UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Simulator für GP-evolvierte Laufrobotersteuerungsprogramme

PG 368

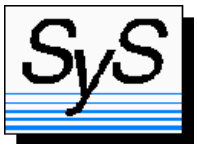
Teilnehmer:
Mihai-Christian Varcol, Holger Türk, Volker Strunk,
Daniel Sawitzki, André Roß, Michael Gregorius,
Abdeladim Benkacem, Christian Aue,
Salah Raiyan Abdallah

Betreuer:
Jens Ziegler, Jens Busch

August 2001

ENDBERICHT

Lehrstuhl für Systemanalyse
Fachbereich Informatik
der Universität Dortmund



Inhaltsverzeichnis

Vorwort **5**

1 Einleitung **7**
1.1 Problemstellung 7
1.2 Zieldefinition 10

2 Grundlagen **11**
2.1 Evolutionäre Algorithmen 11
2.2 Kinematische Modellierung 25
2.3 Geometrische Modellierung 43
2.4 Kollisionserkennung 46
2.5 Dynamik 52
2.6 Numerische Methoden 67

3 Entwicklungsumgebung und Werkzeuge **79**
3.1 Die Programmiersprache C++ 79
3.2 Die Bibliothek DynaMo 83
3.3 Die Bibliothek SOLID 84
3.4 Die Bibliothek DynaMechs 85
3.5 Die Bibliothek Newmat 87
3.6 Die Bibliothek Qt 88
3.7 Das Werkzeug Qt Designer 92
3.8 Die Bibliothek OpenGL 95
3.9 Die Bibliothek cv97 102
3.10 Die Bibliothek PVM 102
3.11 Das Werkzeug Concurrent Version System 107
3.12 Das Werkzeug Make 109

4 Das System SIGEL **113**
4.1 Architektur 113
4.2 Repräsentation und Einlesen von Robotermodellen 115
4.3 Aufbau der Robotersteuerungsprogramme 128
4.4 Der Dynamiksimulator 133
4.5 Das GP-System 137

4.6 Bewertung von Individuen und Fitnessfunktion 148

5 Ergebnisse **151**
5.1 Funktionalität mit hoher Crossoverrate 151
5.2 Evolution mit hoher Mutationsrate 155
5.3 Evolution mit importierten Individuen 158
5.4 Evolutionen mit reduzierten Roboterbefehlssätzen 160
5.5 Evolution eines raupenähnlichen Roboters 165
5.6 Verkürzter raupenähnlicher Roboter 169
5.7 Dreibeiniger Roboter mit SimpleFitness 173
5.8 Dreibeiner mit NiceWalkingFitness 176
5.9 Evolution eines Sechsbeiners 179
5.10 Achsensymmetrischer Sechsbeiner mit NiceWalking 183
5.11 Zusammenfassung 186

6 Der Umgang mit dem System SIGEL **187**
6.1 Inbetriebnahme 187
6.2 Ein erster Überblick über SIGEL 189
6.3 Die Konstruktion von Robotermodellen 197
6.4 Einstellungen und Grenzen des Simulators 212
6.5 Die Evolution und ihre Parameter 215
6.6 Import und Export von GP Daten 220

Nachwort **223**

Literatur **225**

Vorwort

„Alles auf Anfang!“, ertönte es im Raum. Die Luft war zum Schneiden dick und konnte nur so von der im Labor liegenden Spannung ablenken. Ein Fenster zu öffnen würde zwar frische Luft hereinlassen, aber der dabei zwangsläufig entstehende Luftzug würde das Messergebnis verfälschen. „Energieversorgung positiv.“, „Steuerungseinheit positiv.“, „Sensoren positiv.“, nacheinander meldeten die einzelnen Mitarbeiter in den ihnen anvertrauten Bereichen, alle Werte bewegten sich innerhalb der Toleranzen.

Der Versuchleiter, geduldig auf das letzte OK wartend, versuchte tief und regelmäßig zu atmen. Als das letzte OK kam, hielt er noch einen Augenblick inne und sprach dann in die angespannt wartenden Gesichter die zwei entscheidenden Worte “Testlauf starten.“ Irgendwo in dem Dschungel von Technik und Technikern, drückte ein Finger einen Knopf. Augenblicklich schossen Energieimpulse durch Metall um integrierte Schaltungen zu steuern, abzufragen oder zu verändern. Alle Augen waren auf einen Kreis in der Mitte des Raumes gerichtet.

In diesem Kreis stand aufrecht einen kleine Figur, nicht viel größer als eine Spielzeugpuppe für Kinder. Doch im Gegensatz zu ihrer zierlichen und leblosen Schwester, war der Rumpf und die Glieder der Figur klobig, mit kraftstrotzenden Servomotoren und höchstbelastbaren Gelenken vollgestopft. Sie erweckte beim Betrachter den Eindruck, dass sie für extreme Belastungen konzipiert worden war. Im Moment stand sie jedoch nur reglos da. Sorgenvolle Blicke wurden unter den Anwesenden ausgetauscht. Würde sich der Roboter diesmal wenigstens bewegen? Einige richteten schon den Blick von der mechanischen Puppe auf ihre Bildschirme, um die übertragenen Daten zu kontrollieren.

Doch der Versuchsleiter hielt seinen Blick starr auf die Raummitte gerichtet. Hatte die Figur gezittert? Er hielt den Atem an. Da, er konnte genau erkennen, wie der Roboter, fast in Zeitlupe, sein rechtes Bein hob und zu einem Schritt nach vorne strecken wollte. Jetzt hatten es alle bemerkt. Ein paar Mitarbeiter hielt es nicht mehr auf ihren Stühlen. Sie fingen an zu jubeln und den Roboter anzufeuern, so als ob seine Bewegung schon ein Grund zum Feiern wäre. Der Versuchsleiter blieb angespannt. Ein Schritt machte noch keinen laufenden Roboter und dieser hier hatte noch nicht mal seinen ersten Schritt zu Ende gemacht. Als das rechte Hüftgelenk seine maximale Auslenkung erreicht hatte, erstarrte die Figur. Sie stand jetzt da wie ein Zinnsoldat, gegossen in die schönste Parademarschpose. Schlagartig wurde es still im Raum, alle waren wieder auf dem Grund der Tatsachen.

„Alles wieder auf Anfang!“, seufzte der Versuchsleiter. Doch wie eine stummer Aufschrei des Roboters gegen die andauernden Versuche, ging ein letzter Ruck durch den Körper der Puppe. „Fangt den Roboter auf!“, schrie noch jemand, doch er schwankte schon, geriet dann ganz aus dem Gleichgewicht und fiel zu Boden. Eine Technikerin wollte ihn noch auffangen, griff aber vorbei. Jetzt kniete sie in der Mitte des Raumes und hielt den Roboter in zwei Teilen in ihren Händen. Durch den Aufprall war das rechte Bein abgebrochen.

Kapitel 1

Einleitung

In diesem Kapitel soll dem Leser eine kurze Einführung in die Aufgabenstellung der Projektgruppe gegeben werden. In den nächsten Abschnitten werden die Problemstellung und die Zieldefinition näher erläutert.

1.1 Problemstellung

Aufgabe der PG368-SIGEL ist die Realisierung eines Simulators und Evaluators für beliebige Laufroboter-Architekturen, deren Steuerungen mittels genetischer Programmierung evolviert werden sollen. Der Simulator soll hierbei die grundlegenden physikalischen Gegebenheiten der natürlichen Umwelt möglichst genau nachbilden.

Die Zielsetzung ist ferner, dass nicht nur Laufrobotersteuerungsprogramme für eine festgelegte Roboterarchitektur entwickelt werden sollen, sondern auch für ganz beliebige Architekturen. Ist aber ein solches System entwickelt, das in der Lage ist, Laufrobotersteuerungsprogramme zu erzeugen, so kann es auch dafür eingesetzt werden, auf Änderungen der Architektur während der Betriebsphase einer realen Robotersteuerung (z.B. auf ein unbeweglich gewordenes Gelenk) mit einer Anpassung des Steuerungsprogrammes zu reagieren. Auf der Grundlage der veränderten Roboterarchitektur kann ein Kontrollprogramm durch Evolution verbessert werden, so dass es in der Lage ist, den Roboter weiter zu bewegen. Diese Eigenschaft des Systems sorgt für zusätzliche Robustheit. Die Ausführung der evolvierten Robotersteuerungen soll in einem Simulator erfolgen, der die Roboterkinematik und -dynamik in ein Umweltmodell integriert.

Die im Projekt benutzte Genetische Programmierung ist ein Konzept der automatischen Programmierung, welches sich an der Entwicklungsstrategie der natürlichen Evolution orientiert. Grundlegend wird dabei anhand einer Beschreibung der Problemlösung der Lösungsraum nach einer guten Lösung durchsucht. Dabei wird eine Menge von zufällig erzeugten Lösungen durch Anwenden von genetischen Operationen hin zu guten Lösungen verändert. Die Lösungsbeschreibung

gibt dabei an, bei welchen Lösungen durch genentische Veränderung eine Verbesserung versucht wird. Wenn man Genetische Programmierung (Abkürzung: GP) als evolutionäre Strategie zum Lösen von Problemen durch Auslese *und* genetische Veränderung ansieht, dann ist die Simulation der Roboter das Versuchsfeld, in der das GP-System seine Problemlösungen testen kann.

Das Testen von Robotersteuerungsprogrammen in einer simulierten Umwelt hat aber noch weitere Vorteile.

- Das Unfallrisiko, dem ein realer Roboter beim Test ausgesetzt wird, ist so auf ein Minimum reduziert.
- Ein simulierter Testlauf bedarf weniger Aufwand als ein Testlauf mit einem realen Roboter.
- Erst durch den Simulator werden die Voraussetzungen für den Einsatz der Genetischen Programmierung geschaffen.

Alle Ergebnisse sind allerdings mit Vorsicht zu genießen, da es sich nur um Daten aus simulierten Modellen handelt. So mögen die Daten lediglich als Referenz für einen späterern Realtest benutzt werden.

Wie brauchbar ein Ergebnis einer Simulation ist, hängt von der Qualität des Simulators ab. Ist eine physikalische Tatsache in einem Simulator unzureichend berücksichtigt, so kann das Ergebnis einer Simulation nur dann eine brauchbare Aussage über das reale Verhalten des Gegenstandes der Simulation machen, wenn die unberücksichtigte Tatsache keine Auswirkung auf das Verhalten des Gegenstandes in der Realität hat. Das wirft natürlich die Frage nach dem Sinn und Unsinn von Simulationen auf. Eine Simulation kann nie alle Aspekte der Realität simulieren, sondern nur einen ausgewählten Teil. Damit eine Simulation trotzdem Ergebnisse liefern kann, die auf die Realität angewendet werden können, müssen die unwichtigen Faktoren aus der Menge aller Faktoren herausgenommen werden. Die verbleibenden Faktoren müssen richtig, d.h. realitätsgetreu, in der Simulation umgesetzt werden. Nur wenn dies erfüllt ist, kann die Simulation der Roboter mit den genetisch programmierten Laufrobotersteuerungsprogrammen Daten liefern, die das GP-System zu einer Lösung der ihm gestellten Aufgabe benutzen kann.

Die Beschreibung der Roboterarchitektur ist dabei ein grundlegender Faktor, der sowohl Quelle von Diskrepanzen zur Realität ist, welche wiederum den Roboter später real behindern können, als auch Ansatzpunkt nötiger Abstraktionen, ohne die kein funktionierender Simulator auskommen kann. Die Forderung, einen Simulator für verschiedene Roboterarchitekturen zu erstellen, macht eine Vergröberung der realen Roboterarchitektur unumgänglich. Jede Verfeinerung der Architektur muss sich in einer Verfeinerung der Roboterbeschreibungssprache wiederfinden. Wäre der Gegenstand der Untersuchung nur eine Roboterarchitektur, so wäre der Aufwand wesentlich geringer, eine praktikable Roboterbeschreibungssprache zu finden.

Die hohe Flexibilität des Simulators wirkt sich natürlich auch auf die Visualisierung der Simulation aus. Durch sie wird ein Eindruck von der Qualität der

Simulation vermittelt. Die dreidimensionale Visualisierung ist zwar für die Funktion der Genetischen Programmierung unnötig, aber für den Anwender essentiell. Ohne sie könnte der Benutzer sich graphisch nicht vom Fortschreiten der Evolution überzeugen, geschweige denn Tendenzen in den Lösungsansätzen erkennen und darauf reagieren. Verschiedene Ausprägungen in der Entwicklung eines Roboterkontrollprogramms können durch die Visualisierung erkannt und gezielt zu anderen Programmpopulationen hinzugefügt werden, um einen Entwicklungssprung zu erzeugen. Auch *schlechte* gute Lösungen, die zwar der Lösungsdefinition entsprechen, aber in offen gelassenen Punkten ein unerwünschtes Verhalten aufweisen, können nur mit der 3D-Visualisierung erkannt werden, bevor sie auf einem realen Roboter getestet werden.

1.1.1 Anwendungsbereich

Der Anwendungsbereich des Programms liegt eher im wissenschaftlichen Bereich, aber auch im semiprofessionellen Bereich kann das Programm eingeschränkt eingesetzt werden, denn es kann überall dort angewandt werden, wo Roboter zum (oder wieder zum) Laufen gebracht werden sollen. Allerdings sollte ein prinzipielles Verständnis der im Simulator benutzten Bibliotheken hinsichtlich ihrer einstellbaren Parameter mitgebracht werden. Diese Parameter sind essenziell bedeutend für die Benutzung des Programmsystems.

1.1.2 Zielgruppe

Da das Programm eher für den wissenschaftlichen Bereich gedacht ist, muss die Zielgruppe ebenfalls über einen wissenschaftlichen Hintergrund verfügen. So wird zum Beispiel ein minimales Grundwissen über Robotik vorausgesetzt. Dazu gehört beispielsweise das Wissen darüber, was für Gelenkarten es gibt. Auch wenn die Eingabe des Roboters in das Programm auf komfortable Weise erfolgen soll, kommt man ohne Vorwissen über den konkreten Roboter nicht sehr weit. Ferner sollte man ein gewisses Repertoire an Grundprinzipien der Mechanik haben. Begriffe wie Drehmoment, Drall, Kinematische Kette, Trägheitsmomente, etc sollten den Anwender nicht abschrecken.

1.1.3 Plattform

Das Programmsystem in seiner aktuellsten Version läuft unter Unix auf Solarissystemen von der Firma SUN. Außerdem lässt sich das Projekt unter Linux kompilieren und wurde unter einigen Versionen getestet. Es wird wohl letztendlich auf allen Unixsystemen und Unixderivaten laufen können.

1.2 Zieldefinition

Es soll die Realisierung eines Simulators und Evaluators für beliebige Laufroboter-Architekturen implementiert werden. Die individuelle Programmsteuerung des Roboters soll mittels Genetischer Programmierung evolviert werden. Ein Simulator soll hierbei die grundlegenden physikalischen Gegebenheiten der natürlichen Umwelt und das physikalische Verhalten der Roboterarchitektur in der modellierten Umwelt nachbilden.

Da für beliebige Roboterarchitekturen Laufrobotersteuerungsprogramme entwickelt werden sollen, muss beim Entwurf der Steuerung auf ein Modell der kinematischen Verhältnisse des Roboters verzichtet werden.

Ein weiteres Ziel ist eine Visualisierung der evolvierten Programme in einer geeigneten 3D-Darstellung, um so dem mit dem Programm arbeitenden Forscher die Möglichkeit zu geben, den errechneten Fitnesswert nachzuvollziehen.

1.2.1 Pflichtkriterien

Es sollen verschiedene Robotermodelle mit Hilfe der GP (Genetische Programmierung) zum Laufen gebracht werden. Das heißt in der Minimalforderung, die zurückgelegte Strecke des Roboters sollte im Laufe der Evolution zunehmen.

1.2.2 Wunschkriterien

Nach erfüllter Minimalforderung sollen an konkreten realen Architekturen Roboterprogramme in der konkreten Roboterprogrammiersprache evolviert werden, welche das nun reale Objekt vorwärts bewegen sollen, das heißt der Transfer vom Modell zur Realität findet statt. Es wäre gezeigt, dass das System Sigel für die Realität brauchbare Ergebnisse errechnet hat.

Kapitel 2

Grundlagen

In diesem Kapitel geben wir einen groben Überblick über die fachlichen Gebiete, die durch das Projekt *SIGEL* gestreift werden.

Abschnitt 2.1 beschreibt Evolutionäre Algorithmen. Aus diesem mittlerweile sehr weitläufigen Gebiet benutzen wir die Genetische Programmierung, um Steuerungsprogramme für Roboter automatisch entwickeln zu können.

Die kinematische und geometrische Modellierung von Robotern wird in den Abschnitten 2.2 und 2.3 beschrieben.

Die Kombination von Robotermodell und (generiertem) Steuerungsprogramm wird in einem Dynamiksimulator, beschrieben im Abschnitt 2.5, zum Leben erweckt. Innerhalb dieses Simulators kommen eine Reihe von numerischen Methoden, insbesondere zur Integration zur Geltung, beschrieben in Abschnitt 2.6. Teil des Simulators ist auch eine Kollisionserkennung, wie sie in Abschnitt 2.4 beispielhaft beschrieben ist.

2.1 Evolutionäre Algorithmen

Die algorithmische Nachbildung der biologischen Evolution auf Computern hat zu sehr robusten, direkten Optimierungsverfahren geführt ([1], [2], und [3]). Diese Algorithmen werden allgemein *Evolutionäre Algorithmen* (kurz *EA*) genannt und beschreiben einen kollektiven Lernprozess innerhalb einer Population von Individuen. Jedes Individuum repräsentiert dabei einen Punkt im Suchraum, der durch das untersuchte Optimierproblem festgelegt ist. In nahezu drei Jahrzehnten der Forschung und Anwendung haben sich mehrere Klassen Evolutionärer Algorithmen entwickelt, welche sich hauptsächlich in der Repräsentation der Individuen unterscheiden. Da das Verständnis dieser spezialisierten Klassen das Verständnis der allgemeinen Klasse Evolutionärer Algorithmen voraussetzt, wird in den folgenden Abschnitten die allgemeine Klasse der Evolutionären Algorithmen beschrieben, um im Folgenden die spezialisierten Klassen, darunter auch die *Genetische Programmierung*, eingehender zu betrachten (vgl. auch [4], [5], [6], [7] und [8]).

Grundlegende Funktionsweise von EA

Die Funktionsweise Evolutionärer Algorithmen ist im Allgemeinen von den Prozessen der natürlichen Evolution abgeleitet. Die erste Population von Individuen wird zufällig initialisiert. Diese Population unterliegt im Verlauf der Evolution einem fortwährenden Veränderungsprozess. Durch die Anwendung von teilweise probabilistischen *genetischen Selektions-* und *Variationsmechanismen* evolviert sie so, dass die durchschnittliche *Qualität* der Individuen im Verlauf der Evolution zunimmt. Die Berechnung der Qualität, auch *Fitness* genannt, ist abhängig vom zugrundeliegenden Optimierproblem.
Bei der Fitness von Individuen handelt es sich im Normalfall um einen reellen Wert, welcher die Qualität eines Individuums im Kontext eines Optimierproblems repräsentiert. Durch den *Selektionsoperator* werden Individuen höherer Fitness bevorzugt (und ggf. variiert) in die nächste Generation übernommen, wodurch der evolutionäre Prozess eine Richtung erhält. Die Variation von Individuen erfolgt durch genetische Operatoren wie *Mutation* oder *Rekombination*. Der Rekombinationsoperator ermöglicht den Austausch elterlicher Informationen bei der Erzeugung von Nachkommen, während die Mutation genetische Innovationen durch punktuelle Variationen innerhalb des Individuums erzeugt. Ob sich die Veränderungen der Individuen vor- oder nachteilhaft auswirken, wird im Folgenden durch den Selektionsoperator entschieden, indem bessere Individuen selektiert und schlechtere aus der Population entfernt werden.

Grundlegender Evolutionärer Algorithmus

Der grundlegende evolutionäre Algorithmus (s. Abb. 2.1) ist sehr einfach. Im folgenden Algorithmus (nach [1]) ist *t* ein Generationszähler. Die Population *P(t)* bezeichnet bezeichnet eine Population, welche zum diskreten Zeitpunkt t aus μ Individuen besteht. *P'(t)* und *P"(t)* sind intermediär gebildete Zwischenstationen bis zur Population P(t+1). Die Individuen, welche durch die Funktionen *rekombiniere* und *mutiere* rekombiniert bzw. mutiert worden sind, sind in der Menge *P"(t)* enthalten und werden durch die Funktion *bewerte* bewertet. Die Selektion wird schließlich durch die Funktion *selektiere* durchgeführt, wodurch die besseren Individuen in die nächste Generation P(t+1) übernommen werden.

Bei diesem Algorithmus handelt es sich lediglich um den Basisalgorithmus, welcher abhängig von der Art des EA variieren kann. Die drei wesentlichen Vertreter dieser Variationen seien in den folgenden Kapiteln nun näher beschrieben:

- Genetische Algorithmen
- Evolutionsstrategien
- Genetische Programmierung

Zu den einzelnen EA Variationen existieren unterschiedliche theoretische Betrachtungen, welche hier zwar kurz vorgestellt, aber leider nicht tiefergehend be-

$t := 0$;
initialisiere $P(t)$;
bewerte $P(t)$;
while not terminiere $P(t)$ **do**
  $P'(t)$:=rekombiniere $P(t)$;
  $P''(t)$:=mutiere $P'(t)$;
  bewerte $P''(t)$;
  $P(t+1)$:=selektiere $P''(t)$;
  $t := t+1$;
end while

Abbildung 2.1: Grundlegender Evolutionärer Algorithmus, zur Erläuterung siehe Text

handelt werden können. Für detaillierte theoretische Betrachtungen sei auf weiterführende Literatur wie z.B. [1], [2], [3] und [9] verwiesen.

2.1.1 Genetische Algorithmen

Genetische Algorithmen (kurz GA) verwenden in ihrer Standardform Individuen, welche als binäre Zeichenketten $\mathbf{a} \in \{0,1\}^l$ repräsentiert werden, wobei l konstant bleibt. Die Fitnessfunktion $f_{GA} : I \longrightarrow R$ evaluiert die Qualität des Individuums I und repräsentiert diese als reellen Wert R. Da alle Individuen als binäre Zeichenketten repräsentiert werden, liegt der Schluss nahe, dass alle mit GA betrachteten Probleme auf pseudo-Bool'sche Probleme beschränkt seien. Das dies nicht so ist, wird ausführlich in [3] beschrieben, indem eine Methode vorgestellt wird, mit der GA auch auf kontinuierliche Optimierprobleme angewendet werden können.

Wenn alle Individuen einer Population bewertet worden sind, wählt der Selektionoperator $s : I^\lambda \longrightarrow I^\lambda$ λ Individuen aus der alten Population aus. Im elementaren Genetischen Algorithmus wird dazu eine fitness-proportionale Selektion verwendet. Die Auswahlwahrscheinlichkeit p_s eines jeden Individuums a_i ist dabei gegeben durch die *relative Fitness*:

$$p_i(a_i) = \frac{f(a_i)}{\sum_{j=1}^{\lambda} f(a_j)} \tag{2.1}$$

Mit der Wahl dieser Fitnessmethode wird genau den Individuen eine größere Auswahlwahrscheinlichkeit eingeräumt, deren Fitness über der des Durchschnitts liegt. Allgemein wird diese Methode auch *Rouletterad-Selektion* genannt, da man sich leicht vorstellen kann, dass ein besseres Individuum mehr Plätze (proportional zur Fitness) auf einem Rouletterad zugewiesen bekommt als ein schlechteres, wodurch die Wahrscheinlichkeit einer Auswahl natürlich steigt.

Nach der Selektion alleine stiege der Anteil der besseren Individuen in einer neuen Population. Aber es sind noch keine neuen Suchpunkte im Raum aufgesucht

worden. Wie in 2.1 bereits erwähnt, werden Mutations- und Rekombinationstechniken zu diesem Zweck eingesetzt. Die so veränderten, selektierten Individuen bilden die neue Generation. Traditionell ist die *Rekombination* im Rahmen der GA der dominante genetische Operator, indem $\omega_c : I \times I \longrightarrow I \times I$ (im Allgemeinen entstehen zwei Kinder) die genetischen Informationen der Eltern verwendet, um ein neues Individuum I zu erzeugen. Es gibt sehr viele Rekombinationsvarianten. Die wahrscheinlich einfachste Variante wird durch die *Ein-Punkt-Rekombination* realisiert. Bei dieser Variante wird ein *Kreuzungspunkt* zufällig gewählt, wonach die elterlichen Informationen kreuzweise ausgetauscht und auf das neue Individuum übertragen werden. Entsprechend gibt es auch *n-Punkt-Rekombinationen*, bei denen *mehrere* Kreuzungspunkte gewählt werden. [3] stellt diese Varianten weiter vor. Die *Mutation* wird durch die Inversion einzelner Bits realisiert, wodurch $\omega_m : I \longrightarrow I$ ein Individuum nur in einem Bit verändert. Die Mutation wird im Rahmen der GA eher als *Hintergrundoperator* gesehen, weshalb die Mutation in den meisten Realisierungen im Gegensatz zur Rekombination eine eher untergeordnete Rolle spielt.

