunigen kann. Es wurde gezeigt, dass die Kollisionsabfragen den Hauptteil der Berechnungszeit beanspruchen. Die parallele Berechnung der Kollisionsabfragen verringert die Rechendauer daher enorm. Zur Verteilung des Pfadplanungsschrittes wurde das ODER-Paradigma eingesetzt, bei dem eine Suchanfrage mehrfach gestellt wird. Es wurde jedoch auch gezeigt, dass sich prinzipiell auch der RRT-Algorithmus selbst mit der DISPLACE-Architektur parallelisieren lässt. Innerhalb des Optimierungsschrittes wurde das Linear Shortcut Verfahren auf mehrere Rechnern und Kernen mithilfe des DISPLACE-Frameworks parallelisiert.

Zusätzlich wurde die Ausführung der geplanten Pfade auf dem humanoiden Leichtbauroboter Rollin' Justin untersucht. Dabei erzeugen Gravitation und Beschleunigungen entlang der Trajektorie Abweichungen von der geplanten Bahn. Um diese Abweichungen zu minimieren, wurde der Pfad um Abstandsinformationen zwischen dem Roboter und dessen Umgebung erweitert. Die Abstandsinformationen dienen als Grundlage zur Limitierung der Geschwindigkeiten und Beschleunigungen nahe Hindernissen. Die Berechnung der Abstandsinformationen wurde dabei ebenfalls mit der DISPLACE-Architektur vorgenommen.

In Kombination können die Algorithmen dazu verwendet werden, zeitnah sichere Trajektorien für die Ausführung auf einem realen Leichtbauroboter zu erzeugen. Versuche auf dem mobilen humanoiden Roboter Rollin' Justin haben ergeben, dass die Planungszeit durchaus mit der Ausführungszeit mithalten kann und das die um die Abstandsinformation erweiterten Pfade zusätzliche Sicherheiten bei der Ausführung bieten. Zusammengefasst kann Bewegungsplanung nun für den regulären Einsatz des Roboters verwendet werden.

A Anhang

A.1 Versuchsaufbauten der Evaluierungsszenarien

Die hier aufgelisteten OpenRAVE-Umgebungen bilden die Ausgangsstellung der einzelnen Versuchsszenarien und dienen der genaueren Dokumentation des Versuchsaufbaus.

A.1.1 Szenario 1 - leer

```
<Environment>
    <camtrans>2.639794 1.328300 2.049023</camtrans>
    <camrotaxis>0.272653 0.540387 0.796017 133.268397</camrotaxis>

    <Robot name="Justin" file="robots/justin.robot.xml">
        <translation>0 0 0.010</translation>
    </Robot>

    <KinBody name="floor" file="data/floor.kinbody.xml">
        <translation>0 0 -0.050</translation>
    </KinBody>
</Environment>
```

A.1.2 Szenario 2 - Pfeiler

```
<Environment>
    <camtrans>2.639794 1.328300 2.049023</camtrans>
    <camrotaxis>0.272653 0.540387 0.796017 133.268397</camrotaxis>

    <Robot name="Justin" file="robots/justin.robot.xml">
        <translation>0 0 0.010</translation>
    </Robot>

    <KinBody name="floor" file="data/floor.kinbody.xml">
        <translation>0 0 -0.050</translation>
    </KinBody>

    <KinBody name="tisch" file="data/tisch.kinbody.xml">
        <translation>1.080 -0.600 0</translation>
    </KinBody>

    <KinBody name="pole" file="data/pole.kinbody.xml">
        <translation>0.7 -0.32 0</translation>
    </KinBody>
</Environment>
```

A.1.3 Szenario 3 - Küche

```
<Environment>
    <Robot name="Justin" file="robots/justin.robot.xml">
        <translation>0 0 0.010</translation>
    </Robot>

    <KinBody name="floor" file="data/floor.kinbody.xml">
        <translation>0 0 -0.050</translation>
    </KinBody>

    <KinBody name="tisch" file="data/tisch.kinbody.xml">
        <translation>1.080 -0.600 0</translation>
    </KinBody>

    <KinBody name="tablett" file="data/tablett.kinbody.xml">
        <translation>0.788 0.184 0.76608</translation>
        <RotationAxis>0 0 1 112</RotationAxis>
    </KinBody>

    <KinBody name="kaffee" file="data/kaffeemaschine.kinbody.xml">
        <translation>0.850 -0.520 0.76608</translation>
        <RotationAxis>0 0 1 13</RotationAxis>
    </KinBody>

    <KinBody name="becher" file="data/becher.kinbody.xml">
        <translation>0.611 -0.364 0.76608</translation>
    </KinBody>

    <KinBody name="zuckerdose" file="data/zuckerdose.kinbody.xml">
        <translation>0.607 0.573 0.76608</translation>
    </KinBody>
</Environment>
```