Das Schema-Theorem

Eine der grundlegenden, aber auch sehr umstrittenen Theorien im Rahmen der Genetischen Algorithmen ist das so genannte *Schema-Theorem*. Das Schema-Theorem stützt sich auf das Konzept der Schemata. Ein *Schema* $H \in \{0,1,*\}^l$ ist prinzipiell die Definition eines Unterraumes im gesamten Suchraum. *Instanzen* eines Schemas H sind all jene Zeichenketten $\mathbf{a} \in \{0,1\}^l$, welche in allen Positionen der Einsen und Nullen genau mit H übereinstimmen. Dabei können genau die Positionen in **a** beliebig besetzt sein, in denen H das *don't care* Symbol * trägt (Beispiel: H=$\{$1001*1101$\}$)Je nach Beschaffenheit der Population, kann zu jedem Schema H^t die Anzahl der Instanzen $m(H^t)$ berechnet werden. Weiterhin beschreibt $\sigma(H)$ die *Ordnung* des Schemas H, also die Anzahl fester Positionen in H, während $\delta(H)$ die *definierende Länge* beschreibt, also den Abstand zwischen der ersten und der letzten festen Position von H. Die *durchschnittliche Schema Fitness* des Schemas H^t in der Population $P(t)$ wird berechnet durch

$$f(H^t) = \frac{1}{m(H^t)} \cdot \sum_{a_i \in H^t} f(a_j) \tag{2.2}$$

Sei $\bar{f}^t$ die durchschnittliche Fitness von $P(t)$. So folgt im Fall der verwendeten proportionalen Selektion (s. auch Gleichung 2.1) für die Anzahl der Instanzen eines Schemas in der nachfolgenden Generation $t+1$

$$m(H^{t+1}) = m(H^t) \cdot \frac{f(H^t)}{\bar{f}^t} \tag{2.3}$$

Unter der Annahme, dass die durchschnittliche Schema Fitness von H^t über dem Durchschnitt der Population liegt, z.B. $f(H^t) = \bar{f}^t + c\bar{f}^t$ mit einer Konstanten c > 0, kann nach t Zeitschritten (startend bei $t_0 = 0$) folglich angenommen werden, dass gilt

$$m(H^t) = m(H^0) \cdot (1+c)^t \tag{2.4}$$

Zusammenfassend führt dies zur Aussage, dass die proportionale Selektion überdurchschnittliche Individuen in einer exponentiell ansteigenden Anzahl berücksichtigt, während vice versa unterdurchschnittliche Individuen exponentiell entsprechend weniger berücksichtigt werden. Die bisherige Analyse hat die genetischen Operatoren jedoch noch nicht berücksichtigt. Durch die Berechnung der *Überlebenswahrscheinlichkeit* eines Schemas H^t kann dies jedoch nachgeholt werden. Sei p_c die Rekombinationswahrscheinlichkeit und p_m die Mutationswahrscheinlichkeit. Die Annahme der Überlebenswahrscheinlichkeit für eine Schema H^t beim enfachen Crossover (als Rekombinationsvariante) mit $1 - p_c \cdot \delta(H^t)/(l-1)$ und bei der Mutation mit $(1-p_m)^{\sigma(H^t)}$ führt direkt zum *Schema-Theorem*:

$$m(H^{t+1}) \geq m(H^t) \cdot \frac{f(H^t)}{\bar{f}^t} \cdot \left(1 - p_c \frac{\delta(H^t)}{l-1}\right) \cdot (1-p_m)^{\sigma(H^t)} \tag{2.5}$$

Das Schema-Theorem besagt, dass kurze überdurchschnittliche Schemata geringer Ordnung, die so genannten *Building Blocks*, im Laufe der Generationen mit hoher Wahrscheinlichkeit eine exponentielle Verbreitung finden werden. Somit gehört die Kombinationen von Building Blocks zu stets besseren Zeichenketten zu den wichtigsten Arbeitsprinzipien der Genetischen Algorithmen.

2.1.2 Evolutionsstrategien

Evolutionsstrategien (kurz ES) benutzen zur Repräsentation der Individuen im Gegensatz zu den Genetischen Algorithmen reellwertige Vektoren $\mathbf{x} \in R^n$. Eine Population besteht aus μ *Eltern*, auf deren Basis λ *Nachkommen* durch genetische Operatoren erzeugt werden. Die Fitnessfunktion $f_{ES} : I \longrightarrow R$ evaluiert die Qualität des Individuums I und repräsentiert diese ebenfalls als reellen Wert. Im Gegensatz zu den Genetischen Algorithmen ist bei den Evolutionsstrategien die *Mutation* der dominante genetische Operator. Bei der Mutation handelt es sich um additiv normalverteilte Modifikationen, wobei die gewählte Standardabweichung ebenfalls dem Evolutionsprozess unterliegen kann und somit *selbstadaptiv* erlernt werden kann. Ebenfalls verwenden Evolutionsstrategien auch die *Rekombination*, bei der, ähnlich wie bei den GA, elterliche Informationen kreuzweise ausgetauscht werden. Auch im Rahmen der ES gibt es von der *1-Punkt-Rekombination* bis zur *n-Punkt-Rekombination* mehrere Rekombinationsvarianten. Der wohl größte Unterschied liegt jedoch in der Selektionsmethode . Im Allgemeinen werden bei Evolutionsstrategien zwei Selektionsverfahren unterschieden:

1. (μ,λ) – *Selektion*
   Bei dieser Selektionsmethode werden deterministisch die besten μ Individuen aus aus den λ Nachkommen als Eltern für die nächste Generation bestimmt. Offensichtlich muss im Fall dieser Selektionsmethode $\lambda \geq \mu$ gelten.

2. $(\mu+\lambda)$ – *Selektion*
   Bei dieser Selektionsmethode werden deterministisch die besten μ Individuen aus aus den λ Nachkommen *und* den μ Eltern (der letzten Generation) als Eltern für die nächste Generation bestimmt.

Während bei der $(\mu+\lambda)$ - Selektion keine Verschlechterung der aktuell besten Lösung im Laufe der Evolution möglich ist, da die neuen Eltern schließlich aus der Vereinigungsmenge der μ Eltern und der λ Nachkommen bestimmt werden, ist eine Verschlechterung der aktuell besten Lösung bei der (μ,λ) - Selektion durchaus möglich, da bei dieser Variante die Eltern der letzten Generation bei der Selektion nicht mehr berücksichtigt werden. Auf den ersten Blick mag die (μ,λ) - Selektion nachteilhaft erscheinen, schließlich wird eine Verschlechterung der Lösungen im Laufe der Evolution zugelassen. Auf den zweiten Blick ist jedoch ein Vorteil gegenüber der $(\mu+\lambda)$ - Selektion erkennbar: Bei der $(\mu+\lambda)$ - Selektion ist eine Stagnation der Lösung in einem *lokalen Optimum* möglich. Abhängig vom Optimierproblem (und einigen evolutionären Parametern) kann die Evolution einen Zustand innerhalb der Population erreichen, dem sie mit der $(\mu+\lambda)$ - Selektion nicht mehr entrinnen kann, da stets die zwar *aktuell* besseren Individuen gewählt werden, für ein Fortkommen jedoch schlechtere Individuen notwendig wären. Bei der (μ,λ) - Selektion hingegen ist dieser notwendige kurzzeitige evolutionäre Rückschritt möglich, und das Verfahren kann somit nicht in einem lokalem Optimum stagnieren. Die Vorteile dieser Methode sind natürlich nicht nur auf dieses Beispiel beschränkt. Weiterhin sei erwähnt, dass diese Methode ebenfalls Vorteile bezüglich *dynamischer Optima* besitzt. Für Details sei auf [3] verwiesen.

Schrittweitenkontrolle - Die 1/5-Regel

Die Schrittweitenkontrolle der Mutation gibt die Möglichkeit die Mutation hinsichtlich des Erfolges zu beeinflussen ([3]), indem die Sensibilität der Mutation dem Erfolg angepasst wird. Diese Regel ist sehr einfach und lautet: *Die Rate der erfolgreichen Mutationen unter allen Mutationen sollte 1/5 betragen. Wenn die Rate grösser ist, erhöhe, wenn sie kleiner ist, verkleinere die Standardabweichung* σ *[der Mutation].*

Selbstadaption

Wurde ein Individuum bislang nur durch einen n-dimensionalen problemabhängigen Parametervektor

$$(x_1, x_2, \ldots, x_n)$$

repräsentiert, so wird dieser Vektor im Falle der angewandten *Selbstadaption* durch evolutionäre Parameter erweitert, welche nun ebenfalls der Evolution unterliegen und durch Mutation und Rekombination variiert werden:

$$(x_1, x_2, \ldots, x_n, \sigma_1, \sigma_2, \ldots, \sigma_n)$$

Dabei beinhalten $\sigma_1, \ldots, \sigma_n$ die Standardabweichungen, mit denen $x_1, \ldots, x_n$ mutiert werden. Bei der Mutation wird der Parameter x_i einfach zu einer $(0, \sigma_i)$-normalverteilten Zahl addiert. Natürlich sind noch weitere Parameter denkbar. Für Details seien [1], [2] und [3] empfohlen.

2.1.3 Genetische Programmierung

Die *Genetische Programmierung* (kurz *GP*) ist ebenfalls ein Zweig der Evolutionären Algorithmen, speziell der Genetischen Algorithmen, mit dem Ziel der automatischen Programmerzeugung. Ein Computer soll in die Lage versetzt werden, sich mittels GP selbst zu programmieren. Bereits in den 50er Jahren formulierte Arthur Samuel eine Frage, auf welche die Genetische Programmierung die ersten Ansätze einer Beantwortung zu liefern scheint: *"How can computers learn to solve problems without being explicitly programmed? In other words: How can computers be made to do what is needed to be done, without being told exactly how to do it?"*(Arthur Samuel, 1950s)

Im Unterschied zu den Genetischen Algorithmen oder den Evolutionsstrategien repräsentieren die Individuen im Rahmen der GP vollständige *Computerprogramme*. Dabei kann es sich durchaus um C oder C++ Programme handeln, oder um eine eigens definierte Sprache. Im Allgemeinen existiert eine gesamte *Population* von Programmen, welche nach nun teilweise bereits bekannten Methoden evolviert wird.

Definition Genetische Programmierung

Nach [9] kann eine allgemeine Definition des Genetischen Programmierens lauten:

Die Genetische Programmierung ist die direkte Evolution von Programmen oder Algorithmen zum Zweck des induzierten Lernens.

Diese Definition kann noch etwas spezifiziert werden, woraus sich auch recht einfach die Minimalanforderungen an das von uns zu entwickelnde GP-System ableiten lassen:

1. GP repräsentiert ein Problem als die Menge aller möglichen Computerprogramme (oder als eine Untermenge), ggf. mit beschränkter Länge. Die GP Repräsentation ist eine Übermenge über alle möglichen Repräsentationen des Maschinellen Lernens. [1]

[1] Für eine nähere Erläuterung und Betrachtung dieses Begriffs sei auf [9] verwiesen. In [9] wird der GP-Begriff ausführlich im Kontext des Maschinellen Lernens betrachtet.

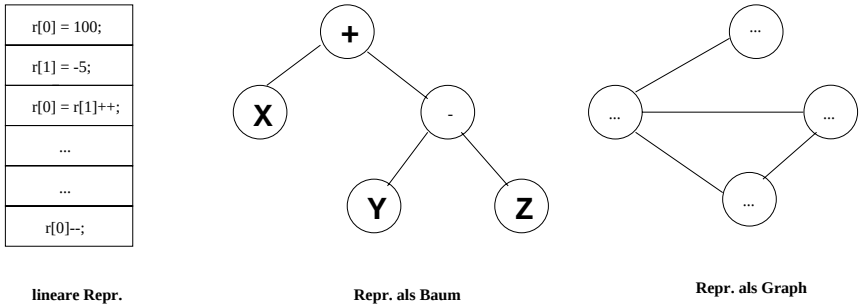


Abbildung 2.2: Repräsentationen

2. GP verwendet Mutation und Crossover als Transformationsoperatoren.

3. GP verwendet *Beamsearch*, ein populationsbasiertes Suchverfahren, welches Populationen von Suchpunkten bildet, bei welchem die Populationsgröße die Größe des Beams darstellt. Die Fitnessfunktion beschreibt die Evolutionsmetrik. Für Details siehe [9].

4. GP ist typischerweise als geleitetes Lernen implementiert.

Repräsentation von Individuen

Im Allgemeinen haben sich drei Repräsentationen (s. Abb. 2.2) für Individuen(programme) etabliert:

1. Baum:

   Im Rahmen dieser Repräsentation werden die Programme als Bäume repräsentiert.

2. Linear:

   Im Rahmen dieser Repräsentation werden die Programme in einem linearen Code repräsentiert. Diese Darstellung ist vergleichbar mit der herkömmlichen (und gewohnten) Darstellung von Programmcode als Programmlisting.

3. Graph:

   Im Rahmen dieser Reprs̈entation werden die Programme als Graphen repräsentiert. Eine Darstellung, welche mit der mathematischen Darstellung von Graphen vergleichbar ist.

Die Fitnessfunktion $f_{GP} : I \longrightarrow R$ evaluiert die Qualität des Individuums, indem das Program, welches durch das aktuelle Individuum repräsentiert wird, auf die aktuelle Problemstellung angewendet wird. Auch bei der GP wird die Fitness durch einen rellen Wert repräsentiert. Bei der Interpretation des Programms wird die Ausführung eines realen Programms (unabhängig von der gewählten Repräsentation) nachempfunden.

Terminal- und Funktionenmenge

Die *Terminal- und Funktionenmenge* legen die Eigenschaften des Suchraums fest. Die Definition dieser Mengen hat einen direkten Einfluss auf die Dimension des Suchraums. Im einzelnen haben die Mengen die folgenden Bedeutungen:

- Terminalmenge

  Die Terminalmenge setzt sich aus den Elementen zusammen, die als *Eingabe* für das GP Programm verstanden werden können. Darunter sind sowohl Konstanten, Variablen als auch Funktionen zu verstehen, welche keine Argumente haben.

- Funktionenmenge

  Die Elemente der Funktionenmenge bestehen aus Operatoren und Funktionen, welche vom GP System unterstützt werden. Dazu können z.B. die folgenden Funktionen gehören:

  - Bool'sche Funktionen:
    AND, OR, NOT, XOR
  - Arithmetische Funktionen:
    ADDITION, SUBTRAKTION, DIVISION, MULTIPLIKATION
  - Variablenzuweisungen:
    ASSIGN
    Kontrollstrukturen:
    IF, THEN, ELSE, CASE, SWITCH

Nach [9] ist es ratsam, die Funktionenmenge nicht zu groß zu wählen, da dies einen direkten Einfluss auf die Größe des Suchraums hat. Im Allgemeinen ist die GP in der Lage, bereits mit wenigen Möglichkeiten sehr gute Lösungen zu evolvieren.

Selektion

Im Rahmen der GP können sowohl die hier bereits erläuterten Selektionsverfahren verwendet werden:

- Fitnessproportionale Selektion, s. 2.1.1,
- $(\mu,\lambda)-$*Selektion*, s. 2.1.2,
- $(\mu+\lambda)-$*Selektion*, s. 2.1.2,

als auch die *Turnierselektion*. In die Turnierselektion werden nicht alle Individuen einer Populationen mit einbezogen. Es wird lediglich nur eine Subpopulation betrachtet. Eine Anzahl von Individuen, auch *Tournamentgröße* genannt, wird

zufällig ausgewählt. Diese Individuen führen nun untereinander *Turniere* aus, aus denen jeweils ein bzw. mehrere Gewinner hervorgehen. Den Gewinnern ist es erlaubt, die *Verlierer* (ggf. variiert) in der Population zu ersetzen. Das einfachste Turnier hat die Turniergröße 2, wobei das Individuum mit der besseren Fitness (im direkten Vergleich) das Individuum mit der schlechteren Fitness (z.B. durch Mutation variiert) ersetzt. Durch die Turniergröße ist es möglich, im Rahmen der Evolution einen *Selektionsdruck* zu erzeugen. Die Turnierselektion ist im Bereich der GP sehr beliebt und ist auch häufig unter dem Namen *Steady-State-Selektion* zu finden.

Mutation

Mit der Mutation verfolgt man das Ziel, gänzlich andere Lösungen zu finden - also Lösungen, die mit der Kreuzung zweier Individuen allein nicht generiert werden können. Die Mutation soll gewährleisten, dass die Vielfalt (= viele unterschiedliche Lösungen) in der Population erhalten bleibt. Eine sehr bekannte Mutationsvariante ist die sogenannte Bit-Flip-Mutation.

Der Algorithmus der Bit-Flip-Mutation ist folgendermaßen definiert. Ähnlich der Kreuzung werden hintereinander alle Individuen der Population betrachtet. Bei jedem Individuum wird entschieden, ob es mutiert wird oder nicht (die Wahrscheinlichkeit für eine Mutation wird durch den Parameter Mutationsrate eingestellt). Im Falle einer Mutation wird zufällig eine Stelle innerhalb des Individuums verändert (siehe Abb. 2.3).

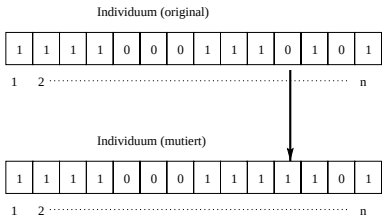


Abbildung 2.3: Mutation eines Individuums

Reproduktion

Die Reproduktion ist die nachempfundene Einzelfortpflanzung. Der Nachfahre des Individuums ist, sofern keine Mutation auf ihn angewandt wurde, eine identische Kopie seines Vorfahren.

Rekombination

Durch Rekombination gehen die Anlagen von zwei Eltern in den Nachfahren ein. Erwünschtes Ziel ist, dass auf diesem Weg die guten Erbanlagen zweier Individuen

im Durchschnitt noch bessere Nachfahren produzieren. Die Rekombination wird in GA in Form des Crossovers umgesetzt.

Kreuzung: Crossover (Sexual Recombination)

Sinn der Kreuzung ist es, Lösungen miteinander zu kombinieren und daraus neue, nämlich bessere zu erzeugen. Im Zusammenhang mit der Selektion lautet die grundlegende Annahme dahinter: ”GUT + GUT = BESSER”. Es gibt mehrere Arten, die Kreuzung in einem Genetischen Algorithmus durchzuführen. Eine sehr bekannte Kreuzungsvariante ist das 1-Punkt-Crossover (siehe [10]).

Der Algorithmus des 1-Punkt-Crossovers ist wie folgt aufgebaut. Die ersten zwei Individuen der Population werden herausgenommen. Mit einer bestimmten Wahrscheinlichkeit, die als Kreuzungsrate bezeichnet wird, werden sie ”gepaart” oder andernfalls unverändert in die Population zurückgelegt. Die ”Paarung” (Kreuzung) läuft folgendermaßen ab: Zunächst werden beide Individuen an der gleichen, zufällig bestimmten Position zerschnitten. Die entstandenen Teile werden kreuzweise miteinander verbunden, so dass zwei neue Indviduen (die Kinder) entstehen (siehe Abbildung 2.4). Hier wird der erste Teil des ersten Arrays mit dem ersten Teil des zweiten Arrays vertauscht. Die neu entstandenen Individuen werden an in die Population eingefügt und ersetzen in der Regel entweder die Eltern oder aber die Verlierer eines Turniers. Dann werden die nächsten zwei zu kreuzenden Individuen herausgenommen, und die Prozedur beginnt von vorne bis alle Individuen betrachtet wurden.

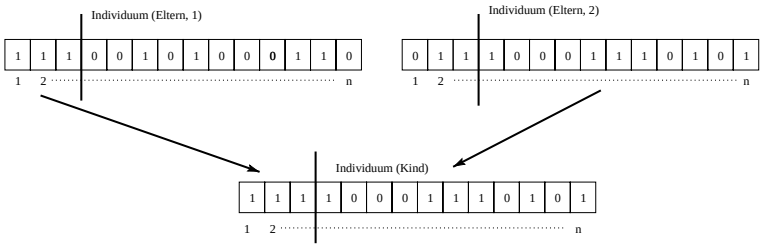


Abbildung 2.4: Kreuzung zweier Individuen (das zweite Kind setzt sich aus den anderen beiden Teilen zusammen)

Crossover - Erweiterte Betrachtungen

Der Crossoveroperator ist in seiner Wirkungsweise in der GP umstritten, obwohl die meisten GP Anwender diesen Operator (häufig sogar hauptsächlich) einsetzen. Es ist in der Zukunft noch zu klären, ob der *konstruktive* Effekt dieses Operators den *destruktiven* Effekt überwiegen kann. Teilweise wird sogar darüber diskutiert, ob dieser Operator der Evolution *überhaupt* nutzen kann. Diese Fragen können bis zum heutigen Tage leider nicht vollständig beantwortet werden, aber es gibt einige

Theorien, welche die These stützen, dass Crossover in seiner Wirkungsweise einer *Makromutation* sehr nahe kommt und der Evolution durchaus wichtige Impulse geben kann.

Der destruktive Effekt des Crossoveroperators liegt darin begründet, dass mit einer hohen Wahrscheinlichkeit (durch ungünstige Kreuzungspunkte) eine gute Funktionseinheit eines Individuums wieder zerstört werden kann. Ebenso ist es (mit vermutlich geringerer Wahrscheinlichkeit) möglich, dass durch günstige Kreuzungspunkte zwei gute Teilindividuen zusammengeführt werden und in der Summe ein noch besseres Individuum bilden. Diese Annahme führte dazu, dass das unter 2.1.1 vorgestellte Schema-Theorem auch auf die Genetische Programmierung übertragen wurde (s. auch [9]). Ähnlich wie beim GA-Schema-Theorem werden auch beim GP-Schema-Theorem Schemata definiert, nur mit dem Unterschied, dass GP-Schemata Teilprogramme beinhalten. Entsprechend kann auch die Bedeutung der *Building Blocks* auf die GP übertragen werden (s. auch [9]):

Bei einem Building Block der GP kann es sich um irgendeinen Baum (im Fall der Baum-Repräsentation) handeln, welcher sich als Teil der Population präsentiert. Die GP *Building Block Hypothese* folgt der Argumentation der GA Building Block Hypothese (s. 2.1.1): Gute Building Blocks begünstigen die Fitness des Individuums, welches den entsprechenden Building Block enthält. Daher werden Individuen mit guten Building Blocks auch eher zur Variation und zur Reproduktion herangezogen, weshalb auch hier der Schluss nahe liegt, dass sich gute Building Blocks auch wesentlich schneller über die Population ausbreiten. Dies führt zu der Annahme, dass ein System mit Crossover auch schneller gute Lösungen entwickelt als ein System, welches nur Mutation und Reproduktion unterstützt, da die Wahrscheinlichkeit besteht, dass durch die Rekombination guter Building Blocks noch bessere Building Blocks evolviert werden.

Crossover - Weiterentwicklungen

Insgesamt lässt sich die Crossover-Problematik recht allgemein auf zwei wesentliche Punkte reduzieren:

1. *Verminderung des destruktiven Effekts* durch den Schutz von guten Building Blocks.

2. Steigerung der *Systemperformance* bei implementierten Schutzmechnismen, da diese Mechnismen sehr Rechenzeitaufwenig sein können.

Es existieren mehrere Ansätze, wovon nur zwei hier vorgestellt werden sollen. Für weitergehende Informationen sei auf [9] verwiesen.

1. Die biologische Variante - homologes Crossover:

   Im Rahmen der Biologie des Menschen hat das Crossover eine sehr entscheidende Bedeutung. Während das Auftreten von Mutationen bei der genetischen Informationsübertragung eher unerwünscht ist, ist das Auftreten des

Crossovers unverzichtbar. Im Rahmen der menschlichen Biologie arbeitet dieser Operator (über einen Zeitraum von Milliarden von Jahren) offensichtlich äusserst erfolgreich. Doch wo liegt der Unterschied zwischen der GP und der Biologie ?

Im Gegensatz zur GP arbeitet das biologische Crossover stets in einem *semantischen Zusammenhang* zwischen beiden zu kreuzenden DNA [2] - Strängen. Die DNA kann in funktionale Abschnitte unterteilt werden. Ein Crossover zwischen zwei DNA - Strängen erfolgt also stets an funktional zusammenhängenden Stellen. Das biologische Crossover ist im Gegensatz zur GP *homolog*. Der biologische Crossover-Operator arbeitet sehr fein bis auf die molekulare Ebene. Es werden nur Codebestandteile an gleichen oder ähnlichen Stellen ausgetauscht.

Da das biologische Crossover über Jahre hinweg seine Leistungsfähigkeit unter Beweis gestellt hat, liegt es nahe zu vermuten, dass eine Crossovervariante umso erfolgreicher ist, je ähnlicher diese Variante der biologischen Variante ist. Im Allgemeinen ist es jedoch sehr schwierig gerade die Eigenschaften des biologischen Crossovers für die GP zu implementieren, welche das biologische Crossover erfolgreich machen. Einen Ansatz liefert das so genannte *Intelligent Crossover*, dessen Algorithmus im Folgenden (für die Baum Reprsentation) vorgestellt sein soll:

(a) Zwei Bäume werden zufällig gewählt.

(b) Berechnung der strukturellen Ähnlichkeit.

Durch die Berechnung von Distanzen ist die Berechnung eines Ähnlichkeitsmaßes für Strukturen mit variablen Längen möglich. Es ist nur eine Reihenfolge festzulegen, in welcher der Baum traversiert werden soll, z.B. *Depth First*. Alle Kanten werden entsprechend der Traversion nummeriert. Wenn für jede Kante k, welche die zwei Knoten A und B verbindet, im größeren Baum ein Pfad mit kürzester Distanz (zwischen den Knoten A' und B') im kleineren Baum *imin(k)* berechnet wurde, kann nun die Distanz bezüglich beider Werte ($D_S(k, imin(k))$) berechnet werden. Durch die Addition aller dieser Werte

$$\bar{D_S} = \sum_k D_S(k, imin(k)) \tag{2.6}$$

mit anschließender Normalisierung durch die Teilung durch $\bar{D_S}$ kann ein quantitativer Wert $D_S^N(k, imin(k))$ für die strukturelle Ähnlichkeit berechnet werden.

(c) Berechnung der funktionalen Ähnlichkeit.

[2] Träger der menschlichen Erbinformation, also das Äquivalent zu den GP Programmen

Typ	Wahrscheinlichkeit
Strukturelle Ähnlichkeit (S)	$D_S^N(k, imin(k))$
Funktionale Ähnlichkeit (F)	$1 - D_F^N(k, jmin(k))$
S / F	$D_S^N(k, imin(k)) \cdot (1 - D_F^N(k, jmin(k)))/n$

Tabelle 2.1: Wahrscheinlichkeiten für Crossoverpunkte.

Die funktionale Ähnlichkeit zweier Bäume kann dadurch berechnet werden, dass die Ausgabe für *jeden* Baum bezüglich einer kleinen Menge von Testdaten berechnet wird. Die quantitative funktionale Ähnlichkeit D_F wird schließlich durch

$$D_F(k, jmin(k)) = \sum_k |O^{(\alpha)} - O^{(\alpha)}_{jmin(k)}| \tag{2.7}$$

berechnet. Auch dieser Wert sollte durch die Teilung der Gesamtsumme $\bar{D_F}$ normiert werden:

$$D_F^N(k, jmin(k)) = \frac{D_F(k, jmin(k))}{\sum_k D_F(k, jmin(k))} \tag{2.8}$$

Die zuvor genannten Berechnungen können nun verwendet werden, um die Wahrscheinlichkeiten für bestimmte Kreuzungspunkte zu bestimmen. Die folgende Tabelle (1c) stellt beispielhaft die Wahrscheinlichkeiten dar, mit der Crossover an einer Kante k durchgeführt werden könnte, wenn ein bestimmter Berechnungstyp (oder die Kombination beider Berechnungstypen) angewendet wird. Dabei handelt es sich bei *n* um einen Faktor für die Normalisierung:

(d) Bestimmung der Crossover-Punkte

2. Brood Recombination

In der Natur hat eine Spezies in den meisten Fällen mehrere Nachkommen, wovon die Stärksten überleben. Dieses Prinzip wird bei der sogenannten *Brood Recombination* angewendet. Bei diesem Verfahren wird eine *Brood Size n* festgelegt, welche definiert, wieviele Rekombinationen (auf Basis der Eltern) gebildet werden sollen, um schließlich die beste Rekombination aus diesen n Rekombinanten zu wählen.

Bei dieser Variante wird dem destruktiven Effekt des Crossoveroperators entgegengewirkt, da schließlich aus mehreren Rekombination die scheinbar erfolgreichste gewählt wird, wodurch auch die Wahrscheinlichkeit erhöht wird, gute Building Blocks kombiniert zu haben.

Steady-State-GP Algorithmus

An dieser Stelle sei alternativ zu den generationalen Methoden ein allgemeiner (problemunabhängiger) Steady-State Algorithmus vorgestellt:

1. Initialisieren der Population.
2. Wähle zufällig eine Subpopulation der aktuellen Population. Auf diese Subpopulation soll nun die Turnierselektion angewendet werden.
3. Berechne den Fitnesswert für jeden Turnierteilnehmer.
4. Selektiere den oder die Gewinner der Turniere.
5. Wende auf die Gewinner genetische Operatoren an.
6. Ersetze die Verlierer der Turniere durch die in 5. variierten Individuen.
7. Wiederhole Schritte 2. bis 7. bis das Terminierungskriterium erfüllt ist.
8. Wähle das beste Individuum in der Population als Ausgabe des Algorithmus.

2.2 Kinematische Modellierung

Dieser Abschnitt soll auf der einen Seite eine Einführung in die Robotik geben und auf der anderen Seite Methoden zur Modellierung von Robotern vorstellen.

Dazu gehört die Abgrenzung der relevanten Attribute eines Roboters und deren formale Repräsentation im Hinblick auf die Simulation, die Bestandteil des Projekts sein soll. Es werden Algorithmen zur Lösung wichtiger Probleme, die im Zusammenhang mit speziellen Attributen entstehen, beschrieben.

2.2.1 Einführung in die Robotik

Da der größte Teil der Theorie in der Kinematischen Modellierung durch die Robotik eingeführt wurde und da wir in unserem Projekt hauptsächlich mit Robotern arbeiten, soll an dieser Stelle ein wenig auf die Robotik eingegangen werden.

2.2.1.1 Definition des Begriffs *Roboter*

Das *Robot Institute of America* hat 1979 den Begriff *Roboter* folgendermaßen definiert:

> "A reprogrammable, multifunctional manipulator designed to move material, parts, tools, or specialized devices through various programmed motions for the performance of a variety of tasks."

In der industriellen Produktion sind viele Aufgaben nicht auf herkömmliche Weise zu automatisieren. Sie sollen aber möglichst nicht von Menschen ausgeführt werden, da sie monoton oder gefährlich sind. Roboter können Bewegungen vollziehen, die denen eines menschlichen Arms ähneln.

Roboterkontrollsysteme können für verschiedenste Arten von Aufgaben eingesetzt werden. Industrieroboter sind Vielzweckmaschinen, die im Verbund flexible automatisierte Produktionseinheiten bilden.

2.2.1.2 Klassifizierung von Robotern

Es gibt drei Grundklassen von Robotern.

Manipulator-Robotersysteme: Typischerweise besteht der Manipulator aus einem Roboterarm und einem *Effektor*. Während der Arm den Effektor an die gewünschte Stelle bringt, führt dieser dann die eigentliche Aufgabe aus (z.B. ein Greifer, der ein Werkstück aufnimmt, oder ein Schweißgerät).

Mobile Robotersysteme: Darunter versteht man automatisch gesteuerte Transportsysteme, um Werkstücke oder Werkzeuge zwischen Lager und Maschinen zu transportieren.

Auf einem mobilen Robotersystem kann auch ein Manipulator installiert sein.

Informations- und Kontrollroboter: Am Besten stellt man sich hier einen Roboter vor, der automatisch Informationen durch Messungen oder installierte Kameras sammelt.

Allgemein kann hier aber die Aufgabe sein, Informationen zu verarbeiten, zu übertragen oder Kontrollsignale zu geben.

Zwar ist in dieser Klassifizierung nicht die Art der Fortbewegung festgelegt, aber Laufroboter haben in dieser Aufteilung eigentlich keinen Platz. Sie stellen eher Exoten dar.

Obwohl die in unserer PG zu simulierenden Laufroboter mobil sind, haben sie mehr mit Manipulator-Robotersystemen gemein als mit mobilen Robotersystemen. Die Methoden zur Modellierung, Steuerung und Simulation von Roboterarmen sind nämlich im Prinzip die gleichen wie bei Laufrobotern, während man sich im Zusammenhang mit mobilen Robotern eher mit Problemen wie der Pfadplanung beschäftigt.

Manipulator-Robotersysteme lassen sich wiederum unterscheiden nach dem Grad ihrer Autonomität. Es gibt Systeme, die direkt von einem Benutzer gesteuert werden, andere werden ferngesteuert.

Für uns sind jedoch die *automatischen Roboter* von Interesse, die von einem Programm gesteuert werden. Roboter der sogenannten ersten Generation führen dabei ihr Programm aus ohne dabei Informationen aus der Umwelt zu berücksichtigen.

Adaptive Roboter der zweiten und dritten Generation besitzen Sensoren. Von der so erhaltenen Information hängt die Ausführung des Programms ab. Auch die von uns betrachteten Roboter sollen Sensoren besitzen dürfen.

2.2.1.3 Grundelemente eines Roboters

Roboter bestehen aus zwei verschiedenen Grundelementen: *Gliedern* und *Gelenken*.

Glieder sind die einzelnen Bestandteile des Roboters, die in Modellen meist als Festkörper betrachtet werden, d.h. ihre Geometrie und ihre physikalischen Attribute sind unveränderlich. Bei feststehenden Roboterarmen oder Laufrobotern, die zu einem Zeitpunkt den Boden berühren, gilt auch die Basis (also der Boden) als Glied.

Bei der Modellierung sind kinematische und physikalische Parameter von Gliedern zu berücksichtigen. Typische kinematische Parameter sind die Länge und der Winkel der Gelenkachsen, die mit dem Glied verbunden sind. Die genaue Notation dieser Parameter wird später vorgestellt.

Typische physikalische Parameter sind Masse, Ort, Massepunkt und Trägheitstensor des Glieds.

Gelenke verbinden jeweils genau zwei Glieder. Von der Art des Gelenkes hängt ab, inwiefern die Bewegungen der Glieder voneinander abhängig und somit eingeschränkt sind. Ein Gelenk kann aktiv sein, also selber Bewegungen verursachen. Dies wird in der Praxis z.B. durch Drehmotoren oder hydraulische Mechanismen realisiert.

Aus Sicht der Modellierung sind Gelenke jedoch masselos. Sie werden nur durch eine Achse repräsentiert sowie einer Anzahl an Parametern, z.B. Art des Gelenkes (Rotation oder Translation) und maximale Auslenkung.

2.2.1.4 Übliche Maßeinheiten in der Simulation

In den in diesem Abschnitt vorgestellten Modellierungsmethoden und Algorithmen wird nicht genauer auf verwendete Maßeinheiten eingegangen. Bei der Entwicklung einer Simulation muss man sich darüber natürlich Gedanken machen. Üblich sind die Einheiten Meter, Kilogramm, Sekunde, Newton.

2.2.1.5 Beispiele

Als Beispielabbildungen sind für Industrieroboter 2.5 und Laufroboter 2.6 anzusehen.

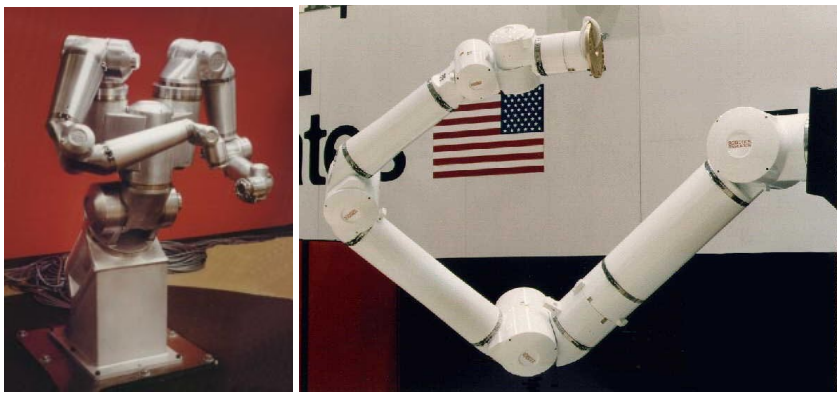


Abbildung 2.5: Zweiarmiger Industrieroboter und einarmiger NASA-Roboter

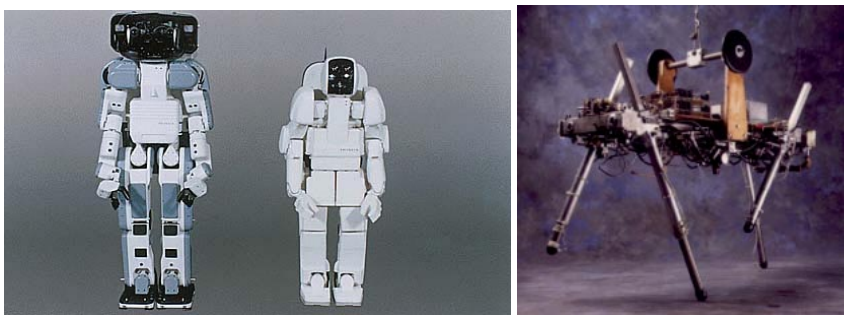


Abbildung 2.6: Zweibeiniger Laufroboter Asimov der Honda Co. und vierbeiniger Laufroboter Quadruped des MIT

2.2.2 Mathematische Grundlagen

Für die Modellierung von Robotern werden einige mathematischen Grundlagen benötigt.

2.2.2.1 Affine Abbildungen

Eine affine Abbildung f im $\Re^3$ ist durch eine 3×3 Matrix $\mathbf{A}$ und einen Vektor $\bar{t} \in \Re^3$ gegeben und definiert durch

$$f(x) := \mathbf{A} \cdot \bar{x} + \bar{t}$$

Um einen Punkt $\bar{x} \in \Re^3$ um einen Vektor $\bar{t}$ zu verschieben (Translation) muss für $\mathbf{A}$ einfach die Einheitsmatrix gewählt werden.

Die Multiplikation folgender Matrizen $\mathbf{A}_{\theta,\mathbf{T}}$ für einen Winkel θ und eine Achse $T \in \{X,Y,Z\}$ mit einem Vektor $\bar{x}$ rotiert diesen um die T-Achse um den Winkel θ. Es handelt sich also auch um eine affine Abbildung mit $\bar{t} = \bar{0}$.

Rotation um die X-Achse:

$$\mathbf{A}_{\theta,\mathbf{X}} := \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{pmatrix} \tag{2.9}$$

Rotation um die Y-Achse:

$$\mathbf{A}_{\theta,\mathbf{Y}} := \begin{pmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{pmatrix} \tag{2.10}$$

Rotation um die Z-Achse:

$$\mathbf{A}_{\theta,\mathbf{Z}} := \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.11}$$

2.2.2.2 Orientierungsdarstellung durch Eulerwinkel

Die Orientierungsbeschreibung mit Eulerwinkeln ist durch sukzessive Rotation des Bezugskoordinatensystems um die Winkel α, β und γ gekennzeichnet, also:

1. Drehung des ursprünglichen Systems um die z-Achse um den Winkel α.
2. Weitere Drehung um die neuentstandene y-Achse um den Winkel β.
3. Letzte Drehung diese neuen Systems um die neue z-Achse um den Winkel γ.

Die so definierte Orientierung des Körpers wird als $Euler(\alpha,\beta,\gamma)$ bezeichnet. Es gilt also:

$$Euler(\alpha,\beta,\gamma) = Rot(\vec{z},\alpha) \cdot Rot(\vec{y},\beta) \cdot Rot(\vec{z},\gamma) \tag{2.12}$$

2.2.2.3 Orientierungsdarstellung durch Roll-Pitch-Yaw-Winkel

Die Orientierungsdarstellung Roll-Pitch-Yaw-Winkel ist ebenfalls durch drei sukzessive Rotationen um drei Winkel Θ_1, Θ_2 und Θ_3 gekennzeichnet:

1. Drehung des ursprünglichen Systems um die z-Achse um den Winkel Θ_1.
2. Weitere Drehung um die neuentstandene y-Achse um den Winkel Θ_2.
3. Letzte Drehung diese neuen Systems um die neue x-Achse um den Winkel Θ_3.

Diese Orientierung wird als $RPY(\Theta_1,\Theta_2,\Theta_3)$ bezeichnet. Es gilt daher:

$$RPY(\Theta_1,\Theta_2,\Theta_3) = Rot(\vec{z},\Theta_1) \cdot Rot(\vec{y},\Theta_2) \cdot Rot(\vec{x},\Theta_3) \tag{2.13}$$

Mit dieser Darstellungsform werden sämtliche Rotationen im Rahmen von Initialpositionsberechnung durchgeführt. Dies wird aber in späteren Kapiteln noch ausführlich behandelt.

2.2.2.4 Drehvektor, Drehwinkel

In dieser Darstellung werden ein Drehvektor $\vec{k}$ und ein Drehwinkel Θ so bestimmt, dass ein Körper mit Normalorientierung bei der Rotation um den Vektor $\vec{k}$ um den Winkel Θ die gegebene Orientierung erhält. Es wird also eine Rotation um diese (beliebige) Rotationsachse durchgeführt. Dafür existiert die weiter unten angegebene Transformationsmatrix, die später bei den Berechnungen der Initialpositionen in Anwendung kommt. Diese Matrix setzt sich aus sieben Einzeltransformationen zusammen. Als erstes hat man eine Translationsmatrix, welche die Gerade (Rotationsachse) in den Ursprung transformiert. Dann werden zwei Rotationen durchgeführt, um die verschobene Achse auf eine beliebige Koordinatenachse zu drehen. Nun dreht man um den gewünschten Winkel (um den man um die Achse ursprünglich drehen wollte) um diese Koordinatenachse. Anschließend führt man zwei inverse Rotationen aus um die vorherigen Rotationen (die die Achse auf die Koordinatenachse drehten) auszugleichen. Am Ende verschiebt man via Translationsmatrix die Achse wieder an ihre ursprüngliche Position zurück. Insgesamt sieht die Matrix folgendermaßen aus:

$$Rot(\vec{v},\alpha) = \begin{pmatrix} v_x^2u+c & v_yv_xu-v_zs & v_zv_xu+v_ys \\ v_xv_yu+v_zs & v_y^2u+c & v_zv_yu-v_xs \\ v_xv_zu-v_ys & v_yv_Zu-vxs & v_z^2u+c \end{pmatrix} \tag{2.14}$$

mit $s = \sin(\alpha)$, $c = \cos(\alpha)$ und $u = 1-\cos(\alpha)$.

2.2.2.5 Homogene Darstellung

Um affine Abbildungen kompakter darstellen zu können und ihre Komposition zu vereinfachen, arbeitet man mit der sogenannten *homogenen Darstellung* von affinen Abbildungen und Vektoren. Dabei bezeichnet x_i im folgenden die i-te Komponente des Vektors $\bar{x}$.

Vektoren in homogener Darstellung erhalten eine zusätzliche vierte Komponente. Die Beziehung zwischen einem Vektor $\bar{x}$ und seiner homogenen Darstellung $\bar{y}$ ist gegeben durch

$$\bar{x} = \begin{pmatrix} y_1/y_4 \\ y_2/y_4 \\ y_3/y_4 \end{pmatrix}. \tag{2.15}$$

Für eine affine Abbildung $f : \Re^3 \to \Re^3$, die durch die Matrix $\mathbf{A} = (a_{ij})_{i,j}$ und den Vektor $\bar{t} = (t_1,t_2,t_3)^T$ charakterisiert ist, gilt

$$f(\bar{x}) = \begin{pmatrix} g_1(y) \\ g_2(y) \\ g_3(y) \end{pmatrix} \tag{2.16}$$

für $\bar{y} = (x_1,x_2,x_3,1)^T$ und die Abbildung $g : \Re^4 \to \Re^4$,

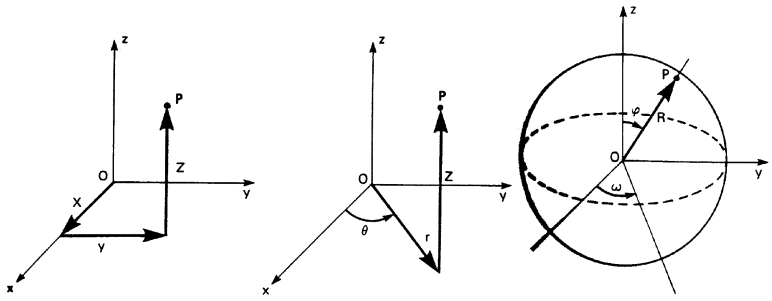


Abbildung 2.7: Punkte im kartesischen, zylindrischen und spherischen Koordinatensystem.

$$g(\bar{y}) := \begin{pmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \bar{y}. \tag{2.17}$$

Das kann man leicht nachrechnen. Insbesondere ist $g_4(\bar{y}) = 1$.

In homogener Darstellung lassen sich affine Abbildungen nun leicht durch Matrizenmultiplikation hintereinander ausführen.

2.2.2.6 Koordinatensysteme

Punktpositionen im dreidimensionalen Raum kann man auf verschiedene Art darstellen. In allen Koordinatensystemen gibt es einen ausgezeichneten Punkt, den *Ursprung*, sowie drei *Achsen* X, Y und Z. Inwiefern Punkte relativ zu Ursprung und Achsen dargestellt werden, hängt vom verwendeten Koordinatensystem ab.

Das Kartesische Koordinatensystem Ein Punkt p wird durch einen Vektor $\bar{x}$ mit drei Komponenten dargestellt, die als Skalare dreier Basisvektoren des Vektorraums $\Re^3$ aufzufassen sind (siehe Abbildung 2.7).

Verwendet man die *kanonische Basis*

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \tag{2.18}$$

so entspricht x_1 der Projektion von p auf die X-Achse, x_2 der Projektion auf die Y-Achse und x_3 der Projektion auf die Z-Achse.

Ist die Darstellung $\bar{x}$ eines Punktes p bezüglich einer Basis **B** bekannt, und liegen die Vektoren $(\bar{b}_1, \bar{b}_2, \bar{b}_3)$ von **B** bezüglich einer Basis **C** vor, so ergibt sich die Darstellung $\bar{y}$ von p bezüglich **C** einfach durch $\bar{y} = \mathbf{B} \cdot \bar{x}$.

Liegt umgekehrt die Darstellung $\bar{x}$ eines Punktes p ebenso wie die Vektoren einer Basis **B** bezüglich einer Basis **C** vor, so gilt $\mathbf{B} \cdot \bar{y} = \bar{x}$ für die Darstellung $\bar{y}$ von p bezüglich **B**. Demnach gilt $\bar{y} = \mathbf{B}^{-1} \cdot \bar{x}$. Da **B** Basis des $\Re^3$ ist, existiert $\mathbf{B}^{-1}$.

Zylindrisches Koordinatensystem Im zylindrischen Koordinatensystem wird die Position eines Punktes p in der X-Y-Ebene durch einen Rotationswinkel θ und einen Radius r dargestellt (siehe Abbildung 2.7). Die Höhe wird wie im kartesischen Koordinatensystem als Skalar des Z-Einheitsvektors festgehalten.

Sphärisches Koordinatensystem Im sphärischen Koordinatensystem wird die Position eines Punktes p durch zwei Rotationswinkel φ und ω und einen Radius R dargestellt (siehe Abbildung 2.7). Der Radius beschreibt eine Kugel mit Mittelpunkt im Ursprung, die zwei Winkel einen Randpunkt dieser Kugel.

2.2.2.7 Koordinatensystem-Transformation

Bei der Modellierung von Robotern ist mit jedem Glied ein eigenes (orthonormales) Koordinatensystem assoziiert. Vektoren, die Positionen bezüglich eines bestimmten Koordinatensystems beschreiben, sollen in die Darstellung bezüglich eines anderen Koordinatensystems überführt werden. Dies wird für die Lösung des Problems der Vorwärtskinematik benötigt (siehe auch Abbildung 2.8).

Das orthonormale Koordinatensystem eines Gliedes i wird beschrieben durch die Position $\vec{0}_i$ seines Ursprungs sowie seine Achsen-Vektoren $\vec{x}_i$, $\vec{y}_i$ und $\vec{z}_i$. Diese vier Vektoren müssen jedoch auch bezüglich eines Koordinatensystems dargestellt werden. Deswegen soll $\vec{r^i}$ ausdrücken, dass der Vektor $\vec{r}$ bezüglich des Koordinatensystems des Glieds i dargestellt ist.

Sei $\mathbf{A_i^{i-1}}$ die Rotationsmatrix zwischen den Koordinatensystemen $i-1$ und i, und $r_{i-1,i}^{i-1}$ der Distanzvektor zwischen den Ursprüngen der Koordinatensysteme $i-1$ und i.

Die Transformation eines Vektors $\bar{r}_p^i$, der die Position des Punktes p bezüglich des Koordinatensystems i beschreibt, zum Vektor $\bar{r}_p^{i-1}$, der die Position des Punktes p bezüglich des Koordinatensystems $i-1$ beschreibt, ergibt sich nun folgendermaßen:

$$\bar{r}_p^{i-1} = \mathbf{A_i^{i-1}} \cdot \bar{r}_p^i + \bar{r}_{i-1,i}^{i-1} \tag{2.19}$$

Veranschaulicht werden die beiden Koordinatensysteme in Überdeckung gebracht, indem zunächst die drei Achsen von $i-1$ so rotiert werden, dass sie zu denen von i parallel sind. Anschließend wird der Ursprung von $i-1$ in den von i verschoben. So errechnet sich der Positionsvektor von p bezüglich $i-1$ aus dem bezüglich i.