A.1.4 Szenario 4 - Abstandsüberprüfung

```
<Environment>
    <Robot name="Justin" file="robots/justin.robot.xml">
        <translation>0 0 0.010</translation>
    </Robot>

    <KinBody name="floor" file="data/floor.kinbody.xml">
        <translation>0 0 -0.050</translation>
    </KinBody>

    <KinBody name="tisch" file="data/tisch.kinbody.xml">
        <translation>1.080 -0.600 0</translation>
    </KinBody>

    <KinBody name="kaffee" file="data/kaffeemaschine.kinbody.xml">
        <translation>0.650 -0.400 0.76608</translation>
        <RotationAxis>0 0 1 90</RotationAxis>
    </KinBody>
</Environment>
```

Literaturverzeichnis

[1] Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *Spring Joint Computer Conference (SJCC)*, 1967

[2] Bal, H.E.: *Programming distributed systems*. Silicon Pr, 1990

[3] Balzert, H.: *Lehrbuch der Software-Technik 1, 2*. 1998

[4] Berenson, D. ; Srinivasa, S.S. ; Ferguson, D. ; Kuffner, J.J.: Manipulation planning on constraint manifolds. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2009

[5] Bohlin, R. ; Kavraki, L.E.: Path planning using lazy PRM. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2000

[6] Borst, C. ; Ott, C. ; Wimbock, T. ; Brunner, B. ; Zacharias, F. ; Bauml, B. ; Hillenbrand, U. ; Haddadin, S. ; Albu-Schaffer, A. ; Hirzinger, G.: A humanoid upper body system for two-handed manipulation. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2007

[7] Butterfass, J. ; Grebenstein, M. ; Liu, H. ; Hirzinger, G.: DLR-Hand II: Next generation of a dextrous robot hand. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2001

[8] Carpin, S. ; Pagello, E.: On parallel RRTs for multi-robot systems. In: *8th Conference of the Italian Association for Artificial Intelligence (AI*IA)*, 2002

[9] Caselli, S. ; Reggiani, M.: ERPP: An experience-based randomized path planner. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2000

[10] Chen, X. ; Li, Y.: Smooth path planning of a mobile robot using stochastic particle swarm optimization. In: *IEEE International Conference on Mechatronics and Automation (ICMA)*, 2006

[11] Choset, H.M. ; Hutchinson, S. ; Lynch, K.M. ; Kantor, G. ; Burgard, W. ; Kavraki, L.E. ; Thrun, S.: *Principles of robot motion: theory, algorithms, and implementation*. The MIT Press, 2005

[12] Colorni, A. ; Dorigo, M. ; Maniezzo, V. u. a.: Distributed optimization by ant colonies. In: *European Conference on Artificial Life (ECAL)*, 1992

[13] Diankov, R. ; Ratliff, N. ; Ferguson, D. ; Srinivasa, S. ; Kuffner, J.: Bispace planning: Concurrent multi-space exploration. In: *Proceedings of Robotics: Science and Systems IV* (2008)

[14] Diankov, Rosen: *Automated Construction of Robotic Manipulation Programs*, Carnegie Mellon University, Robotics Institute, Diss., August 2010

[15] Ferré, E. ; Laumond, J.P. ; Arechavaleta, G. ; Estevés, C.: Progresses in assembly path planning. In: *International Conference on Product Lifecycle Management (PLM)*, 2005

[16] Fuchs, M. ; Borst, C. ; Giordano, P.R. ; Baumann, A. ; Kraemer, E. ; Langwald, J. ; Gruber, R. ; Seitz, N. ; Plank, G. ; Kunze, K. u. a.: Rollin'Justin–Design considerations and realization of a mobile platform for a humanoid upper body. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2009