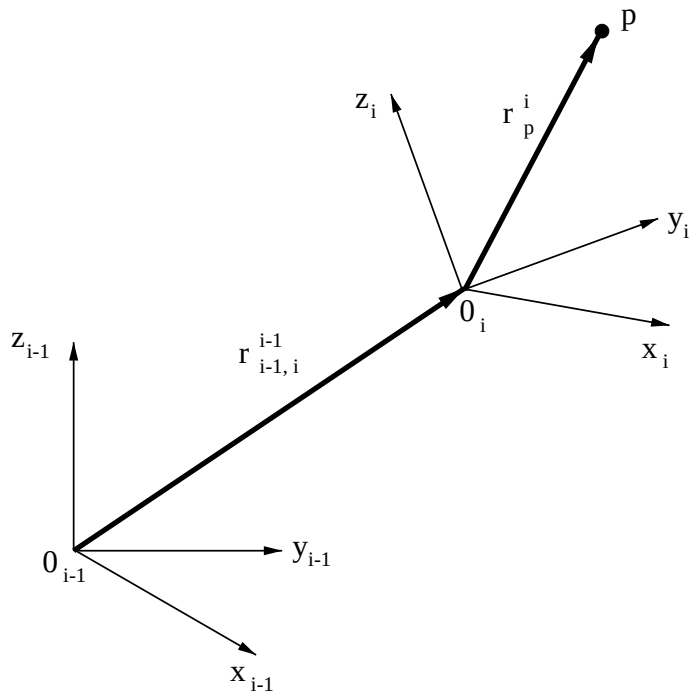


Abbildung 2.8: Transformation von Koordinatensystem i nach $i-1$

Die Abbildung hat die Form einer affinen Transformation, die homogen durch folgende Transformationsmatrix $\mathbf{T_i^{i-1}}$ dargestellt werden kann:

$$= \left(\begin{array}{ccc|c} & \mathbf{A_i^{i-1}} & & \bar{r}_{i-1,i}^{i-1} \\ \hline 0 & 0 & 0 & 1 \end{array} \right) \tag{2.20}$$

2.2.3 Kinematische Modellierung

Im Folgenden werden Methoden zur kinematischen Modellierung von Robotern vorgestellt, die sich mit den Beziehungen zwischen verschiedenen Gliedern durch Gelenke befasst.

2.2.3.1 Grundbegriffe und Definitionen

Aus der Sicht der theoretischen Mechanik sind Roboter *aktive Mechanismen*, allgemein sogenannte komplexe *kinematische Ketten*.

Aktive Mechanismen können in verschiedene Kategorien eingeordnet werden (vgl. Abbildung 2.9).

***einfache aktive Mechanismen*:** Sie bestehen lediglich aus einer kinematischen Kette.

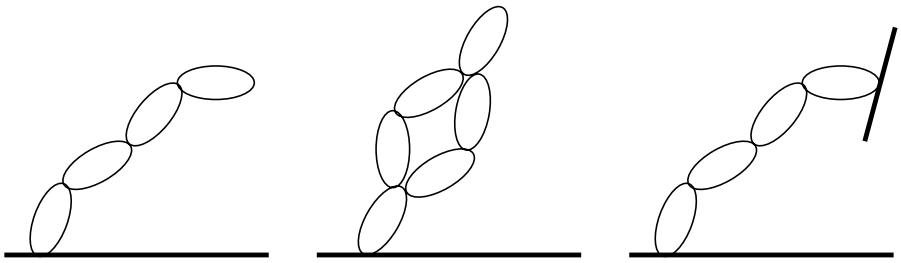


Abbildung 2.9: Beispiele für verschiedene aktive Mechanismen.

***komplexe aktive Mechanismen*:** Sie bestehen aus einer Anzahl kinematischer Ketten.

Einfache kinematische Ketten können *offen* oder *geschlossen* sein. Komplexe kinematische Ketten können deswegen eingeteilt werden in

***verzweigte Ketten*:** Diese Ketten bestehen nur aus mehreren einfachen offenen Ketten.

***kombinierte Ketten*:** Diese Ketten können sowohl offene als auch geschlossene Ketten enthalten.

Zur Veranschaulichung dieser Begriffe kann man sich einen Roboter und seine Umgebung (z.B. den Boden, auf dem er zu einem Zeitpunkt steht) als ungerichteten zusammenhängenden Graphen vorstellen (siehe Abbildung 2.10). Die Glieder des Roboters sind dabei die Knoten, die Gelenke Kanten zwischen Gliederknoten. Dieser Graph lässt sich in disjunkte Teilgraphen zerlegen, deren Knotengrad kleiner gleich zwei ist. Diese Teilgraphen beschreiben einfache kinematischen Ketten.

Enthält der Roboter-Graph also Knoten mit Grad > 2, so handelt es sich um einen komplexen aktiven Mechanismus, ansonsten um einen einfachen. Enthält ein Teilgraph, der eine einfache kinematische Kette beschreibt, Kreise, ist er geschlossen, ansonsten ist er offen.

Ein aktiver Mechanismus heißt *offen*, wenn der entsprechende Graph Knoten mit Grad eins enthält. Ansonsten heißt er *verbunden* oder *geschlossen*. Dies ist nicht zu verwechseln mit einfachen geschlossenen kinematischen Ketten. Ein offener aktiver Mechanismus, der aus mehreren einfachen kinematischen Ketten besteht, kann durchaus Kreise enthalten.

Zwei durch ein Gelenk verbundene Glieder werden ein *kinematisches Paar* genannt.

Insbesondere für die Betrachtung von Laufrobotern ist wichtig, dass sich die Struktur des aktiven Mechanismus mit der Zeit ändern kann. Stehen z.B. beide Beine eines Menschen auf dem Boden, bilden sie eine geschlossene kinematische

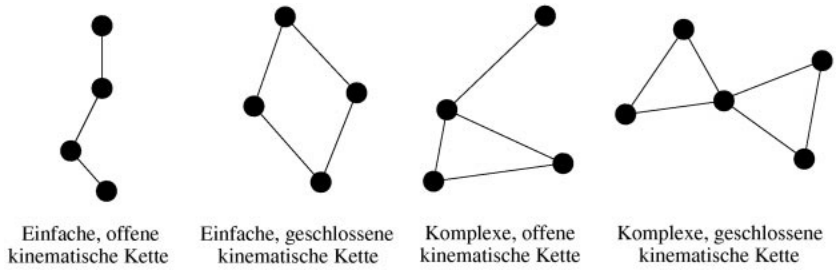


Abbildung 2.10: Die verschiedenen aktiven Mechanismen in Graphdarstellung

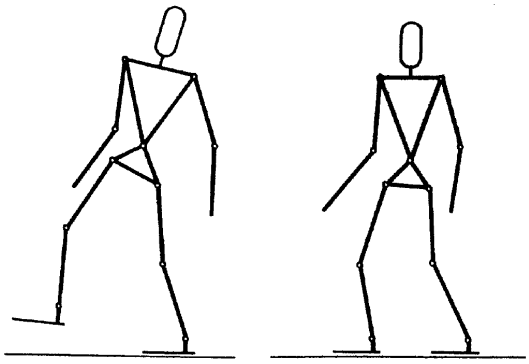


Abbildung 2.11: Die Kinematik eines Laufroboters verändert sich mit der Phase des Laufens.

Kette. Ist ein Bein angehoben, ist dieser Kreis im entsprechenden Graphen aufgebrochen (siehe Abbildung 2.11).

In der folgenden Beschreibung der Methoden zur kinematischen Modellierung werden wir nur auf den Fall einfacher, offener kinematischer Ketten eingehen. In Graphdenkweise sind dies Pfade.

Dies lässt sich jedoch leicht erweitern. Um z.B. das Problem der Vorwärtskinematik mittels des in diesem Referat vorgestellten Verfahrens (siehe Abschnitt 2.2.3.4) auch für kinematische Bäume lösen zu können, wendet man dieses einfach für jeden Pfad eines Blattes zur Wurzel des Baumes an. Somit hat man das Problem wieder auf einfache kinematische Ketten reduziert.

2.2.3.2 Gelenkarten

Kinematische Paare werden in verschiedene Klassen eingeteilt, abhängig von den Einschränkungen der Bewegungen, die das entsprechende Gelenk erlaubt.

Allgemein hat ein Objekt, dessen Bewegung nicht eingeschränkt ist, sechs sogenannte *Freiheitsgrade*:

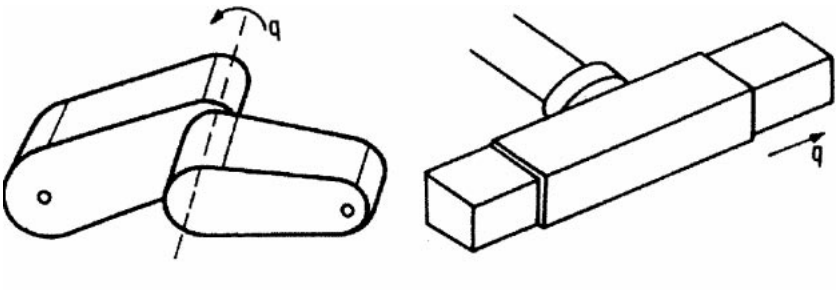


Abbildung 2.12: Links: Ein Rotationsgelenk; Rechts: Ein Translationsgelenk

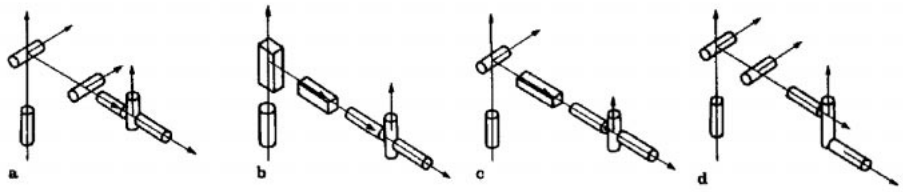


Abbildung 2.13: Aus den Grundgelenktypen lassen sich Gelenke mit mehr als einem Freiheitgrad zusammensetzen.

- Verschiebung, oder *Translation*, entlang der X-, Y- und Z-Achsen. Dies können beliebige Basisvektoren des $\Re^3$ sein.
- Rotation um die X-, Y- und Z-Achsen.

Ein Gelenk stellt eine Einschränkung dieser Freiheitsgrade dar, indem es die Bewegungen der beteiligten Glieder voneinander abhängig macht. Man unterscheidet zwei Grundarten von Gelenken, die die Glieder kinematischer Paare verbinden (vgl. Abbildung 2.12):

Translationsgelenke: Diese Gelenke erlauben die Verschiebung eines Gliedes relativ zum anderen um eine gemeinsame Achse.

Rotationsgelenke: Diese Gelenke erlauben die Rotation eines Gliedes relativ zum anderen entlang einer gemeinsamen Achse.

Diese Grundgelenke haben einen Freiheitsgrad. In der kinematischen Modellierung betrachtet man nur solche. In der Praxis gibt es auch Gelenke, die mehrere Freiheitsgrade (sinnvollerweise bis zu fünf) haben. Beispielsweise kann ein Gelenk gleichzeitig drei Rotationen zulassen (Kugelgelenk).

Diese lassen sich jedoch modellieren durch mehrere einfache Grundgelenke, die durch masselose Körper mit der Geometrie eines Punktes verbunden sind (siehe Abbildung 2.13).

2.2.3.3 Koordinatensysteme

Für die Darstellung von Position und Rotation der Gelenke eines Roboters gibt es unter anderem folgende Möglichkeiten:

Gelenkkoordinaten Skalare Größen, die den Auslenkungszustand eines Gelenkes beschreiben, werden *Gelenkkoordinaten* genannt. Für Translationsgelenke ist dies die aktuelle Verschiebung (in der jeweilig verwendeten Einheit). Für Rotationsgelenke ist dies der aktuelle Winkel, um den das Gelenk gedreht ist. Wo der jeweilige Nullwert liegt, hängt von den Koordinatensystemen der beiden verbundenen Glieder ab (darauf wird später genauer eingegangen).

Die Gelenkkoordinaten der n Gelenke des aktiven Mechanismus werden üblicherweise in einem Vektor $\vec{q} = (q_1, \ldots, q_n)^T$ notiert. Dabei kann jedes Gelenk eine minimale und maximale Auslenkung haben. Das bedeutet $q_i^{min} \leq q_i \leq q_i^{max}$ für $1 \leq i \leq n$. Dabei bezeichnet q_i^{min} die minimale Auslenkung bzw. q_i^{max} die maximale Auslenkung des Gelenkes i.

Externe Koordinaten *Externe Koordinaten* beschreiben die Position und Orientierung der Glieder bezüglich eines Referenzkoordinatensystems. Dies kann verschieden gewählt sein. Im Falle eines fest angebrachten Roboterarms ist dies typischerweise das Koordinatensystem der Basis.

Die Position eines Glieds ist meist in kartesischen Koordinaten notiert, aber auch zylindrische oder sphärische Koordinaten sind denkbar.

Die Orientierung eines Glieds wird durch die drei Rotationswinkel notiert, die die relative Rotation zwischen dem Koordinatensystem des Glieds und dem Referenzkoordinatensystem beschreiben.

Auch hier ergibt sich ein Vektor $\vec{x_e}$, diesmal der Länge $6 \cdot n$ (für die Beschreibung aller n Glieder des aktiven Mechanismus):

$$\vec{x_e} = (x_1, y_1, z_1, \psi_1, \theta_1, \varphi_1, \ldots, x_n, y_n, z_n, \psi_n, \theta_n, \varphi_n)^T. \tag{2.21}$$

2.2.3.4 Das Problem der Vorwärtskinematik

Es gibt zwei kinematische Probleme. Bei dem Problem der Vorwärtskinematik soll für den bekannten Vektor $\vec{q}$ der Gelenkkoordinaten der Vektor $\vec{x_e}$ der externen Gliedkoordinaten bestimmt werden.

Beim Problem der inversen Kinematik hingegen sollen für die externen Koordinaten des letzten Glieds in der kinematischen Kette Gelenkkoordinaten bestimmt werden, die dieses Glied an die gewünschte Position und in die gewünschte Orientierung versetzen. Dies ist insbesondere für Industrieroboter interessant, deren Effektor an eine bestimmte Stelle in einer bestimmten Orientierung gebracht werden soll, um eine Aufgabe zu erledigen. Um ein Steuerungsprogramm zu entwickeln, sollen entsprechende Gelenkauslenkungen ermittelt werden.

Unseres Erachtens ist für unsere Zwecke nur das Problem der Vorwärtskinematik von Interesse.

Kinematische Denavit-Hartenberg Parameter Abhängig von der aktuellen Auslenkung eines Gelenkes werden die Koordinatensysteme der adjazenten Glieder gewählt. Ihre Wahl ermöglicht im Anschluss eine effiziente Konstruktion von Transformationsmatrizen, die, wie im Abschnitt 2.2.2.7 beschrieben, die Lösung des Problems der Vorwärtskinematik durch verknüpfte affine Abbildungen ermöglichen.

Man betrachte eine einfache offene kinematische Kette, die aus n Gliedern und $n-1$ Gelenken besteht, wobei Gelenk i die Glieder $i-1$ und i verbindet ($1 \leq i \leq n$). Dabei wird auch das Referenzkoordinatensystem als ein Glied (Glied 0) aufgefasst. Dessen Verbindung mit Glied 1 wird als Gelenk 1 aufgefasst.

Es werden nun Ursprung $\vec{0}_i$ und Achsen $\vec{x}_i$, $\vec{y}_i$ und $\vec{z}_i$ des lokalen Koordinatensystems für Glied i konstruiert. Dazu werden einige Definitionen benötigt (siehe Abbildung 2.14).

- g_i sei die Rotations- bzw. Translationsachse von Gelenk i.
- Sei h_i eine Gerade, die orthogonal zu g_i und g_{i+1} verläuft und diese außerdem schneidet. Sind die Gelenkachsen nicht parallel zueinander, gibt es nur eine solche Gerade (siehe Abschnitt 2.2.3.4).

  Sind die Gelenkachsen g_i und g_{i+1} parallel, wird h_i so gewählt, dass $d_j = 0$ für das kleinste $j > i$ gilt bei dem g_j und g_{j+1} nicht parallel sind.
- $\vec{n}_i$ sei der Vektor zwischen den beiden Schnittpunkten von h_i mit g_i und g_{i+1}, der von Achse i nach Achse $i+1$ gerichtet ist. Es handelt es sich um einen gemeinsamen Normalenvektor der beiden Gelenkachsen.
- Der kinematische Parameter a_i sei die Länge von $\vec{n}_i$.
- Der kinematische Parameter α_i sei der Winkel zwischen den Projektionen von g_i und g_{i+1} auf eine zu h_i orthogonale Ebene.
- Jede Gelenkachse g_i schneidet zwei Geraden h_{i-1} und h_i. Der euklidische Abstand der beiden Schnittpunkte sei d_i.

Als Ursprung $\vec{0}_i$ des lokalen Koordinatensystems von Glied i wird der Schnittpunkt von g_i und h_i gewählt. Seine Z-Achse $\vec{z}_i$ wird parallel zur Gelenkachse g_{i+1} gewählt. Als seine X-Achse $\vec{x}_i$ wird ein zu $\vec{z_{i-1}} \times \vec{z}_i$ paralleler oder antiparalleler Vektor so gewählt, dass er in Richtung des Gelenkes $i+1$ zeigt. Die Y-Achse $\vec{y}_i$ wird orthogonal zu $\vec{x}_i$ und $\vec{z}_i$ gewählt, so dass $\vec{x}_i \times \vec{y}_i = \vec{z}_i$ gilt.

Die Gelenkkoordinaten sind nun folgendermaßen definiert:

Gelenkkoordinate q_i für ein Rotationsgelenk i: q_i entspricht dem Winkel zwischen $\vec{x_{i-1}}$ und $\vec{x}_i$. Im Falle $q_i = 0$ zeigen die beiden X-Achsen der Glieder $i-1$ und i in die gleiche Richtung. Die Glieder befinden sich in einer Linie.

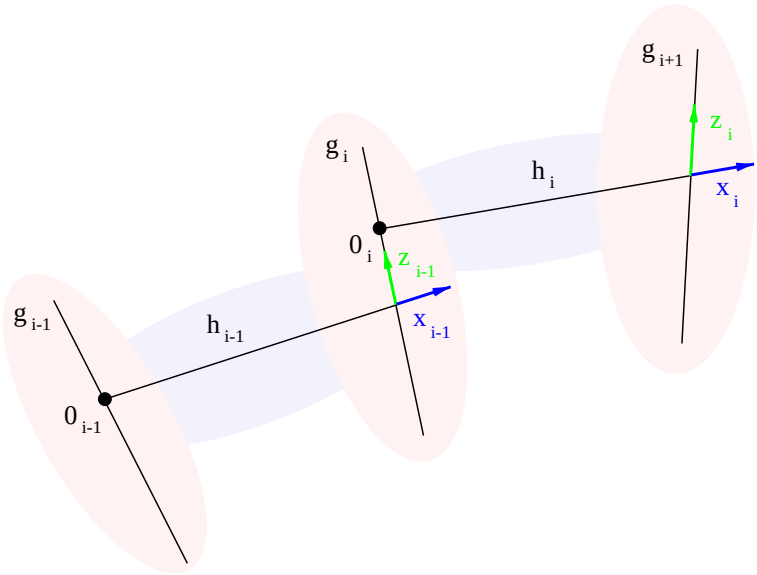


Abbildung 2.14: Veranschaulichung der Denavit-Hartenberg Parameter am Beispiel von drei Rotationsgelenken.

Gelenkkoordinate q_i für ein Translationsgelenk i: q_i entspricht d_i.

Da die Glieder $i-1$ und i nicht gegeneinander verdreht werden können, ist der Rotationswinkel θ_i zwischen $\vec{x_{i-1}}$ und $\vec{x_i}$ fest und wird zusammen mit α_i als kinematischer Parameter aufgefasst.

Der Ursprung des (globalen) Referenzkoordinatensystems wird in Übereinstimmung mit dem Ursprung $\vec{0_1}$ des ersten Gliedes gewählt. Gelenk 1 wird als Rotationsgelenk zwischen erstem Glied und Referenzkoordinatensystem aufgefasst.

Die Berechnung von h_i Seien die Gelenkachsen und h_i folgendermaßen dargestellt:

$$\begin{aligned} g_i(t) &= \bar{a} + t \cdot \bar{u} \\ g_{i+1}(t) &= \bar{b} + t \cdot \bar{v} \\ h_i(t) &= \bar{c} + t \cdot \bar{w} \end{aligned} \tag{2.22}$$

- 1. Fall: g_i und g_{i+1} sind parallel zueinander.

  Es gibt offensichtlich unendlich viele Geraden h_i, die orthogonal zu beiden Gelenkachsen sind und diese schneiden. Wählt man eine dieser Geraden durch Festlegung des Schnittpunkts $\vec{s}$ mit g_i, erhält man somit $\vec{c} := \vec{s}$. Nun fehlt noch $\bar{w}$.

Man ermittelt $\vec{w}$ durch die Suche nach einem Punkt $\vec{p} = g_{i+1}(t_0)$, für den $\vec{w} := \vec{p} - \vec{c} \perp \vec{u}$ gilt. Aufgrund der Parallelität der Gelenkachsen gilt dann auch $\vec{w} \perp \vec{v}$.

Für das Skalarprodukt "$\bullet$" soll also gelten:

$$\begin{array}{rrcl} & \vec{u} \bullet \vec{w} & = & 0 \\ \Leftrightarrow & \vec{u} \bullet (\vec{p} - \vec{c}) & = & 0 \\ \Leftrightarrow & \vec{u} \bullet ((\vec{b} + t_0 \cdot \vec{v}) - \vec{c}) & = & 0 \\ \Leftrightarrow & (\vec{u} \bullet \vec{v}) \cdot t_0 + \vec{u} \bullet (\vec{b} - \vec{c}) & = & 0 \\ \Leftrightarrow & (\vec{u} \bullet \vec{v}) \cdot t_0 & = & \vec{u} \bullet \vec{c} - \vec{u} \bullet \vec{b} \end{array} \tag{2.23}$$

Da sinnvollerweise $\vec{u} \neq \vec{0}$ gilt, existiert also eine Lösung t_0 und somit $\vec{p}$ und auch $\vec{w}$.

- 2. Fall: g_i und g_{i+1} sind nicht parallel zueinander.

  Mit $\vec{w} := \vec{u} \times \vec{v}$ erhält man den Richtungsvektor $\vec{w}$ für h_i, der zu beiden Gelenkachsen orthogonal ist. Nun fehlt noch $\vec{c}$.

  Man sucht Parameter t_0, t_1 und t_2, so dass für $\vec{c} := g_i(t_0)$ folgendes gilt:

$$\begin{array}{rrcl} & h_i(t_1) & = & g_{i+1}(t_2) \\ \Leftrightarrow & g_i(t_0) + t_1 \cdot \vec{w} & = & \vec{b} + t_2 \cdot \vec{v} \\ \Leftrightarrow & \vec{a} + t_0 \cdot \vec{u} + t_1 \cdot \vec{w} & = & \vec{b} + t_2 \cdot \vec{v} \\ \Leftrightarrow & t_0 \cdot \vec{u} + t_1 \cdot \vec{w} - t_2 \cdot \vec{v} & = & \vec{b} - \vec{a} \end{array} \tag{2.24}$$

  Die letzte Gleichung lässt sich als Gleichungssystem folgender Form schreiben:

$$\begin{pmatrix} \vec{u} & \vec{w} & -\vec{v} \end{pmatrix} \cdot \begin{pmatrix} t_0 \\ t_1 \\ t_2 \end{pmatrix} = \vec{b} - \vec{a}. \tag{2.25}$$

  Da $\vec{u}$, $\vec{v}$ und $\vec{w}$ nach Voraussetzung linear unabhängig sind, besitzt das Gleichungssystem eine eindeutige Lösung.

Ferner erhält man den minimalen (senkrechten) Abstand zwischen zwei windschiefen Geraden aus dieser Gleichung unmittelbar als

$$|\vec{w^0} \cdot (\vec{t} - \vec{a})|, \tag{2.26}$$

mit $\vec{w^0}$ als (normierten) Einheitsvektor von $\vec{w}$.

Berechnung der Transformationsmatrizen Nächster Schritt ist, eine affine Transformation in Form einer homogenen Transformationsmatrix zu berechnen, die durch Translation und Rotation das Koordinatensystem eines Glieds $i-1$ mit dem des Glieds i in Überdeckung bringt.

Die Transformation setzt sich aus folgenden Einzelschritten zusammen:

1. Rotation um $\vec{z_{i-1}}$ um den Winkel q_i (für ein Translationsgelenk um den festen Winkel θ_i).
2. Translation um d_i an $\vec{z_{i-1}}$ entlang (für ein Translationsgelenk wird um q_i verschoben).
3. Translation um a_i entlang der nun rotierten Achse $\vec{x_{i-1}} = \vec{x_i}$.
4. Rotation um $\vec{x_i}$ um den Winkel α_i.

Die Transformationen können durch Multiplikation der homogenen Matrizen zu einer homogenen Transformationsmatrix zusammengefasst werden. Für ein Rotationsgelenk *i* sieht das so aus:

$$\begin{aligned}\mathbf{T_i^{i-1}} = & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha_i) & -sin(\alpha_i) & 0 \\ 0 & sin(\alpha_i) & cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ & \cdot \begin{pmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ & \cdot \begin{pmatrix} cos(q_i) & -sin(q_i) & 0 & 0 \\ sin(q_i) & cos(q_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\end{aligned} \tag{2.27}$$

Die Berechnung der globalen Gliedpositionen Nun kann endlich das Grundproblem der Vorwärtskinematik gelöst werden. Liegt ein interessanter Punkt im Koordinatensystem von Glied *i* vor (z.B. der Massepunkt), kann dessen Position im globalen Referenzkoordinatensystem wie in Abschnitt 2.2.2.7 gezeigt durch Multiplikation mit der Transformationsmatrix $\mathbf{T_0^i} = \mathbf{T_0^1} \cdot \mathbf{T_1^2} \cdot \ldots \cdot \mathbf{T_{i-1}^i}$ berechnet werden.

Zu beachten ist, dass, obwohl für jeden Vektor $\vec{q}$ der Gelenkkoordinaten jedes Glied ein neues Koordinatensystem erhält, die Positionen von geometrischen Eigenschaften eines Gliedes (wie z.B. der Massepunkt) relativ zum lokalen Koordinatensystems des Gliedes gleich bleiben.

2.2.4 Modellierung physikalischer Eigenschaften

Um in der Dynamik-Simulation anhand von Kräften und Steuerungen aktiver Gelenke die Auswirkungen auf die Gelenkkoordinaten berechnen zu können, werden neben kinematischen Informationen auch physikalische Informationen über die Bestandteile eines Roboters benötigt.

2.2.4.1 Glieder

Glieder können z.B. durch folgende 4 Attribute charakterisiert werden:

- Massepunkt, z.B. bezüglich des wie in Abschnitt 2.2.3.4 berechneten lokalen Koordinatensystems des Gliedes.
- Masse, typischerweise in Kilogramm angegeben.
- Trägheitstensor. Dabei handelt es sich um eine 3×3-Matrix, die die Trägheitseigenschaften eines Festkörpers bezüglich seiner Raumachsen beschreibt.

  Ihre Berechnung wird an anderer Stelle im Grundlagenkapitel für Dynamik (2.5) angegeben.
- Reibungseigenschaften, z.B. nach dem Modell der *Coloumb'schen Reibung*. In diesem Modell berechnet sich die Reibungskraft, die zwischen zwei parallelen Flächen i und j wirkt, aus dem Reibungskoeffizient μ_{ij}.

  Wichtig dabei ist (wie in Abbildung 2.15 zu sehen), dass Reibung nur dann stattfindet, wenn eine Kraft die beiden Flächen aneinanderdrückt. Wird Fläche j gegen Fläche i mit einer Kraft von N gedrückt (diese Kraft wirkt parallel zum Normalenvektor beider Flächen), und wirkt Kraft T tangential zu den Flächen auf Fläche j, so errechnet sich die Reibungskraft F folgendermaßen (siehe Abbildung 2.15):

  - 1. Fall: $T \leq \mu_{ij} \cdot N$

    In diesem Fall ist $F = T$. Somit wirkt die Reibungskraft F T entgegen. Die Kräfte heben sich auf, es findet keine Verschiebung der beiden Flächen statt. Dies wird *statische Reibung* genannt.
  - 2. Fall: $T > \mu_{ij} \cdot N$

    In diesem Fall ist $F = \mu_{ij} \cdot N$. Da T größer als die Reibungskraft F ist, bewegen sich die beiden Flächen gegeneinander. Dies geschieht mit der effektiven Kraft $T - F$. Dies wird *dynamische Reibung* genannt.

  Die Kraft $\mu_{ij} \cdot N$ kann man sich also als obere Schranke für die Reibungskraft F vorstellen, die überwunden werden muss, um tatsächlich eine Bewegung der Flächen bewirken zu können.

  Zu beachten ist, dass der Reibungskoeffizient μ eine Eigenschaft eines Paares von Festkörpern ist.

  Das Modell der Coloumb'schen Reibung ist nur eine Approximation des tatsächlichen Reibungsverhaltens von Festkörpern. Es gibt auch realistischere Reibungsmodelle.

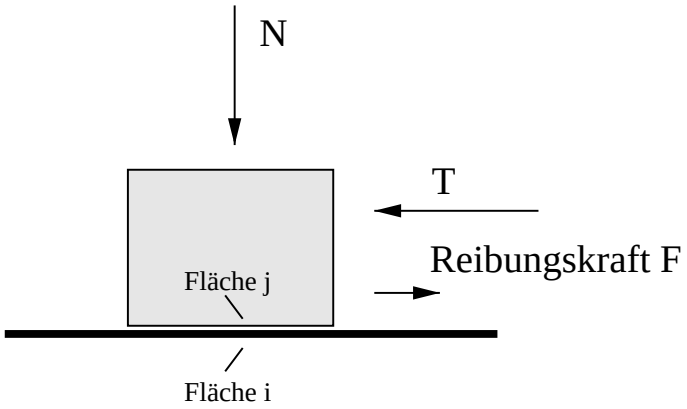


Abbildung 2.15: Die Gravitation drückt die Kiste auf den Boden; daher entsteht Reibungskraft.

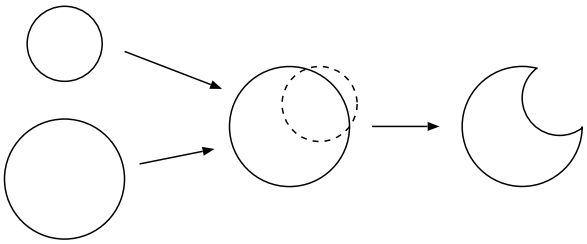


Abbildung 2.16: Zwei Kreise werden mit CSG zu einem „Halbmond“ verarbeitet.

2.3 Geometrische Modellierung

Die geometrische Modellierung von Robotern befasst sich mit der Form der Oberfläche der in der kinematischen Modellierung erkannten Glieder. Erzeugung und Verarbeitung dieser Information ist Gegenstand dieses Abschnittes.

2.3.1 Constructive Solid Geometry

Constructive Solid Geometry, kurz *CSG*, ist ein auf der Mengentheorie aufbauendes Verfahren, mit dem man aus einfachen Grundkörpern, den sogenannten *Primitiven*, komplizierte Objekt zusammensetzen kann. Jede Form, und damit auch jedes Primitiv, wird als Punktmenge aufgefasst, die man invertieren und untereinander schneiden und vereinigen kann.

In Abbildung 2.16 befindet sich ein Beispiel für eine CSG-Operation. Die Fläche des kleinen Kreises wird auf dem Rand des großen Kreises platziert und invertiert, womit das „Innere“ des kleinen Kreises außen liegt. Anschließend erfolgt ein Schnitt, um die rechte Figur zu erhalten.

Als Primitiv kann jedes beliebige mathematische Gebilde verwendet werden, sofern es eine Punktmenge beschreibt, die von derselben Dimension ist wie der verwendete Raum. Gängige Primitive sind Kugeln, Zylinder, Ebenen (wobei eine Seite als „innen“ definiert wird), Tori, polynomielle implizite Funktionen, usw.

Der große Vorteil von CSG ist, dass intuitives Modellieren ermöglicht wird und die errungene Beschreibung des Körpers bei aller Präzision sehr kompakt ist. Nachteilig ist, daß die Kollisionserkennung zwar mathematisch einfach (Man muss nur die Schnittmenge bilden.), praktisch aber schwierig zu bewerkstelligen ist. Zudem arbeiten viele grafische System nicht mit Volumen, sondern mit Flächen, und hier insbesondere mit Polygonen. Aus einem Volumenmodell mit vielen geschwungenen Kanten eine Oberflächenbeschreibung mit Polygonen zu erzeugen, ist jedoch schwierig. Wegen des streng mathematischen Hintergrundes machen zahlreiche praktische Operationen Schwierigkeiten, wie z.B. das Abrunden von Ecken.

2.3.2 Implizite Funktionen

Implizite Funktionen sind Funktionen

$$f : R^3 \to R \tag{2.28}$$

Die Oberfläche bilden alle Punkte $\vec{x}$ mit $f(\vec{x}) = 0$. Falls $f(\vec{x}) < 0$, liegt $\vec{x}$ innen, falls $f(\vec{x}) > 0$, liegt $\vec{x}$ außen.

Als Funktion f kommt jeder Funktionstyp in Frage, besonders häufig anzutreffen sind jedoch Polynome.

Für eine gegebene Form eine implizite Funktion zu finden, die dieser Form nahe kommt, ist äußerst schwierig. Richtig nützlich werden implizite Funktionen für unser Anwendungsgebiet erst, wenn man sie als Primitive in CSG-Systemen verwendet.

2.3.3 Parametrische Funktionen

Parametrische Funktionen beschreiben Kurven und Flächen durch Punkte auf oder in der Nähe der Figur.

Zu diesem Typ zählen die Bézierkurve und der B-Spline.

2.3.3.1 Bézierkurven

Sind die Punkte $\vec{b_0}, \ldots, \vec{b_n}$ gegeben, dann ist die *Bézierkurve* gegeben durch

$$b(t) = \sum_{i=0}^{n} \vec{b_i} \cdot B_i^n(t) \ , \ 0 \le t \le 1 \tag{2.29}$$

wobei B_i^n die Bernsteinpolynome sind:

$$B_i^n(x) = \left(\begin{array}{c} n \\ i \end{array} \right) x^i (1-x)^{n-i} \tag{2.30}$$

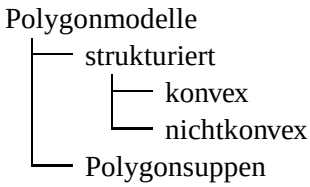


Abbildung 2.17: Klassifizierung von Polygonmodellen.

Die Bézierkurve endet in den Punkten $\vec{b_0}$ und $\vec{b_n}$ und liegt nicht notwendig auf den anderen. Das Verschieben eines Punktes wirkt sich auf die gesamte Kurve aus, so daß lokale Änderungen nicht möglich sind.

2.3.3.2 B-Splines

Sind die Punkte $\vec{b_0}, \ldots, \vec{b_m}$ gegeben, dann ist die *B-Spline-Kurve* gegeben durch

$$b(t) = \sum_{i=0}^{m} \vec{b_i} \cdot N_i^n(t) \ , \ x_0 \leq t \leq x_k \tag{2.31}$$

wobei N_i^n die B-Spline-Funktion zu einem Knotenvektor $\xi = (x_0, x_1, \ldots, x_k)$ ist:

$$N_i^0(x) = \begin{cases} 1, & \text{falls } x \in [x_i, x_{i+1}) \\ 0, & \text{sonst} \end{cases} \tag{2.32}$$

$$N_i^n(x) = \frac{x - x_i}{x_{i+n} - x_i} \cdot N_i^{n-1}(x) + \frac{x_{i+n+1} - x}{x_{i+n+1} - x_{i+1}} \cdot N_{i+1}^{n-1}(x) \tag{2.33}$$

$$0 \leq i \leq k-n-1, \ 1 \leq n \leq k-1, \ \frac{0}{0} := 0 \tag{2.34}$$

Die Beschaffenheit der B-Spline-Kurve ändert sich je nach Knotenvektor. Dieser wird jedoch so gewählt, daß die Kurve in den Punkten $\vec{b_0}$ und $\vec{b_m}$ endet und eine Änderung eines Punktes die Kurve nicht global, sondern nur in einem Abschnitt verschiebt.

2.3.4 Polygone

Polygonmodelle lassen sich grob wie in Abbildung 2.17 klassifizieren.

Strukturierte Polygonmodelle enthalten neben den reinen Eckpunktdaten auch Beziehungen zwischen Polygonen, wie z.B. Nachbarschaft. Konvexe Polygonmodelle haben gegenüber nichtkonvexen den Vorteil, dass sich viele Operationen schneller und/oder einfacher durchführen lassen. Dazu zählt die grafische Wiedergabe, und auch die Berechnung von „closest features“, d.h. von zwei Punkten, Kanten oder Flächen von Körpern, die sich von allen Elementen („features“)

der Körper am nächsten sind. Der hier nicht beschriebene am Mitsubishi Electric Research Laboratory entwickelte V-Clip-Algorithmus basiert auf der Ausnutzung der Konvexität. Polygonsuppen sind unstrukturierte Polygonmodelle in dem Sinne, dass nur die beteiligten Polygone, nicht aber Beziehungen zwischen diesen Polygonen gegeben sind.

Der besondere Vorteil von Polygonmodellen besteht in der hohen Verarbeitungsgeschwindigkeit sowohl für diverse Berechnungen als auch für die grafische Wiedergabe. Nachteilig ist, dass die Modellierung mit Polygonen ziemlich mühsam ist, besonders wenn man Krümmungen darstellen will. Polygone sind grundsätzlich eben. Krümmungen müssen mit einer großen Anzahl kleiner Polygone beschrieben werden. Ein weiteres Manko ist die geringe Verwendbarkeit in CSG-Systemen, weil Polygone keine Volumen, sondern Flächen beschreiben.

2.3.5 Fazit

Während CSG-Systeme eine intuitive Modellierung, parametrische Funktionen und weiche Formen erlauben, bestechen Polygone dagegen durch schnelle Berechnung und Darstellung.

Idealerweise würde man eine Form mit einem CSG-System unter Zuhilfenahmen von B-Splines modellieren und anschließend die resultierende Oberfläche in ein Polygonmodell konvertieren, damit es maschinell einfach weiterverarbeitet werden kann und die Darstellung unproblematisch ist.

2.4 Kollisionserkennung

Kollisionserkennung beschäftigt sich mit dem Problem, festzustellen, ob die Oberflächen zweier Körper sich schneiden.

Die hier im einzelnen vorgestellte Lösung arbeitet mit unstrukturierten Polygonmodellen (Polygonsuppen), was sicherstellt, dass alle Modelle verarbeitet werden können.

Das Problem mit Polygonen ist der Laufzeitzuwachs, wenn die Modelle größer werden, in dem Sinne, dass zusätzliche Polygone, auch bei einer Verfeinerung, hinzukommen. Besteht ein Körper aus M und der andere aus N Polygonen, dann gilt für die Laufzeit T des naiven Algorithmus, der alle Polygone gegeneinander auf Schnitt testet, $T \in O(M \cdot N)$.

Erste Entwürfe zur Beschleunigung sind Hüllkörperhierarchien aus achsenparallelen Quadern, Würfeln oder Kugeln. Andere Algorithmen verwenden BSP-Bäume, die den Raum sukzessive in zwei Teile spalten, bis nur noch ein Polygon in einem Segment verweilt.

Diese Verfahren funktionieren gut, solange die Objekte relativ weit voneinander entfernt sind. Kommen sie sich näher, werden noch engere Hüllkörper benötigt. Deshalb befassen wir uns hier mit OBBs, objektausgerichteten Hüllquadern (engl. Object Bounding Boxes), (vgl. auch [11], [12] und [13]).

2.4.1 Kosten der Kollisionserkennung

Als Kostenfunktion wird

$$T = N_i \cdot C_i + N_p \cdot C_p \tag{2.35}$$

verwendet, wobei die Symbole folgende Bedeutung haben:

C_p	Kosten eines Primitiventests
N_p	Anzahl der Primitiventests
C_i	Kosten eines Hüllkörpertests
N_i	Anzahl der Hüllkörpertests
T	Gesamtkosten der Kollisionserkennung

Auf die Parameter kann durch die Wahl der Hüllkörper (AABBs, Kugeln, OBBs) Einfluss genommen werden. Die Hüllkörpertypen werden weiter unten beschrieben.

Weniger einflussreich ist die Wahl der *hierarchischen Dekomposition*. Man kann hier *top-down* oder *bottom-up* vorgehen. Wenn man top-down vorgeht, geht man von allen Polygonen aus und teilt sie solange, bis nur noch Mengen mit je einem Polygon vorliegen. Im Falle bottom-up betrachtet man einzelne Polygone und vereinigt sie möglichst geschickt, bis alle in einer Menge liegen. Die jeweiligen Aufteilungs- oder Vereinigungswege bilden die hierarchische Aufteilung der Polygone.

Ferner muss man entscheiden, ob man die Polygone so aufteilen möchte, dass der Raum möglichst gleichgewichtig verteilt ist, oder ob die Polygone gleichgewichtig verteilt sind.

2.4.2 Achsenausgerichtete Hüllquader

AABBs sind Quader, die das Objekt verhüllen und deren Kanten parallel zu den Koordinatenachsen liegen. Ein Test auf Kollision zweier achsenausgerichteter Hüllquader A und B:

$$\forall\, i \in \{x,y,z\} : \neg\ (A_{i\max} < B_{i\min} \ \vee\ B_{i\max} < A_{i\min})\,. \tag{2.36}$$

Der Vorteil von AABBs ist, dass deren Konstruktion sehr einfach ist. Es sind nur die Extremwerte der Polygonpunkte in allen Dimensionen zu ermitteln. Zudem ist der Test auf Kollision sehr einfach (C_i klein).

Dem steht der Nachteil gegenüber, dass AABBs sehr groß werden können, insbesondere wenn ein längliches Objekt schräg im Raum liegt (N_i eher groß). Zudem erfordern Orientierungsänderungen des umhüllten Körpers eine Neuberechnung des Hüllquaders. Dem kann man entgegenwirken, wenn man statt Quadern Würfel verwendet, deren Kantenlänge dem Maximum der Distanzen aller Punkte zueinander entspricht. In dem Falle werden die Hüllkörper weitaus größer.

2.4.3 Hüllkugeln

Als Hüllkörper wird die kleinste Kugel gewählt, die alle Punkte in ihrem Inneren einschließen kann.

Ein Test auf Kollision zweier Kugeln A und B lautet:

$$\text{dist}\,(M_A, M_B) \leq r_A + r_B \tag{2.37}$$

, Wobei M_A und M_B die Mittelpunkte und r_A und r_B die Radien der Kugeln bezeichnen.

Hüllkugeln lassen sich mit moderatem Aufwand konstruieren und sind sehr schnell zu testen (C_i klein). Außerdem müssen sie bei einer Orientierungsänderung nicht neu berechnet werden. Nachteilig ist, dass Hüllkugeln i.A. noch größer sind als AABBs, so dass mit noch mehr Tests in den unteren Hierarchieebenen zu rechnen ist (N_i groß). Das schränkt die Benutzbarkeit bei nahe beieinander liegenden Körpern stark ein.

2.4.4 Objektausgerichtete Hüllquader

Als Hüllkörper wird ein Quader mit möglichst geringem Rauminhalt verwendet. Dazu muss er am zu umhüllenden Objekt selbst ausgerichtet sein, damit keine „toten Ecken“ entstehen.

Der Vorteil ist, dass man sehr eng anliegende Hüllkörper erhält (N_i klein). Daher eignen sich objektausgerichtete Hüllquader auch für Kollisionstests, bei denen die Kandidaten sich ständig relativ nahe sind. Ändert das Objekt seine Orientierung, muss der Quader nicht neu berechnet werden, sondern seine Orientierung kann einfach mitverändert werden. Nachteilig ist, dass die Konstruktion der Hüllquader ein kompliziertes Verfahren erfordern. Schlimm ist das nicht, denn wie oben bereits ausgeführt, muss die Berechnung nur einmal erfolgen. Sie muss erst dann wiederholt werden, wenn sich die geometrische Beschaffenheit des Körpers verändert. Ein weitere Nachteil ist, dass ein einzelner Kollisionstest komplizierter ist, als bei den einfachen Hüllkörpern (C_i eher groß).

2.4.4.1 Konstruktion

Zur Konstruktion eines objektausgerichteten Hüllquaders müssen alle Polygone des betreffenden Quaders trianguliert werden. Seien $\mathbf{p_i}, \mathbf{q_i}, \mathbf{r_i}$ die Eckpunkte dieser Dreiecke. Dann ist der Schwerpunkt μ des Objekts

$$\mu = \frac{1}{3 \cdot n} \cdot \sum_{i=1}^{n} \left(\mathbf{p_i} + \mathbf{q_i} + \mathbf{r_i}\right). \tag{2.38}$$

Sei nun $\bar{\mathbf{x}_\mathbf{i}} = \mathbf{x_i} - \mu$ und $\bar{\mathbf{x}_\mathbf{i}} = \begin{pmatrix} \bar{\mathbf{x}_\mathbf{i}}_1 \\ \bar{\mathbf{x}_\mathbf{i}}_2 \\ \bar{\mathbf{x}_\mathbf{i}}_3 \end{pmatrix}$, dann sei die *Kovarianzmatrix* C aller Punkte wie folgt konstruiert:

$$\forall\,(j,k) \in \{1,2,3\}^2 : C_{jk} = \frac{1}{3 \cdot n} \cdot \sum_{i=1}^{n} \left(\bar{\mathbf{p}_\mathbf{i}}_j \cdot \bar{\mathbf{p}_\mathbf{i}}_k + \bar{\mathbf{q}_\mathbf{i}}_j \cdot \bar{\mathbf{q}_\mathbf{i}}_k + \bar{\mathbf{r}_\mathbf{i}}_j \cdot \bar{\mathbf{r}_\mathbf{i}}_k\right). \tag{2.39}$$

Eigenvektoren symmetrischer Matrizen, wie C eine ist, sind paarweise orthogonal. Zwei davon zeigen außerdem in die Richtungen der maximalen und minimalen Varianz. Die Kanten des gesuchten Quaders sollen also mit den Eigenvektoren parallel sein. Die Ausmaße des Hüllquaders werden durch die in Richtung der Eigenvektoren am weitesten entfernt liegenden Punkte bestimmt.

Auf dem Weg zu einer guten Kovarianzmatrix gibt es jedoch zwei mögliche Störquellen:

1. Dichte Punktansammlungen im Innern des Körpers könnten die Ausrichtung beeinflussen. Dieses Problem kann man lösen, indem man nur die Punkte der konvexen Hülle berücksichtigt.

2. Nun können immer noch dichte Punktansammlungen auf der konvexen Hülle stören. Als Konsequenz müsste die konvexe Hülle gerastert werden, so dass die Rasterung feiner ist als die Punktansammlung. Wir erreichen diesen Effekt, indem wir die konvexe Hülle triangulieren und die Schwerpunkte der Dreiecke mit der Fläche des Dreiecks gewichten.

Die Fläche eines Dreiecks ist

$$A_i = \frac{1}{2} \cdot |(\mathbf{p_i} - \mathbf{q_i}) \times (\mathbf{p_i} - \mathbf{r_i})| \tag{2.40}$$

Die Fläche der konvexen Hülle ist dann

$$A_H = \sum_{i=1}^{m} A_i \tag{2.41}$$

Der Schwerpunkt eines Dreiecks ist

$$\mathbf{s_i} = \frac{1}{3} \cdot (\mathbf{p_i} + \mathbf{q_i} + \mathbf{r_i}) \tag{2.42}$$

Der Schwerpunkt der konvexen Hülle ist dann

$$\mathbf{s_H} = \frac{\sum_{i=1}^{m} A_i \cdot \mathbf{s_i}}{A_H} \tag{2.43}$$

Dadurch ergibt sich eine neue Kovarianzmatrix C, die für alle $(k,j) \in \{1,2,3\}^2$ folgende Form hat:

$$\begin{aligned} C_{jk} \quad = \quad & \sum_{i=1}^{m} \frac{A_i}{12 \cdot A_H} \cdot (9 \cdot \mathbf{s_i}_j \cdot \mathbf{s_i}_k + \\ & \mathbf{p_i}_j \cdot \mathbf{p_i}_k + \mathbf{q_i}_j \cdot \mathbf{q_i}_k + \mathbf{r_i}_j \cdot \mathbf{r_i}_k) - \mathbf{s_H}_j \cdot \mathbf{s_H}_k \end{aligned} \tag{2.44}$$

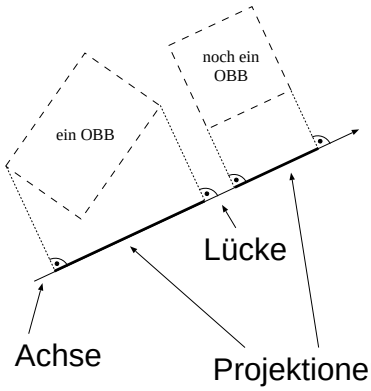


Abbildung 2.18: Prinzip der Trennungsachse.

Die weitere Vorgehensweise entspricht der bereits beschriebenen: Eigenvektoren als Basis nehmen und Extrema suchen.

2.4.4.2 Kollisionserkennung

Die naive Variante würde alle 12 Kanten eines Quaders mit den sechs Flächen des anderen auf Kollision testen, was wechselseitig 144 Kante/Fläche-Tests bedeutet. Aus Gründen der Geschwindigkeit ist dieser Algorithmus unbrauchbar. Es gibt andere Lösungen mit linearer Optimierung oder mit Entfernungsmessung, die mäßige Ergebnisse liefern.

Der hier vorgestellte Algorithmus basiert auf einer einfachen Idee: Projeziert man einen Quader orthogonal auf eine Gerade, so ergibt sich auf der Geraden ein Intervall. Schneiden sich die Intervalle zweier Quader nicht, dann schneiden sich auch die Quader selbst nicht. Abbildung 2.18 verdeutlicht den Zusammenhang. Aufgabe des Algorithmus des Algorithmus ist es, eine solche Gerade, die *Trennungsachse* genannt wird, zu finden.

Hier stellt sich die Frage, wo der Algorithmus nach Trennungsachsen zu suchen hat. Es ist bekannt, dass disjunkte konvexe Polyeder durch eine der folgenden Ebenen getrennt werden können:

- eine zu einer Polyederfläche parallele Ebene.
- eine Ebene, die parallel zu einer Ebene ist, die durch jeweils eine Kante beider Polyeder aufgespannt werden.

Eine Normale zu einer solchen trennenden Ebene muss eine Trennungsachse sein. Da eine Achse eine ganze Gruppe paralleler Ebenen erfasst, sind, weil jeder Quader drei Flächen- und Kantenorientierungen hat, folgende Tests ausreichend:

- 3 Tests über Flächen von Box *A*.

- 3 Tests über Flächen von Box B.
- $3 \cdot 3$ Tests über Kantenkombinationen.

Insgesamt sind das 15 Tests.