[17] Geraerts, R. ; Overmars, M.H.: Clearance based path optimization for motion planning. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2005

[18] Geraerts, R. ; Overmars, M.H.: Creating high-quality paths for motion planning. In: *The International Journal of Robotics Research* 26 (2007), Nr. 8, S. 845

[19] Hirzinger, G. ; Sporer, N. ; Albu-Schaffer, A. ; Hahnle, M. ; Krenn, R. ; Pascucci, A. ; Schedl, M.: DLR's torque-controlled light weight robot III-are we reaching the technological limits now? In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2002

[20] Ihme, T.: *Steuerung von sechsbeinigen Laufrobotern unter dem Aspekt technischer Anwendungen*, Universität Magdeburg, Diss., 2002

[21] Koren, Y. ; Borenstein, J.: Potential field methods and their inherent limitations for mobile robot navigation. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 1991

[22] Kuffner, J.J. ; LaValle, S.M.: RRT-connect: An efficient approach to single-query path planning. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2000

[23] Larsen, E. ; Gottschalk, S. ; Lin, M.C. ; Manocha, D.: Fast proximity queries with swept sphere volumes. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2000

[24] Latombe, J.C.: *Robot motion planning*. Springer, 1991

[25] LaValle, S.M.: Rapidly-exploring random trees: A new tool for path planning. 1998. – Forschungsbericht

[26] LaValle, S.M.: *Planning algorithms*. Cambridge Univ Pr, 2006

[27] Leidner, D.: *Planung und Ausführung von alltäglichen zweihändigen Manipulationsaufgaben*, Hochschule Mannheim, Diplomarbeit, 2010

[28] MVTec: *HALCON*. http. `http://www.mvtec.com/halcon/`. Version: March 2011

[29] Ott, C. ; Eiberger, O. ; Friedl, W. ; Bäuml, B. ; Hillenbrand, U. ; Borst, C. ; Albu-Schäffer, A. ; Brunner, B. ; Hirschmüller, H. ; Kielhöfer, S. u. a.: A humanoid two-arm system for dexterous manipulation. In: *IEEE/RAS International Conference on Humanoid Robots*, 2006

[30] Paul, R.P.: *Robot manipulators*. MIT Press, 1989

[31] Pfeifer, R. ; Iida, F.: Embodied artificial intelligence: Trends and challenges. In: *Embodied Artificial Intelligence* (2004), S. 629–629

[32] Shi, Y.: Reevaluating amdahl's law and gustafson's law. In: *Computer Sciences Department, Temple University* (1996)

[33] SpiegelOnline: *Roboter: Auf der Cebit gibt es neben miniaturisierter Elektronik auch handfestes zu sehen*. http. `http://www.spiegel.de/netzwelt/gadgets/bild-679421-61714.html`. Version: Februar 2010

[34] Stilman, M.: Task constrained motion planning in robot joint space. In: *IEEE/RSJ International Conference onIntelligent Robots and Systems (IROS)*, 2007

[35] Tannenbaum, A.S. ; Kaashoek, M.F. ; Bal, H.E.: Parallel programming using shared objects and broadcasting. In: *Computer* 25 (2002), Nr. 8, S. 10–19

[36] Vougioukas, S.G.: Optimization of robot paths computed by randomized planners. In: *IEEE International Conference on Robotics and Automation (ICRA)* IEEE, 2005, S. 2148–2153

[37] WillowGarage: *PR2 specs*. http. `http://www.willowgarage.com/pages/pr2/specs`. Version: November 2010

[38] Zacharias, F. ; Borst, C. ; Hirzinger, G.: Bridging the gap between task planning and path planning. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006

[39] Zacharias, F. ; Borst, C. ; Hirzinger, G.: Capturing robot workspace structure: representing robot capabilities. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2007

[40] Zacharias, F. ; Leidner, D. ; Schmidt, F. ; Borst, C. ; Hirzinger, G.: Exploiting Structure in Two-armed Manipulation Tasks for Humanoid Robots. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010

[41] Zacharias, F. ; Schlette, C. ; Schmidt, F.: Making planned paths look more human-like in humanoid robot manipulation planning. In: *IEEE International Conference on Robotics and Automation (ICRA)*, 2011