2.4.4.3 Testdurchführung

Ein einzelner Test mit einer Achse läuft im Wesentlichen folgendermaßen ab:

1. Die Strecke zwischen den Quaderzentren wird auf die Achse projeziert. Die Länge der projezierten Strecke ergibt den Wert d.
2. Es werden die Intervallradien berechnet, was die Werte r_A und r_B ergibt. Ein Intervall*durchmesser* ergibt sich durch die Projektion aller *verschiedenen* Kanten auf die Achse und die Addition der resultierenden Längen.
3. $d > r_A + r_b \Rightarrow$ Quader sind disjunkt.

$\mathbf{A_i}$, $i \in \{1,2,3\}$ seien die normierten Richtungsvektoren dreier verschieden orientierter Kanten des Quaders A. Mit den entsprechenden Kantenlängen a_i ergibt sich der Intervallradius auf einer Achse $\mathbf{L}$ zu

$$r_A = \frac{1}{2} \cdot \sum_{i=1}^{3} |a_i \cdot \mathbf{A_i} \bullet \mathbf{L}| \,. \tag{2.45}$$

Ist $\mathbf{T}$ der Differenzvektor der beiden Quaderzentren, dann sieht der gesamte Test *auf Nichtkollision* der beiden Quader A und B so aus:

$$|\mathbf{T} \bullet \mathbf{L}| > \frac{1}{2} \cdot \sum_{i=3}^{3} |a_i \cdot \mathbf{A_i} \bullet \mathbf{L}| + \frac{1}{2} \cdot \sum_{i=3}^{3} |b_i \cdot \mathbf{B_i} \bullet \mathbf{L}| \,. \tag{2.46}$$

Die zu testenden Achsen sind parallel zu Kanten oder Kreuzprodukte von Kanten der zu untersuchenden Quader. Daraus ergibt sich Potential für die Vereinfachung und damit Beschleunigung der Tests.

Weitere Optimierungen sind in den Spezialfällen eines zweidimensionalen Quaders (d.h. der Körper besteht aus einem oder aus mehreren parallelen Polygonen) und eines Quaders mit unendlichem Ausmaß möglich.

2.4.4.4 Erkennung von Kollisionen zwischen zwei Polygonsuppen

Algorithmus 2.1 heißt *testeKollision* und ist ein rekursives Verfahren, mit dem Polygonsuppen, die in binären Hüllkörperhierarchien organisiert sind, auf Kollision getestet werden können.

Die Funktion *polygone* gibt die Menge der Polygone innerhalb eines Hüllkörpers zurück. Die Funktion *trenneBox* gibt die in der Hierarchie direkt unterhalb angeordneten Hüllkörper zurück. *messung* soll ein Maß, z.B. den Rauminhalt,

```
Q := ∅
if A ∼_H B then
  if messung(A) > messung(B) and teilbar(A) then
    (A_1, A_2) := trenneBox(A)
    Q := Q ∪ testeKollision(A_1, A_2)
  else if teilbar(B) then
    (B_1, B_2) := trenneBox(B)
    Q := Q ∪ testeKollision(B_1, B_2)
  else
    for all (p, q) ∈ polygone(A) × polygone(B) do
      if p ∼_P q then
        Q := Q ∪ {schnittlinie(p, q)}
      end if
    end for
  end if
end if
return Q
```

Algorithmus 2.1: Dieser Algorithmus testet zwei Polygonsuppen, gegeben durch ihre Hüllkörper A und B, auf Kollisionen, gespeichert in der Menge der Kollisionsschnitte Q.

einer Box berechnen. Dieses Maß wird dazu verwendet, zu entscheiden, welcher Hüllkörper „geöffnet“ wird.

Die Relation $\sim_H$ bedeutet Schnitt zwischen Hüllkörpern, $\sim_P$ bedeutet Schnitt zwischen Polygonen. Die Funktion *schnittlinie* berechnet die Schnittgerade zwischen zwei Polygonen. Das Prädikat *teilbar* gibt an, ob ein Hüllkörper in zwei kleinere Hüllkörper teilbar, d.h. ob er kein Blatt des Hierarchiebaums ist.

2.5 Dynamik

Die Lehre der Dynamik befasst man sich mit Bewegungen in einem System von Körpern. Es gibt translatorische und rotatorische Bewegungsarten. Außerdem werden infinitesimale Bewegungen betrachtet, um unter Aufstellung von Energiebilanzen letztendlich die Bewegungsgleichungen in Form eines Differentialgleichungssystems aufzustellen (vgl. auch [14], [15], [16] und [17]).

2.5.1 Jakobi-Matrizen - Berechnungsverfahren zu infiniter Bewegung

Im Folgenden werden nun nur kleine (infinitesimale) Auslenkungen betrachtet und ein Verfahren zu deren Berechnung vorgestellt.

2.5.1.1 Ein einführendes Beispiel

Für zwei aneinanderhängende stabförmige Gelenke, im Folgenden mit l_1, l_2 bezeichnet, kann man angeben:

$$p_x = l_1 \cdot \cos\Theta_1 + l_2 \cdot \cos(\Theta_1 + \Theta_2) = x(\Theta_1, \Theta_2), \tag{2.47}$$

$$p_y = l_1 \cdot \sin\Theta_1 + l_2 \cdot \sin(\Theta_1 + \Theta_2) = y(\Theta_1, \Theta_2), \tag{2.48}$$

wobei Θ_1 der absolute Winkel zwischen dem 1.Gelenk und der x-Achse und Θ_2 der relative Winkel zwischen beiden Gelenken ist. Die Differentiale zur Beschreibung kleiner Änderungen von Θ_1 und Θ_2 auf x und y sind:

$$dx = \frac{\varsigma x}{\varsigma \Theta_1} \cdot d\Theta_1 + \frac{\varsigma x}{\varsigma \Theta_2} \cdot d\Theta_2, \tag{2.49}$$

$$dy = \frac{\varsigma y}{\varsigma \Theta_1} \cdot d\Theta_1 + \frac{\varsigma y}{\varsigma \Theta_2} \cdot d\Theta_2. \tag{2.50}$$

Man definiert ferner:

$$\mathbf{J} := \begin{pmatrix} \frac{\varsigma x}{\varsigma \Theta_1} & \frac{\varsigma x}{\varsigma \Theta_2} \\ \frac{\varsigma y}{\varsigma \Theta_1} & \frac{\varsigma y}{\varsigma \Theta_2} \end{pmatrix}. \tag{2.51}$$

Dann gilt:

$$\begin{pmatrix} dx \\ dy \end{pmatrix} = \mathbf{J} \cdot \begin{pmatrix} d\Theta_1 \\ d\Theta_2 \end{pmatrix}. \tag{2.52}$$

Mit dem obigen Beispiel gilt dann hier(!):

$$\mathbf{J} = \begin{pmatrix} -l_1 \cdot \sin\Theta_1 - l_2 \cdot \sin(\Theta_1 + \Theta_2) & -l_2 \cdot \sin(\Theta_1 + \Theta_2) \\ l_1 \cdot \cos\Theta_1 + l_2 \cdot \cos(\Theta_1 + \Theta_2) & l_2 \cdot \cos(\Theta_1 + \Theta_2) \end{pmatrix}. \tag{2.53}$$

Teilt man $\binom{dx}{dy}$ durch das Zeitinkrement dt erhält man für $\dot{l} = \frac{\binom{dx}{dy}}{dt}$:

$$\dot{l} = \mathbf{J} \cdot \dot{\Theta} \tag{2.54}$$

und erhält somit den Zusammenhang zwischen der kartesischen Geschwindigkeit $\dot{l}$ und der Achsgeschwindigkeit $\dot{\Theta}$.

2.5.1.2 Definitionsgleichung der Jakobimatrix

Die Jakobi-Matrix **J** ist für den allgemeinen Fall durch die folgende Gleichung definiert:

$$\dot{l} = \mathbf{J} \cdot \dot{q} \tag{2.55}$$

Dabei ist $\dot{l} = (\dot{x}, \dot{y}, \dot{z}, \dot{\Phi}_x, \dot{\Phi}_y, \dot{\Phi}_z)$ die zeitliche Ableitung des Lagevektors in kartesischen Koordinaten und $\dot{q} = (\dot{q}_1 \ldots \dot{q}_n)$ die zeitliche Ableitung des Lagevektors in verallgemeinerten Roboterkoordinaten ($\vec{q}_i = \vec{\Theta}_i$ für ein Drehgelenk, $\vec{q}_i = \vec{d}_i$ für ein Schubgelenk). Wobei $\dot{\Phi}_x, \dot{\Phi}_y, \dot{\Phi}_z$ die zeitlichen Ableitungen der Drehwinkel um die x/y/z-Achsen sind.

J hat also die folgende Struktur:

$$\mathbf{J} = \begin{pmatrix} \frac{\varsigma x}{\varsigma q_1} & \cdots & \frac{\varsigma x}{\varsigma q_n} \\ \vdots & \cdots & \vdots \\ \frac{\varsigma \Phi_z}{\varsigma q_1} & \cdots & \frac{\varsigma \Phi_z}{\varsigma q_n} \end{pmatrix}; \tag{2.56}$$

nun benötigen wir noch die Lageänderung eines Punktes aufgrund einer infinitesimalen Rotation.

2.5.1.3 Infinitesimale Rotationen

Nähert man die Sinus- bzw. Cosinus-Glieder der Rotationsmatrizen unter Benutzung von $(d\Phi_x, d\Phi_y, d\Phi_z) \approx (0,0,0) \Rightarrow \cos 0 \approx 1, \sin 0 \approx d\Phi$, so gilt für kleine Drehungen $d\Phi_x, d\Phi_y, d\Phi_z$ ungefähr:

$$Rot(\vec{x}, d\Phi_x) := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -d\Phi_x \\ 0 & d\Phi_x & 1 \end{pmatrix}, \tag{2.57}$$

$$Rot(\vec{y}, d\Phi_y) := \begin{pmatrix} 1 & 0 & d\Phi_y \\ 0 & 1 & 0 \\ -d\Phi_x & 0 & 1 \end{pmatrix}, \tag{2.58}$$

$$Rot(\vec{z}, d\Phi_z) := \begin{pmatrix} 1 & -d\Phi_z & 0 \\ d\Phi_z & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.59}$$

Für die Drehung um die drei Raumrichtungen gilt dann (die Reihenfolge ist egal, da die Multiplikation dieser Matrizen kommutativ ist):

$$\begin{aligned} &Rot(\vec{x}, d\Phi_x) \cdot Rot(\vec{y}, d\Phi_y) \cdot Rot(\vec{z}, d\Phi_z) \\ &= \begin{pmatrix} 1 & -d\Phi_z & d\Phi_y \\ d\Phi_z & 1 & -d\Phi_x \\ -d\Phi_y & d\Phi_x & 1 \end{pmatrix} =: Rot(d\vec{\Phi}) \end{aligned} \tag{2.60}$$

mit

$$d\vec{\Phi} = \begin{pmatrix} d\Phi_x \\ d\Phi_y \\ d\Phi_z \end{pmatrix}. \tag{2.61}$$

Für die Verschiebung $d\vec{a}$ eines Vektors

$$\vec{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \tag{2.62}$$

durch eine infinitesimale Drehung $Rot(d\vec{\Phi})$ gilt somit

$$d\vec{a} = \vec{a}' - \vec{a} = Rot(d\vec{\Phi}) \cdot \vec{a} - \vec{a} = \begin{pmatrix} a_z \cdot d\Phi_y - a_y \cdot d\Phi_z \\ a_x \cdot d\Phi_z - a_z \cdot d\Phi_x \\ a_y \cdot d\Phi_x - a_x \cdot d\Phi_y \end{pmatrix}. \tag{2.63}$$

2.5.1.4 Jakobi-Matrix eines allgemeinen Roboters

Die Jakobi-Matrix für einen allgemeinen Roboter mit n Schub- und/oder Drehgelenken sieht wie folgt aus. Alle Gelenke werden in diesen (in der Praxis riesigen) Matrizen erfasst.

$$\dot{l} = \begin{pmatrix} \dot{p} \\ \Phi \end{pmatrix} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \Phi_x \\ \Phi_y \\ \Phi_z \end{pmatrix}, \quad \dot{q} = \begin{pmatrix} q_1 \\ \vdots \\ q_n \end{pmatrix}, \tag{2.64}$$

$$\mathbf{J} = \begin{pmatrix} J_{1x} & \dots & J_{nx} \\ J_{1y} & \dots & J_{ny} \\ J_{1z} & \dots & J_{nz} \\ J_{1\Phi_x} & \dots & J_{n\Phi_x} \\ J_{1\Phi_y} & \dots & J_{n\Phi_y} \\ J_{1\Phi_z} & \dots & J_{n\Phi_z} \end{pmatrix} = \begin{pmatrix} J_{1p} & \dots & J_{np} \\ J_{1\Phi} & \dots & J_{n\Phi} \end{pmatrix} = J_1 \dots J_n. \tag{2.65}$$

Dabei bezieht sich J_{ip} auf das i-te Schubgelenk und $J_{i\Phi}$ auf das i-te Drehgelenk.

2.5.1.5 Bedeutung der Jakobi-Determinanten

Die Jakobi-Matrix kann auch zur Bestimmung der Gelenkgeschwindigkeiten aus vorgegebenen kartesischen Geschwindigkeiten genutzt werden.

$$\dot{l} = \mathbf{J} \cdot \dot{q}, \tag{2.66}$$

ist **J** invertierbar, so folgt hieraus:

$$\dot{q} = \mathbf{J}^{-1} \cdot \dot{l}. \tag{2.67}$$

Vorraussetzungen der Invertierbarkeit:

- **J** ist quadratisch
- $det(\mathbf{J}) \neq 0$

Allgemein gilt (und ist in der Rechnung zu berücksichtigen) für eine Jakobi-Determinante mit k Nullstellen, die Anzahl der singulären Stellen der Kinematik ist gleich k.

2.5.2 Lagrange-Verfahren

Dieses Verfahren kann man hervorragend automatisieren, was auch in jeder Dynamiksimulationsbibliothek geschieht und wird deshalb nur aus Vollständigkeitsgründen mit aufgeführt.

2.5.2.1 Der Begriff der Lagrangeschen Koordinaten

Jeder einzelne Freiheitsgrad eines Roboters stellt eine *Lagrangesche Koordinate* dar, die im weiteren mit q_i bezeichnet wird. In der Regel hat man es mit Dreh- und Schubachsen zu tun. Den Lagrangeschen Koordinaten zugeordnet sind *Lagrangesche Kräfte*, im weiteren mit Q_i bezeichnet, die zur Bewegung des Roboters notwendig sind. Beispielsweise gehört zu einer Schubachse eine Kraft, die eine translatorische Bewegung bewirkt; zu einer Drehachse gehört das entsprechende Moment für rotatorische Bewegungen.

Lager: In ihnen sind die Achsen des Roboters aufgehängt. Die Lager sorgen dafür, dass sich der Roboter nur gemäß seiner Kinematik bewegen kann, stellen damit also technische Realisierungen der Bindungen dar. Anders ausgedrückt sorgt die *Bindung* dafür, dass die Bewegungsfähigkeit eines Körpers auf seine Freiheitsgrade eingeschränkt wird. Die auftretende Lagerbelastung ist identisch mit den Zwangskräften. Diese Kräfte erzwingen somit eine den Bindungen entsprechende Bewegung. Da *Zwangskräfte* von der Bewegung des Körpers und von äußeren auf den Körper einwirkenden Kräften bzw. Momenten abhängen, sind sie *nicht von vorneherein bekannt*. Dagegen sind eingeprägte Kräfte bekannt, da sie durch bekannt Mechanismen wie Federn, Dämpfer, Reibung und Motoren entstehen.

Potential: Falls für einen Mechanismus gilt, dass mechanische Energie erhalten bleibt, lässt sich das Potential V finden, aus dem sich die zugehörige Kraft oder das Momentum durch Bildung der negativen Gradienten leicht berechnen lässt.

$$\vec{F} = \begin{pmatrix} \frac{\varsigma V}{\varsigma x} \\ \frac{\varsigma V}{\varsigma y} \\ \frac{\varsigma V}{\varsigma z} \end{pmatrix}. \tag{2.68}$$

Modelliert man die Mechanik von Robotern als System starrer Körper, so ist es möglich, Methoden zu verwenden, die geradlinig zum Ergebnis führen. Da Roboter eine Kinematik besitzen, die nicht explizit von der Zeit abhängt, sind auch die Bindungen des Roboters zeitinvariant. Da sich die Zwangskräfte so ergeben,

dass sich alle starren Körper der Roboterkinematik entsprechend bewegen, sind folglich auch diese nicht von der Zeit, sondern nur von den Freiheitsgraden, deren Geschwindigkeiten und Beschleunigungen abhängig. D.h. es gilt für die Zwangskräfte:

$$\vec{F}_Z = \vec{F}_Z(q, \dot{q}, \ddot{q}); \tag{2.69}$$

auf den Körper wirkende eingeprägte Kräfte (Gewichtskraft, Reibung, Federkräfte von Riemen oder Kräfte an der Motorabtriebsseite) sind in der Regel ebenfalls nicht explizit von der Zeit abhängig. Bis auf Reibungskräfte und Motorantriebskräfte liegt allen eingeprägten Kräften ein Mechanismus zugrunde, für den *Energieerhaltung im mechanischen System* gilt. Folglich lässt sich für diese Klasse eingeprägter Kräfte ein Potential angeben, mit dem es sich sehr kompakt rechnen lässt.

Die *Methode von Lagrange* macht sich diese Kompaktheit zunutze und bietet einen eleganten Weg, ein Modell für die Robotermechanik zu berechnen. Um Zwangskräfte nicht weiter betrachten zu müssen, ist es sinnvoll, Methoden zu verwenden, die von der im Roboter gespeicherten Enegie ausgehen. Die Methode von Lagrange ist ein solcher Ansatz. Er führt auf den gleichen Satz von Bewegungsgleichungen, nutzt aber dabei aus, dass die Arbeit der Zwangskräfte verschwindet. Um dies plausibel zu machen, betrachtet man zunächst in einem Manipulator mit n Freiheitsgraden und k Körpern den i-ten Teilkörper und dessen Ortsvektor $\vec{r}_i = \vec{r}_i(\vec{q}(t))$.

Bewegt man den Teilkörper i um ein infinitesimal kleines Stück $\delta\vec{r}_i$, das mit den Bindungen verträglich ist, so lässt sich diese Verschiebung schreiben als

$$d\vec{r}_i = \frac{\varsigma\vec{r}_i}{\varsigma\vec{q}_1} \cdot d\vec{q}_1 + \cdots + \frac{\varsigma\vec{r}_i}{\varsigma\vec{q}_n} \cdot d\vec{q}_n, \tag{2.70}$$

wobei $d\vec{q} = [dq_1, \ldots, dq_n]^T$ die Verrückung in den Freiheitsgraden ist.

Naturgemäß stehen die Zwangskräfte $\vec{F}_{Zi}$ auf ihren Bindungen senkrecht. Es folgt sofort, dass die Arbeit der Zwangskräfte verschwindet, wenn man einen einzelnen starren Körper betrachtet, da alle Zwangskräfte $\vec{F}_{Zi}$ stets auf den Verschiebungen $d\vec{r}_i$ senkrecht stehen, so dass keine Arbeit $dW = \vec{F}_{Zi} \cdot d\vec{r}_i = 0$ von ihnen verrichtet wird. Im Falle eines Mehrkörpersystems mit k Körpern und n Freiheitsgraden gilt:

$$dW = \sum_{j=1}^{n} \sum_{i=1}^{k} \vec{F}_{Zj} \cdot \frac{\varsigma\vec{r}_i}{\varsigma q_i} \cdot dq_i = 0 \tag{2.71}$$

Die eingeprägten Kräfte $\vec{F}_e$ teilen sich in konservative und nicht-konservative Kräfte auf. *Konservative Kräfte* werden durch Mechanismen hervorgerufen, die *energieerhalten* sind (z.B. Federn oder Gravitionsfelder). Konservative Kräfte $\vec{F}_k$ besitzen stets ein Potential.

Mechanismen, die nicht energieerhaltend sind (z.B. Reibung), verursachen dagegen Kräfte, die kein Potential besitzen und heißen *nicht-konservative Kräfte.*

Als zweiter Energiespeicher neben Potentialen muss die kinetische Energie in der Bilanzgleichung aufgeführt werden. Analog zur kinetischen Energie beim Massepunkt

$$T = \frac{1}{2}mv^2 \tag{2.72}$$

ergibt sich beim starren Körper

$$T = \frac{1}{2}\int\int_m\int v^2 dm \tag{2.73}$$

mit Geschwindigkeit v als Volumenintegral über den Körper m. Wobei die Integration wie bei der Berechnung des Trägheitstensors als Dreifachintegral über den ganzen Körper V zu verstehen ist.

Mit V als potentieller und T als kinetischer Energie des Manipulators ergeben sich schließlich die Lagrangeschen Bewegungsgleichungen durch Differentation der Bewegungsfunktion.

$$L = T - V \quad \Rightarrow \quad Q_i = \frac{d}{dt}\left(\frac{\varsigma L}{\varsigma \dot{q}_i}\right) - \frac{\varsigma L}{\varsigma q_i} \tag{2.74}$$

Dabei ist festzuhalten, dass die Lagrangeschen Kräfte Q_i in Richtung der Freiheitsgrade wirken. Die Q_i sind Momente bei Rotationsachsen und Kräfte bei Schubachsen.

2.5.2.2 Systematische Vorgehensweise bei Manipulatoren

Die Anwendung des Formalismus von Lagrange auf einen Roboter aus k starren Körpern oder Massenpunkten vollzieht sich in 3 Stufen:

I. Berechnung der potentiellen und kinetischen Energie für jeden Teilkörper

Für jeden Körper i werden sämtliche vorhandenen Potentiale (Gravitationspotential, Federpotential etc.) als Funktion der Freiheitsgrade $\vec{q}$ bestimmt und aufsummiert:

$$V_i(\vec{q}) = V_{iGrav}(\vec{q}) + V_{iFeder}(\vec{q}) + \ldots \tag{2.75}$$

Betrachtet man die Translation eines Körpers mit Schwerpunktgeschwindigkeit v_{Si} und gleichzeitige Rotation des Körpers i mit $\vec{\omega}$, so ergibt sich als kinetische Energie:

$$T_i = \underbrace{\frac{1}{2}\vec{v}_{Si}^T m_i \vec{v}_{Si}}_{Translation} + \underbrace{\frac{1}{2}\vec{\omega}_i^T \vec{\theta}_{Si}\vec{\omega}_i}_{Rotation} \quad . \tag{2.76}$$

$\vec{\omega}_i$ ist der Vektor aller Kreisfrequenzen zu den einzelnen Achsen des Körpers i, und $\vec{\theta}_{Si}$ ist der Trägheitstensor des Körpers bezüglich seines Schwerpunktes S_i. All das ist analog zum skalaren Fall: Kinetische Energie $= 1/2mV^2$ und Potentielle Energie $= 1/2\theta\omega^2$.

Wird die Bewegung bezüglich eines körperfesten Punktes A_i notiert, so ergibt sich aus Gleichung 2.76 unter der Verwendung von $\vec{r}_{ASi}$ als Verbindungsvektor vom körperfesten Bezugspunkt zum Schwerpunkt:

$$\vec{v}_{Si} = \vec{v}_{Ai} + \vec{r}_{ASi} \times \vec{\omega}_i \tag{2.77}$$

$$T_i = \frac{1}{2} m \left|\vec{v}_{Ai}\right|^2 + m_i \vec{v}_{Ai} \left(\vec{\omega}_i \times \vec{r}_{ASi}\right) + \frac{1}{2} \vec{\omega}_i^T \vec{\theta}_{Ai} \vec{\omega}_i \tag{2.78}$$

mit

$$\vec{v}_{Ai}(q, \dot{q}) = \frac{d\vec{r}_{Ai}}{dt}. \tag{2.79}$$

Dabei wurden die Geschwindigkeit $\vec{v}_{Si}$ und Trägheitstensor $\vec{\theta}_{Si}$ auf den neuen Bezugspunkt A_i transformiert ($\rightarrow \vec{v}_{Ai}, \vec{\theta}_{Ai}$). Die Berechnung der kinetischen Energie vollzieht sich nach folgendem Schema:

1. Wahl eines Bezugspunktes A_i. Im Falle der Rotation um einen körperfesten Punkt bietet sich dieser als Bezugspunkt an. In den meisten anderen Fällen liegt der Schwerpunkt günstig.

2. Wahl eines oder mehrerer geeigneter Koordinatensysteme. Zur Berechnung der kinetischen Energie des Teilkörpers i kann jeder Summand, d.h. jeder Anteil der kinetischen Energie, in einem günstig gewählten Koordinatensystem berechnet werden. Dabei muss nur darauf geachtet werden, dass alle Vektoren, die zur Berechnung eines Summanden verwendet werden, konsistent im gleichen Koordinatensystem notiert sind.

3. Der Ortsvektor $\vec{r}_{Ai} = \vec{r}_{Ai}(\vec{q})$ ist nach der Zeit zu differenzieren, um $\vec{v}_{Ai}(q, \dot{q})$ zu erhalten. $\vec{\omega}_{Ai}(q, \dot{q})$ kann dagegen meist sofort abgelesen werden.

4. Der Vektor $\vec{r}_{ASi}(\vec{q})$ und der Trägheitstensor θ_{Ai} bezüglich des Punktes A_i sind zu ermitteln. Durch Einsetzen obiger Größen in Gleichung 2.76 oder 2.78 kann die kinetische Energie $T_i(q, \dot{q})$ berechnet werden.

Nochmal: Für jeden Teilkörper i werden im *1. Schritt* potentielle Energie V_i und kinetische Energie T_i berechnet.

II. Aufsummierung der Energien

Für den Roboter als mechanisches Gesamtsystem werden die kinetischen und potentiellen Energien der Teilkörper aufsummiert.

$$T = \sum_{i=1}^{k} T_i \quad V = \sum_{i=1}^{k} V_i. \tag{2.80}$$

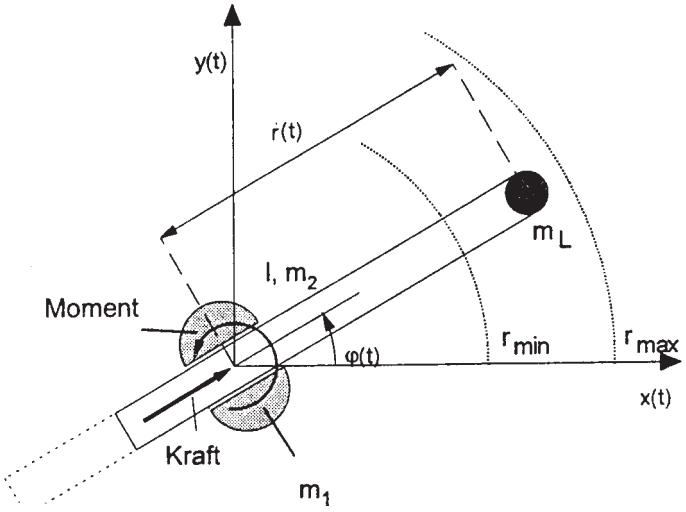


Abbildung 2.19: Zylindersymmetrische Drehsäule

III. Differentation gemäß Gleichung 2.74

Bei Manipulatoren liegt in aller Regel der Fall geschwindigkeitsunabhängiger Potentiale vor. Das kann dann wie folgt gechrieben werden:

$$Q_i = \frac{d}{dt}\left(\frac{\varsigma T}{\varsigma \dot{q}_i}\right) - \frac{\varsigma T}{\varsigma \dot{q}_i} + \frac{\varsigma V}{\varsigma \dot{q}_i}, \tag{2.81}$$

da

$$\frac{\varsigma V}{\varsigma \dot{q}_i} = 0, \qquad \dot{q}_i = \frac{\varsigma q_i}{\varsigma t}. \tag{2.82}$$

2.5.2.3 Beispiel zur Methode von Lagrange

Jeder Teilkörper wird zunächst isoliert betrachtet. Für eine zylindersymmetrische Drehsäule (vgl. Abbildung 2.19) gelten folgende Energiebilanzen.

Kinetische Energie:

$$T = \frac{1}{2} m_1 v^2 = \frac{1}{2} m_1 \frac{(r\dot{\varphi})^2}{2} \tag{2.83}$$

$$r\dot{\varphi} = r\frac{d\varphi}{dt} = \frac{dl}{dt} = v \tag{2.84}$$

dl ist das Bogenelement bei Drehung um $d\varphi$. Aus $T = \frac{1}{2}\theta\omega^2$ folgt wieder $\theta = mr^2$, $\omega = \frac{d\varphi}{dt}$, $\varphi = \omega t$.

Die restlichen Zwischenrechnungen werden ausgespart, es muss nur konsequent mit obigen Formeln gearbeitet werden. Einmal für den Ausleger und dann für die Lastmasse.

Auf der Basis der obigen Teilergebnisse lässt sich die Lagrange-Funktion des Roboters so notieren:

$$L = T - V = \frac{1}{2}\frac{m_1 r_1^2}{2}\dot{\varphi}^2 + \frac{1}{2}m_L\Big((r\dot{\varphi})^2 + \dot{r}^2\Big) + \frac{1}{2}m_2\Big((r\dot{\varphi})^2 + \dot{r}^2\Big) - m_2 r\frac{1}{2}\dot{\varphi}^2 + \frac{1}{6}l^2\dot{\varphi}^2 - const. \qquad (2.85)$$

Durch die Anwendung der Lagrange-Formel für den rotatorischen und den translatorischen Freiheitsgrad, des Roboters kann man nun aus dem Ausdruck für die Gesamtenergie die Bewegungsgleichungen bestimmen. Es gilt dann für das Moment $M(t)$

$$M_{Antrieb} = \frac{d}{dt}\left(\frac{\varsigma L}{\varsigma\dot{\varphi}}\right) - \frac{\varsigma L}{\varsigma\varphi}$$
$$= \left(\frac{m_1 r_1^2}{2} + \frac{m_2 l^2}{3} + (m_2 + m_L)r^2 - m_2 lr\right)\ddot{\varphi} + \Big(2(m_2 + m_L)r - m_2 l\Big)\dot{r}\dot{\varphi} \qquad (2.86)$$

und die Kraft $K(t)$

$$F_{Antrieb} = \frac{d}{dt}\left(\frac{\varsigma L}{\varsigma\dot{r}}\right) - \frac{\varsigma L}{\varsigma r}$$
$$= (m_2 + m_L)\ddot{r} + \Big(-(m_2 + m_L)r + \frac{m_2 l}{2}\Big)\dot{\varphi}^2. \qquad (2.87)$$

2.5.3 Newton-Euler-Verfahren

Ausgangspunkt bei der Methode von Newton und Euler ist der Erhaltungssatz für Impuls und Drehimpuls beim einzelnen Massenpunkt oder starren Körper. Zunächst werden für jeden einzelnen Körper die angreifenden Kräfte und Momente summiert. Zwangskräfte und Zwangsmomente treten in den Gleichungen genauso auf wie eingeprägte Kräfte bzw. Momente. Für jeden Massenpunkt ergeben sich bei dreidimensionaler Betrachtung drei, für jeden Starrkörper sechs Bilanzgleichungen für Impuls bzw. Drehimpuls des Massenpunkts oder Körpers.

Das Auflösen dieses Gleichungssystems nach den eingeprägten Kräften und Momenten, die von außen vorgegeben werden, führt auf die Bewegungsgleichungen des Roboters in einer für dynamische Modelle brauchbaren Form. Weiter ist es möglich, Zwangskräfte und -momente, die bei einer gegebenen Bewegung des Roboters entstehen, aus dem Gleichungssystem zu berechnen.

Im ersten Schritt wird der Roboter gedanklich in seine Teilkörper zerlegt, wobei der Anwender entscheidet, welche Körper zusammengefasst oder gar als Massenpunkte modelliert werden sollen. Eine Skizze der freigeschnittenen Körper erleichtert den Überblick und bietet eine gute Grundlage für die folgenden Schritte.

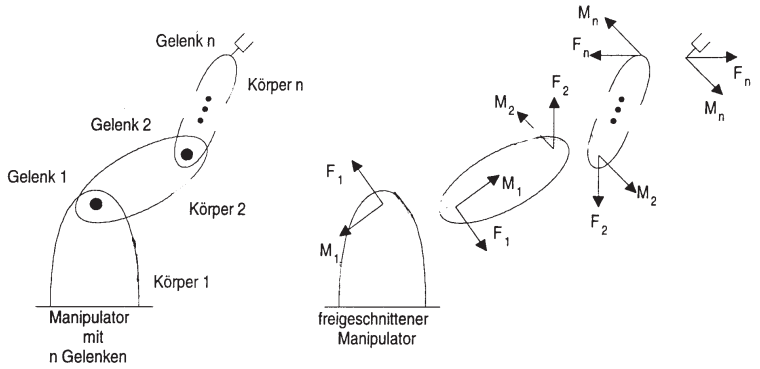


Abbildung 2.20: Freischneiden eines Körpers

Nach dem Freischneiden der Teilkörper werden Kräfte und Momente für jeden Teilkörper aufsummiert und im nächsten Schritt den zeitlichen Änderungen von Impuls und Drehimpuls gleichgesetzt. Dabei ist auf die Wahl der Bezugspunkte zur Berechnung der Impulse und Drehimpulse zu achten. Für die folgenden Ausführungen wird vorausgesetzt, dass sich die Momentensumme und auch die Formulierung der Impuls- bzw. Drehimpulsänderung immer auf den Schwerpunkt des freigeschnittenen Teilkörpers bezieht.

2.5.3.1 Freischneiden

Die Teilkörper des Roboters werden einzeln aufgezeichnet und an den Verbindungsstücken die Kräfte und Momente eingetragen (vgl. Abbildung 2.20). Im allgemeinen Fall wirkt an jedem Verbindungsstück je ein dreidimensionaler Vektor für Kräfte und Momente; wichtig ist es, hierbei keine Komponente zu vergessen. Da in den Verbindungsstücken Kräfte und Momente nur vom Körper n auf den Körper $n+1$ übertragen werden, ist es notwendig, diese in einem bestimmten Gelenk für den Körper $n+1$ entgegengesetzt zu den entsprechenden Kräften und Momenten im Körper n anzufragen.

2.5.3.2 Berechnung der angreifenden Kräfte und Momente

Für jeden Teilkörper i wird die *Kräftesumme* $\vec{F}_{iges} = \sum \vec{F}_e + \vec{F}_z$ der an ihm eingeprägten Kräfte und der Zwangskräfte gebildet. Zur Berechnung der Momentsumme ist es wichtig festzuhalten, ob der Körper sich um einen ortsfesten Punkt dreht.

Nur in diesem Fall wird die Momentensumme bezüglich des ortsfesten Punktes gebildet; sonst ist stets der Schwerpunkt der Bezugspunkt zur Bildung der Momentensumme. Die Momentensumme setzt sich zusammen aus eingeprägten Momenten und Zwangsmomenten sowie aus den Momenten, die von Zwangskräften oder

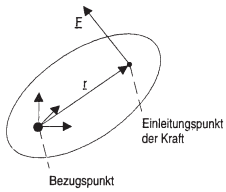


Abbildung 2.21: Relevante Punkte für die Kraftbilanz

eingeprägten Kräften hervorgerufen werden.

$$\vec{M}_{iges} = \sum_{Körper} \vec{M}_e + \vec{M}_z + \vec{r} \times \vec{F}_e + \vec{r} \times \vec{F} za. \tag{2.88}$$

$\vec{r}$ ist dabei der Vektor vom Bezugspunkt für die Momentenbildung (Schwerpunkt oder ortsfester beliebiger Punkt des Körpers) zum Einleitungspunkt der Kraft (vgl. auch Abbildung 2.21).

2.5.3.3 Aufstellung von Impuls- und Drehimpulsbilanz

Die Berechnung von Impuls und Drehimpuls des Starrkörpers erfolgt konsistent zur Berechnung der Kräfte und Momente im Koordinatensystem des Körpers *i*.

Zur systematischen Berechnung des Impulses von Teilkörper *i* benötigt man den Ortsvektor $\vec{x}_i = \vec{x}_i(\vec{q})$ des *Schwerpunktes* in Abhängigkeit von den Freiheitsgraden des Manipulators. Da für die Bilanzierung von Kräften die zeitliche Änderung des Impulses eine Rolle spielt, ist es notwendig zu bilden:

$$\vec{F}_{iges} = \dot{p}_i = m_i \ddot{x}_i = \ddot{x}_i(q, \dot{q}) \tag{2.89}$$

Impuls: $p = mv = m\dot{x}$, daraus folgt die Ableitung

$$\dot{p} = \underbrace{\dot{m}v}_{=0, m=const} + m\dot{v} = m\dot{v} = m\ddot{x} \tag{2.90}$$

Drehimpuls des Teilkörpers *i* bei Drehung um den ortsfesten Punkt A_i, Winkelgeschwindigkeit ω_i gegenüber dem Inertialsystem

$$\vec{L}_i = \vec{\theta}_{Ai} \vec{\omega}_i \tag{2.91}$$

Dabei ist $\vec{\theta}_{Ai}$ der Trägheitstensor des Körpers *i* bezogen auf den Punkt A_i. Man erhält also drei Differentialgleichungen 2. Ordnung für den *i*-ten Teilkörper.

Fehlt ein ortsfester Punkt, um den sich der Körper dreht, so wird der Schwerpunkt als Bezugspunkt genommen. Der Drehimpuls des Körpers bezüglich des Schwerpunktes ist (vgl. Abbildung 2.22):

$$\vec{L}_{Si} = \vec{\theta}_{Si} \vec{\omega}_i \tag{2.92}$$

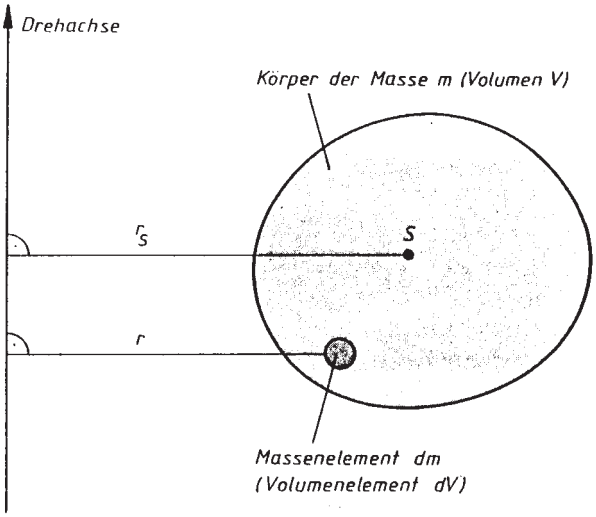


Abbildung 2.22: Graphisches Vorgehen für Schwerpunktberechnungen

Dabei ist $\vec{\theta}_{Si}$ der Trägheitstensor des Körpers i bezogen auf dessen Schwerpunkt S_i.

Der Trägheitstensor bezogen auf den Schwerpunkt ergibt sich durch Integration über den gesamten Körper i:

$$\vec{\theta}_{Si} = \begin{pmatrix} \int (y^2+z^2)dm & -\int yxdm & -\int zxdm \\ -\int xydm & \int (x^2+z^2)dm & -\int zydm \\ -\int xzdm & -\int yzdm & \int (x^2+y^2)dm \end{pmatrix} \tag{2.93}$$

mit den Schwerpunktskoordinaten einens homogenen räumlichen Körpers:

$$x_S = \frac{1}{V}\int xdV, \quad y_S = \frac{1}{V}\int ydV, \quad z_S = \frac{1}{V}\int zdV \tag{2.94}$$

und

$$\vec{r}_S = (x_S, y_S, z_S)^T, \quad mr_S = \int\limits_{(m)} rdm \tag{2.95}$$

Benötigt man den Trägheitstensor des Körpers in Bezug auf einen beliebigen Punkt A_i, so ist der Trägheitstensor des Schwerpunkts auf diesen neuen Punkt A_i zu transformieren. Die Transformation von $\vec{\theta}_{Si}$ auf $\vec{\theta}_{Ai}$ vollzieht sich mit:

$$\vec{\theta}_{Ai} = \vec{\theta}_{Si} + m_i \begin{pmatrix} y_{AS}^2+z_{AS}^2 & -y_{AS}x_{AS} & -z_{AS}x_{AS} \\ -x_{AS}y_{AS} & x_{AS}^2+z_{AS}^2 & -z_{AS}y_{AS} \\ -x_{AS}z_{AS} & -y_{AS}z_{AS} & x_{AS}^2+y_{AS}^2 \end{pmatrix} \tag{2.96}$$

In Analogie zur Impulsbilanz erhält man aus der Drehimpulsbilanz die fehlenden 3 Differentialgleichungen 2. Ordnung für den i-ten Teilkörper $\vec{\theta}_i\dot{\omega}_i = \vec{M}_{iges}$ unter Benutzung des Satzes von Steiner.

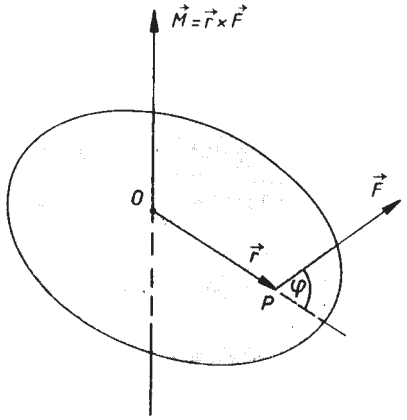


Abbildung 2.23: Drehmoment und Kraft

2.5.3.4 Drehmoment (Moment einer Kraft)

Drehmomente sind vektorielle Größen, die bei der Behandlung statischer Systeme von großer Bedeutung sind.

Wir betrachten einen starren Körper in Form einer Kreisscheibe, der um seine Symmetrieachse drehbar gelagert ist wie in Abbildung 2.23 zu sehen ist.

Massenträgheitsmoment eines homogenen Körpers Man bezeichnet eine geometrische Figur als homogen, wenn seine Dichte stets konstant ist.

Definitionsformel:

$$J = p \int\int\int_V r_A^2 dV \tag{2.97}$$

Dabei ist r_A der Abstand des Volumenelemens dV von der Bezugsachse A und p die konstante Dichte des Körpers.

Der Satz von Steiner sagt: Für eine zur Schwerpunktachse S im Abstand d parallel verlaufende Bezugsachse A gilt:

$$J_A = J_S + md^2 \tag{2.98}$$

J_S ist das Massenträgheitsmoment bezüglich der Schwerpunktachse.

2.5.3.5 Auflösen des Gleichungssytems

Für jeden Körper ergeben sich aus Impuls- und Drallbilanz 6 Differentialgleichungen 2. Ordnung. Die Bindungen, denen die Teilkörper des Roboters unterworfen sind, führen zu einer deutlichen Reduzierung der Freiheitsgrade für die Teilkörper, da die eingeprägten Kräfte und Momente unter Einhaltung der Bindungen die Bewegung des Roboters vollständig bestimmen. Durch Elimination der Zwangskräfte

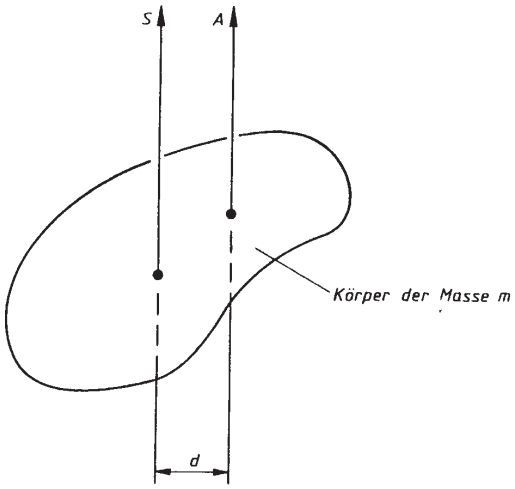


Abbildung 2.24: Schwerpunktachse und Bezugsachse

erhält man genau so viele Differentialgleichungen 2. Ordnung, wie dieses System Freiheitsgrade hat.

Dieses Gleichungssystem ist linear in den eingeprägten Kräften und Momenten, sowie in den zweiten Ableitungen der Freiheitsgrade $\ddot{q}$; es lässt sich also leicht wahlweise nach einem von beiden auflösen.

2.5.3.6 Klärung des Begriffes Inertialsystem

Das *Inertialsystem* ist ein Trägheitssystem, von wo aus Bewegung relativ zum eigenen Zustand beobachtet wird.

Die *Trägheit eines Körpers* ist der Kraftaufwand, der notwendig ist, um den Körper von außen zu bewegen.

Also: Das Inertialsystem entspricht dem betrachteten Körper als System.

Ein Beispiel zur Modellierung via Newton-Euler Verfahren wird hier bewusst nicht weiter aufgeführt, da man selbstverständlich die gleichen Ergebnisse erhält wie mit dem Lagrange Verfahren. Generell kann man aber noch folgendes festhalten:

Bei der Berechnung der Bewegungsgleichungen mit Newton-Euler, wird von der Roboterhand bis zur Basis vorgegangen. Dies ist auch für kompliziertere Robotermodelle sinnvoll. Zusammen mit Standardvorschriften zur Aufstellung der Gleichungen für die einzelnen Roboterachsen wird in der Literatur das „Rekursive Newton-Euler Verfahren“ häufig zitiert, das im Prinzip nur eine exakte Formalisierung der oben anschaulich hergeleiteten Vorgehensweise darstellt und eine besonders gute (algorithmisch) per Computer auswertbare Form der Gleichungen liefert.

2.5.4 Abschließender Vergleich der beiden Methoden

Das Verfahren von Newton-Euler erlaubt von vornherein die Berücksichtigung aller Kräfte, sofern sie sich als Gleichung festhalten lassen. Allgemeine Kraftgesetze, wie nichtlineare Reibung oder Dämpfung ins Modell aufzunehmen, erfordert dabei keinen nennenswerten Mehraufwand. Da bei dieser Methode alle Zwangskräfte explizit in den Bilanzgleichungen für Impuls und Drehimpuls auftreten, kann ohne Probleme danach aufgelöst werden. Entsprechend groß ist dagegen der Aufwand, die Zwangskräfte zu eliminieren, um ein Gleichungssystem in den Kräften und Momenten zu erhalten, die von der Robotersteuerung vorgegeben werden.

Basierend auf der Berechnung kinetischer und potentieller Energien erhält man beim Verfahren von Lagrange durch Anwendung eines einfachen Formalismus die Bewegungsgleichungen des Roboters. Die Vorgehensweise ist geradlinig und erfordert relativ wenig Aufwand. Elastische Elemente, wie Riemen oder Getriebeelastizitäten können durch zusätzliche Freiheitsgrade und eine damit verbundene Erhöhung der Systemordnung problemlos modelliert werden. Dabei können auch Kräfte und Momente im Modell vorgesehen werden, für deren Wirkprinzip die mechanische Energie nicht erhalten bleibt, z.B. Reibung oder Dämpfung.

2.6 Numerische Methoden

Wie bereits in den vorherigen Kapiteln zu bemerken war, handelt es sich bei den Bewegungsgleichungen um Differentialgleichungen. Es ist bekannt, solche Gleichungen mit Hilfe der Integrationsrechnung zu lösen. Da nun aber eine computergerechte Lösungsmethode gesucht wird, müssen diese Gleichungssysteme in eine numerisch auswertbare Form transformiert werden. Der in numerischen Rechnungen stets anfallende algorithmische Fehler (d.h. Diskretisierungsfehler und Rundungsfehler) muss a priori abgeschätzt werden um vorab zu analysieren ob der Algorithmus hinsichtlich der angesprochenen Fehler und der Laufzeit effizient ist.

Bei der vorzunehmenden Transformation der Bewegungsgleichungen behelfen wir uns mit bereits vorhandenem Wissen aus der mathematischen Darstellungsform von Systemgleichungen (bzw. Zustandsgleichungen) aus der Regelungstechnik. Das heißt also: Übergang in die Numerik via standardisierter Form der Regelungstechnik. Dazu erstmal ein erweitertes Beispiel (vgl. Abbildung 2.25).

2.6.1 Beispiel für den Übergang in die Numerik

Durch das Lagrange Verfahren wurden beim Beispielroboter folgende Gleichungen hergeleitet:

$$\underbrace{\begin{pmatrix} F_{Antrieb} \\ M_{Antrieb} \end{pmatrix}}_{\vec{Q}} = \underbrace{\begin{pmatrix} m_2 + {}_m L & 0 \\ 0 & \frac{m_1 r_1^2}{2} + \frac{m_2 l^2}{3} + (m_2 + m_L) r^2 - m_2 l r \end{pmatrix}}_{\vec{\Theta}(\vec{q})} \underbrace{\begin{pmatrix} \ddot{r} \\ \ddot{\varphi} \end{pmatrix}}_{\ddot{q}}$$

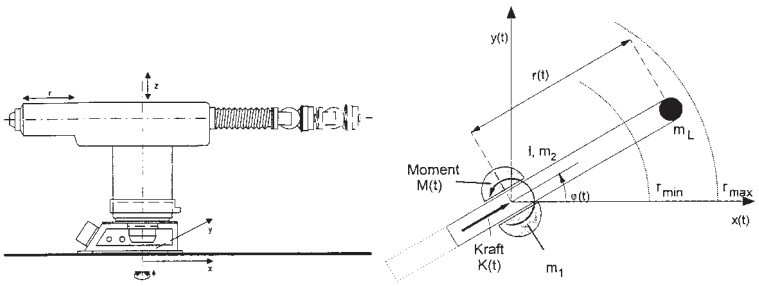


Abbildung 2.25: Industrieroboter, Seitenansicht und Vogelperspektive

$$+\underbrace{\begin{pmatrix}\left(-(m_2+m_L)r+\frac{m_2 l}{2}\right)\dot{\varphi}^2\\ \left(2(m_2+m_L)r-m_2 l\right)\dot{r}\dot{\varphi}\end{pmatrix}}_{CORZEN(q,\dot{q})} \tag{2.99}$$

Man erhält also allgemein einen Satz Bewegungsgleichungen (hier oben 2 Differentialgleichungen 2. Ordnung), sogar mit const. Koeffizienten, d.h. vom Typ: $\ddot{y}+a_1\dot{y}+a_0 y=u(t)$, wobei $y=y(t)$, $a_2=1$.

Nun erfolgt eine Erweiterung auf einen 3-achsigen Roboter mit zylindrischem Arbeitsraum. Das führt zu 3 Differentialgleichungen, da nun auch eine Kraftkomponente in z-Richtung, also in Richtung der Höhe existiert.

Durch die Zylindersymmetrie haben sich die Bezeichnungen nun geändert: $F_{Antrieb} \to F_r$ (in radiale Richtung wirkende Kraft) und $M_{Antrieb} \to M_\varphi$ (in azimuthaler (längs) Richtung wirkendes Drehmoment). Die Kraft der Höhenkomponente ist hier F_z. Die Bewegungsgleichungen des Roboters lauten dann ($g \approx 9,81\frac{m}{s^2}$ ist die Fallbeschleunigung):

$$F_z=(m_1+m_2+m_L)\ddot{z}+(m_1+m_2+m_L)g \tag{2.100}$$

$$F_r=(m_2+m_L)\ddot{r}-\left((m_2+m_L)r-\frac{m_2 l}{2}\right)\dot{\varphi}^2 \tag{2.101}$$

$$M_\varphi=\left(\frac{m_1 r_1^2}{2}+\frac{m_2 l^2}{3}+(m_2+m_L)r^2-m_2 l r\right)\ddot{\varphi}+(2(m_2+m_L)r-m_2 l)\dot{r}\dot{\varphi}. \tag{2.102}$$

Für den Übergang in die Numerik benutzen wir die standardisierte Darstellung der Bewegungsgleichungen für die Zustandsraumbeschreibung für Robotermodelle, die üblicherweise in der Regelung benutzt werden.

Man Unterscheidet Eingangs-, Ausgangs- und Zustandsgrößen des Robotergesamtsystems, die durch die Bewegungsgleichungen miteinander verkoppelt sind.

Eingangsgrößen des Robotermodells sind die von außen aufgeschalteten Motorströme, die innerhalb ihrer Grenzen frei eingestellt werden können. Als Ausgangsgrößen sind die Lagrangeschen Koordinaten des mechanischen Aufbaus von Interesse, die bei Bedarf entsprechend der Kinematik des Roboters zu transformieren sind, um beispielsweise die Position der Hand zu ermitteln. Zustandsgrößen sind in der Regel die Lagrangeschen Koordinaten und deren zeitliche Ableitungen. Das System hat die Zustandsdarstellung:

$$\dot{x} = \vec{A}(\vec{x}) + \vec{B}(\vec{x})\vec{u} \qquad \vec{y} = \vec{C}(\vec{x}). \tag{2.103}$$

Dabei ist $\vec{u}$ der *Eingangsvektor*, $\vec{x}$ der *Vektor der Zustandsgrößen* und $\vec{y}$ der *Ausgangsvektor* des Systems. Anhand der Zustandsgrößen ist es möglich, aus den Bewegungsgleichungen die Zustandsdarstellung relativ geradlinig abzuleiten. Anhand des obigen Roboters soll die Vorgehensweise verdeutlicht werden: Mit der Auswahl der Gelenkpositionen und -geschwindigkeiten

$$x_1(t) = r(t) \qquad x_2(t) = \dot{r}(t) \qquad x_3(t) = \varphi(t) \tag{2.104}$$

$$x_4(t) = \dot{\varphi}(t) \qquad x_5(t) = h(t) \qquad x_6(t) = \dot{h}(t) \tag{2.105}$$

als Zustandsgrößen und

$$u_1 = F_r(t) \qquad u_2 = M_\varphi(t) \qquad u_3 = F_z(t) \tag{2.106}$$

als Eingangsgrößen lauten die Bewegungsgleichungen des Roboters, wenn sie nach den Zustandsgrößen aufgelöst sind (zur Vereinfachung der Schreibweise wird ein Faktor $K(x_1)$ eingeführt):

$$K(x_1) = \frac{m_1 r_1^2}{2} + \frac{m_2 l^2}{3} + (m_2 + m_L)x_1^2 - m_2 l x_1 \tag{2.107}$$

$$\dot{x}_1 = x_2 = \dot{r}(t) \tag{2.108}$$

$$\dot{x}_2 = \frac{1}{m_2 + m_L}\left(\left((m_2 + m_L)x_1 - \frac{m_2 l}{2}\right)x_4^2 + u_1\right) \tag{2.109}$$

$$\dot{x}_3 = x_4 = \dot{\varphi}(t) \tag{2.110}$$

$$\dot{x}_4 = \frac{1}{K(x_1)}\left((-2x_1(m_2 + m_L) + m_2 l)x_2 x_4 + u_2\right) \tag{2.111}$$

$$\dot{x}_5 = x_6 = \dot{h}(t) \tag{2.112}$$

$$\dot{x}_6 = g + \frac{1}{m_1 + m_2 + m_L} u_3. \tag{2.113}$$

Damit ist dann

$$\dot{x} = \vec{A}(\vec{x}) + \vec{B}(\vec{x})\vec{u} = \vec{f}(\vec{x}, \vec{u}) \qquad \vec{y} = \vec{C}(\vec{x}) \tag{2.114}$$

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \end{pmatrix} = \begin{pmatrix} x_2 \\ x_1 x_4^2 - \frac{m_2 l}{m_2 + m_L} x_4^2 \\ x_4 \\ \frac{1}{K(x_1)}(-2x_1(m_2 + m_L) + m_2 l) x_2 x_4 \\ x_6 g \end{pmatrix} \tag{2.115}$$

$$+ \begin{pmatrix} 0 & 0 & 0 \\ \frac{1}{m_2 + m_L} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \frac{1}{K(x_1} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \frac{1}{m_1 + m_2 + m_L} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}. \tag{2.116}$$

Als Ausgangsgrößen des Systems sind die Lagrangeschen Koordinaten r, φ, z von Interesse; der Ausgangsvektor ergibt sich also sehr einfach zu (vgl. Zylinderkoordinaten):

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} \tag{2.117}$$

$$\Rightarrow \qquad y_1 = x_1 = r_t \tag{2.118}$$

$$y_2 = x_3 = \varphi_t \tag{2.119}$$

$$y_3 = x_5 = h_t. \tag{2.120}$$

In $\vec{y} = \vec{C}(\vec{x})$ ist $\vec{C}$ eine konstante Matrix. Durch die geschickt gewählte Substitution haben wir aus einem Differentialgleichungssystem 2. Ordnung der Dimension $n = 3$ ein Differentialgleichungssystem 1.Ordnung der Dimension $n = 6$ gemacht. Man bezeichnet dies als die sogenannte *Mathematische Ordnungsreduktion*.

2.6.2 Numerische Integration des Differentialgleichungssystems

Wir benutzen das mathematische Prinzip der Anfangswertproblembetrachtung.

$$\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \vdots \\ \dot{x}_6 \end{pmatrix} = \vec{f}(t, \vec{x}(t), \vec{u}(t)) = \begin{pmatrix} f_1(\vec{x}, \vec{u}) \\ \vdots \\ f_6(\vec{x}, \vec{u}) \end{pmatrix} \tag{2.121}$$

mit

$$\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_6 \end{pmatrix}, \qquad \vec{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_3 \end{pmatrix} \tag{2.122}$$

und den Anfangswerten

$$\vec{x}_0 = \vec{x}(t_0) = \begin{pmatrix} x_1(t_0) \\ \vdots \\ x_6(t_0) \end{pmatrix} \tag{2.123}$$

Da f linear ist, handelt es sich um eine lineare DGL (Differentialgleichung). Nun integrieren wir zur Lösung des Differentialgleichungssystems:

$$\vec{x}(t) = \vec{x}(t_0) + \int_{t_0}^{t} \vec{f}(\acute{t}, \vec{x}(\acute{t}), \vec{u}(\acute{t}))\, d\acute{t} \tag{2.124}$$

numerische Diskretisierung der Zeit: $t_k := t_0 + kh$; h ist die Schrittweite und $k = 0, \ldots, n$

$$\Rightarrow \vec{x}(t_{k+1}) = \underbrace{\vec{x}(t_0) + \int_{t_0}^{t_k} \vec{f}(\vec{x}, \vec{u})\, d\acute{t}}_{\vec{x}(t_k)} + \int_{t_k}^{t_{k+1}} \vec{f}(\vec{x}, \vec{u})\, d\acute{t} \tag{2.125}$$

$$= \vec{x}(t_k) + \int_{t_k}^{t_{k+1}} \vec{f}(\vec{x}, \vec{u})\, d\acute{t} \tag{2.126}$$

$$\Leftrightarrow \frac{\vec{x}(t_{k+1}) - \vec{x}(t_k)}{h} = \underbrace{\frac{1}{h} \int_{t_k}^{t_{k+1}} \vec{f}(\vec{x}, \vec{u})\, d\acute{t}}_{= \vec{f}_h(\vec{x}, \vec{u}) + \vec{T}_h(\vec{x}, \vec{u})}. \tag{2.127}$$

Dabei ist $\vec{f}_h$ die Inkrementvektorfunktion und $\vec{T}_h$ der lokale Abschnittsfehler. Das Problem ist nun die Wahl der Inkrementvektorfunktion $\vec{f}_h$. Für die Schrittweite gilt (mit n als Anzahl der Stützpunkte): $h = \frac{t_n - t_0}{n}$.

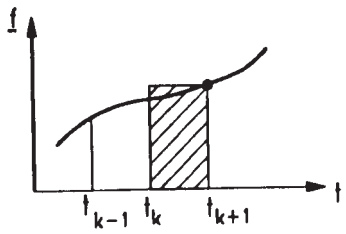


Abbildung 2.26: Euler-Rückwärtsintegration

2.6.3 Euler-Vorwärts-Verfahren

Dieses Verfahren wird aufgrund seiner Einfachheit am häufigsten eingesetzt, wenn man mit relativ groben Werten bereits zufrieden ist. Das Prinzip: Der Ausgangspunkt ist P_0, wobei $P_0 = P_0(t_0, \vec{f}(\vec{x}(t_0, \vec{u}(t_0)))) = \vec{x}_0 = \vec{x}(t_0)$.

Tangente an $\vec{f}$ in P_0. Auswertung der Tangente in P_1, d.h. Bestimmung von $\vec{x}_1$, wobei $\vec{x}_1 \neq \vec{x}(t_1)$! Hier ist also der 1.Diskretisierungsfehler, der sich in den nächsten Punkten vom lokalen Fehler zum globalen Fehler fortpflanzt. Die näherungsweise Berechnung der Lösungskurve $\vec{f} = \vec{f}(x)$ in P_2 erfolgt analog. Der Punkt P_1 ist daher der Ausgangspunkt. Die Lösungskurve wird dann im Intervall $t_1 \leq t \leq t_2$ durch eine Gerade ersetzt, die durch Punkt P_1 verläuft und dieselbe Steigung m_1 besitzt wie die Kurventagente der Lösungskurve durch P_1 an dieser Stelle, usw. für alle t_i

$$\Rightarrow \vec{x}_{k+1} = \vec{x}_k + h \cdot \vec{f}(\vec{x}_k, \vec{u}_k) \tag{2.128}$$

wobei

$$\vec{x}_{k+1} \approx \vec{x}(t_{k+1}) \qquad \vec{x}_k \approx \vec{x}(t_k) \qquad \vec{u}_k \approx \vec{u}(t_k) \tag{2.129}$$

Das ist das sogenannte *Euler-Vorwärts-Polygonzugverfahren*, ein explizites Verfahren mit (hier) der Inkrementfuntion $\vec{f}_h(\vec{x}, \vec{u}) := \vec{f}(\vec{x}_k, \vec{u}_k)$.

2.6.4 Euler-Rückwärts-Verfahren

Als Alternative setzt man auch $\vec{f}_h := \vec{f}(\vec{x}_{k+1}, \vec{u}_{k+1})$, das ist das sogenannte *Euler-Rückwärts-Polygonzugverfahren*, ein implizites Verfahren (siehe auch Abbildung 2.26).

$$\vec{x}_{k+1} = \vec{x}_k + \underbrace{h \cdot \vec{f}(\vec{x}_{k+1}, \vec{u}_{k+1})}_{\textit{Approximationsrechteck}} \tag{2.130}$$

Das Approximationsrechteck hat die Breite h und die Höhe des aktuellen Funktionsvektors $\vec{f}$. Vorweg sei gesagt: Der Stabilitätsbereich vom impliziten Verfahren ist i.a. größer als vom expliziten Verfahren, d.h. $\vec{x}_{k+1} \to \vec{0}$ für einen größeren Bereich von Anfangswerte $\vec{x}_0$ bzw. Störauslenkungen.

2.6.5 Runge-Kutta-Verfahren

Dieses Verfahren hat zwar eine höhere Rechenzeit, liefert aber dafür wesentlich besserer Werte als Euler-Vorwärts. Die Rechenvorschrift für das Verfahren 4.Ordnung ist:

$$\begin{aligned} k_1 &= f(x_k, u_k) \\ k_2 &= f(x_k + \frac{1}{2}hk_1, u_k + \frac{1}{2}) \\ k_3 &= f(x_k + \frac{1}{2}hk_2, u_k + \frac{1}{2}) \\ k_4 &= f(x_k + hk_3, u_k + 1) \\ x_{k+1} &= x_k + h(k_1 + 2k_2 + 2k_3 + k_4)/6, \end{aligned} \tag{2.131}$$

wobei es sich hier bei $k_1, \ldots, k_4, f, x_i, u_i$ weiterhin um Vektoren handelt. Es fließen nun auch mittlere unterschiedlich gewichtet Steigerungswerte ein.

Als Verallgemeinerung die Rechenvorschrift für Runge-Kutta-Verfahren m-ter Ordnung:

$$\begin{aligned} k_1 &= f(x_k, u_k) \\ k_2 &= f(x_k + ha_{21}k_1, u(t_k + b_2h)) \\ &\vdots \\ k_m &= f(x_k + h(a_{m1}k + \cdots + a_{m,m-1}k_{m-1}), u(t_k + b_mh)) \\ x_{k+1} &= x_k + h(c_1k_1 + c_2k_2 + \cdots + c_mk_m). \end{aligned} \tag{2.132}$$

2.6.6 Sonderfall linearer Differentialgleichungen

Dieser Ansatz ist für unsere Zwecke generell verwendbar, denn mathematisch gesehen rechnen wir ohnehin nur mit linearen Differentialgleichungen, da alle bisher aufgeführten Rekursionsgleichungen für numerische Integration rein lineare Operationen beinhalten (also Summierung und gewichtete Summen).

$$\dot{x} = \underbrace{\mathbf{A}\vec{x} + \mathbf{B}\vec{u}}_{\vec{f}(\vec{x},\vec{u})}, \qquad \vec{x}(0) = x_0. \tag{2.133}$$

Die zugehörige analytische Lösung lautet (siehe Trajektoriengleichung aus der Regelungssystemanalysis):

$$\vec{x}(t) = e^{\mathbf{A}t}\vec{x}_0 + \int_{t_0}^{t} e^{\mathbf{A}(t-t')}\mathbf{B}\vec{u}(t')dt'. \tag{2.134}$$

Die Transitionsmatrix ist dann die Taylorreihe

$$e^{\mathbf{A}t} = \sum_{n=0}^{\infty} \frac{\mathbf{A}^n t^n}{n!} = \mathbf{E} + \frac{\mathbf{A}t}{1!} + \frac{\mathbf{A}^2t^2}{2!} + \ldots \tag{2.135}$$

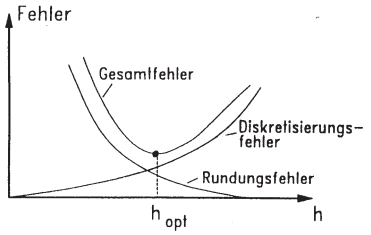


Abbildung 2.27: Fehlerarten in der Numerik

Für den diskreten Zeitpunkt $t_{k+1} = t_{k+h}$ folgt:

$$\vec{x}(t_{k+1}) = e^{\mathbf{A}h} \underbrace{\left(e^{\mathbf{A}t_k}\vec{x}_0 + \int_{t_0}^{t_k} e^{\mathbf{A}(t_k - t')}\mathbf{B}\vec{u}(t')dr \right)}_{\vec{x}(t_k)} + \int_{t_k}^{t_{k+1}} e^{\mathbf{A}(t_{k+1}-r)}\mathbf{B}\vec{u}(r)dt' \quad (2.136)$$

$$\Rightarrow \vec{x}(t_{k+1}) = e^{\mathbf{A}h}\vec{x}(t_k) + \int_{t_k}^{t_{k+1}} e^{\mathbf{A}(t_{k+1}-t')}\mathbf{B}\vec{u}(t')dt'. \quad (2.137)$$

Mit der Annahme $\vec{u} = const$ auf $[t_k, t_{k+1}]$

$$\int_{t_k}^{t_{k+1}} e^{\mathbf{A}(t_{k+1}-t')}\mathbf{B}\vec{u}(t')dt' = \mathbf{A}^{-1}(e^{\mathbf{A}h} - \mathbf{E}) \quad (2.138)$$

$$\Rightarrow \vec{x}_{k+1} = e^{\mathbf{A}h}\vec{x}_k + \mathbf{A}^{-1}(e^{\mathbf{A}h} - \mathbf{E})\mathbf{B}\vec{u}_k. \quad (2.139)$$

Das ist das sogenannte explizite Einschrittverfahren. Für die Berechnung von $e^{\mathbf{A}h}$ und $\mathbf{A}^{-1}(e^{\mathbf{A}h} - \mathbf{E})\mathbf{B}$, zur Integration der Bewegungsgleichungen, gibt es viele numerische Verfahren zur schnellen Matrixmultiplikation etc. unter Ausnutzung von Symmetrien innerhalb der Matrizen.

2.6.7 Allgemeine Fehlerbetrachtung

Ein Problem bei allen Integrationsverfahren ist die Wahl einer geeigneten Schrittweite *h*. Es gibt generell zwei Fehlerursachen (vgl. Abbildung 2.27):

- Diskretisierungsfehler: Verfahrensfehler bei Integration.
- Rundungsfehler: Folge beschränkter Rechengenauigkeit.

Je kleiner die Schrittweite ist, desto höher ist der Rechenaufwand, aber auch die Genauigkeit.

2.6.8 Konsistenzbedingung

Konsistenz heißt, dass der Gesamtdiskretisierungsfehler gegen Null strebt. Das Einschrittverfahren heißt konsistent mit dem Anfangswertproblem

$$\dot{x} = \vec{f}(\vec{x},\vec{u}), \qquad \vec{x}(t_0) = x_0, \tag{2.140}$$

wenn für die Schrittweite $h \to 0$ gilt:

$$\sum_{i=0}^{n} h_i |\vec{T}_h(\vec{x},\vec{u})| \to 0. \tag{2.141}$$

$\vec{T}_h$, der lokale Abschneidefehler, ist:

$$\vec{T}_h = \frac{1}{h}(\vec{x}(t+h) - \vec{x}(t)) - \vec{f}_h(x,u). \tag{2.142}$$

$\vec{x}(t)$ sei die exakte Lösung des AWP. Das Verfahren besitzt dann die *Konsistenzordnung p*, falls (Maß für den lokalen Fehler)

$$|\vec{T}_h| \leq k \cdot h^p. \tag{2.143}$$

Maßzahl für die Genauigkeit einer Rekursionsformel ist die Anzahl der übereinstimmenden Summanden der Taylorentwicklungen von $\vec{x}(t_{k+1})$ und Näherung (durch Rekursion) $\vec{x}_{k+1}$. Die Forderung ist $\vec{x}(t_j) = \vec{x}_j, j = (k,k-1,\ldots,k-n+1)$, d.h. die numerische Näherungsrekursionsformel liefert ab einem bestimmten n, $j = k-n+1$ konsistente Werte.

Als Beispiel das Eulerverfahren: $\vec{x}_{k+1} = \vec{x}_k + h\vec{f}(\vec{x}_k,\vec{u}_k)$. Taylorentwicklung an der Stelle t_k als Entwicklungszentrum.

$$\vec{x}(t_{k+1}) = \vec{x}(t_k) + \frac{\dot{x}(t_k)}{1!}\underbrace{(t_{k+1}-t_k)}_{=h} + \frac{\ddot{x}(t_k)}{2!}\underbrace{(t_{k+1}-t_k)^2}_{=h^2} + \ldots \tag{2.144}$$

mit

$$\frac{d\vec{x}}{dt} = \dot{x} = \vec{f}(t,\vec{x}(t),\vec{u}(t)) \Rightarrow \ddot{x} = grad(\vec{f})\frac{d}{dt}\begin{pmatrix} t \\ \vec{x} \end{pmatrix} = \frac{\varsigma\vec{f}}{\varsigma t} + \frac{\varsigma\vec{f}}{\varsigma\vec{x}}\dot{x}. \tag{2.145}$$

Unter der Bedingung, dass $\vec{X}_k = \vec{x}(t_k)$ und $\vec{f}(\vec{x}_k,\vec{u}_k) = \vec{f}(\vec{x}(t_k),\vec{u}(t_k))$ fordern wir, dass

$$\vec{x}(t_{k+1}) = \vec{x}(t_k) + h\vec{f} + \frac{h^2}{2!}\left(\frac{\varsigma\vec{f}}{\varsigma t} + \frac{\varsigma\vec{f}}{\varsigma\vec{x}}\vec{f} + \ldots\right). \tag{2.146}$$

Vergleich mit Näherung $\vec{x}_{k+1}$:

$$\vec{x}(t_{k+1}) - \vec{x}_{k+1} = \frac{h^2}{2!}(\ldots) = O(h^2). \tag{2.147}$$

Also ist die Fehlerordnung h^2, deshalb hat das Verfahren die Ordnung 1.

Aus den o.g. Berechnungen folgt die Definition für den Lokalen Fehler, dieser berechnet sich durch

$$\underbrace{\vec{x}(t_k)}_{exakterWert} - \underbrace{\vec{x}_k}_{Näherung} . \tag{2.148}$$

Die Inkrementfunktion $\vec{f}_h$ erfüllt bezüglich $\vec{x}$ eine Lipschitzbedingung

$$|\vec{f}_h(\vec{x}_1,\vec{u}) - f_h(\vec{x}_2,\vec{u})| \leq L|\vec{x}_1 - \vec{x}_2|, \qquad L > 0. \tag{2.149}$$

Man kann mit Hilfe geometrischer Reihen beweisen, dass die Ordnung des globalen Fehlers gleich der Konsistenzordnung ist. Ausserdem ist der globale Fehler gleich der Summe der lokalen Fehler.

2.6.9 Günstige Schrittweite

Aus Empirie folgt $h \cdot L \approx 0,1, L > 0$ Lipschitzkonstante. L stellt eine obere Schranke für die auf den Funktionswert bezogene Änderung der Ableitung dar:

$$\left|\frac{\varsigma\vec{f}}{\varsigma\vec{x}}\right| \leq L, \tag{2.150}$$

wobei $\vec{x}$ anfangs noch unbekannt ist. Dieser ist dann aufgrund der Anfangswerte $\vec{x}_0 = \vec{x}(t_0)$ und des ersten Näherungsschrittes abzuschätzen. Das Maß für den Fehler der Näherung ist die Differenz der mit einfacher Schrittweite h und doppelter Schrittweite $2h$ gewonnenen Näherungswerte.

2.6.10 Schrittweitenadaption - Schrittweitensteuerung

Um die günstigste Schrittweite zu erhalten ist es sinnvoll, diese bei der Berechnung abhängig von ihrem Fehler zu verändern. Bei der *Extrapolation* rechnet man zweimal mit verschiedenen h und schätzt aus dem Vergleich den Fehler ab. Die Toleranz ist dann

$$0,1\varepsilon \leq \underbrace{T_h h^p}_{\text{in Schritt k}} \leq \varepsilon \tag{2.151}$$

mit $\varepsilon > 0$ als vorgegebene Schranke. Außerhalb dieser Toleranz wird die Schrittweite in k modifiziert: $h_k \to h'_k \neq h_k$. *Alternativ* kann man zwei Verfahren verschiedener Konsistenzordnungen anwenden: Falls $|T_h| > T_{h,max} \to$, halbiere h_k (also h im Schritt k) und wiederhole diesen Schritt. Falls $|T_h| < T_{h,min} \to$, verdopple h_k im nächsten Schritt. Die Nachteile der Schrittweitenvariation sind:

- bei Echtzeitsimulation wenig sinnvoll, außer durch Ausnutzung von parallelen Rechnern.

- komplizierte Anwendung bei Mehrschrittverfahren.
- Fehlerabschätzung kostet zusätzlichen Rechenaufwand.
- Festlegung von ε bzw. $T_{h,min}$, $T_{h,max}$ schwierig.

Außerdem existiert eine numerische Stabilität als Mindestanforderung an ein Integrationsverfahren, das heißt Fehler konvergiert gegen 0, das Ergebnis somit gegen die reale Lösung. Allgemeine Stabilitätsaussagen sind nur für spezielle Anwendungen vorhanden.

2.6.11 Schrittweitensteuerung beim Runge-Kutta-Verfahren

Im Prinzip werden die Näherungslösungen durch Verkleinerung der Schrittweite h beliebig genau, da der Verfahrensfehler mit h gegen Null geht. Bei der Verringerung der Schrittweite nehmen aber die notwendigen Rechenoperationen und damit die Rundungsfehler entsprechend zu. Man wird deshalb die Schrittweit h nich kleiner als nötig wählen. Mit Hilfe der Größe

$$q = \left| \frac{\vec{k}_3 - \vec{k}_2}{\vec{k}_2 - \vec{k}_1} \right|, \tag{2.152}$$

in der die $\vec{k}_i$ die am Anfang definierten Werte bedeuten, lässt sich die Schrittweite korrigieren.

Als Regel hat sich bewährt, die Schrittweite für $q < 0,005$ zu verdoppeln, für $q > 0,04$ zu halbieren und den vorhergehenden Schritt mit dieser halbierten Schrittweite zu wiederholen. Für $0,005 < q < 0,04$ kann h unverändert bleiben.

Kapitel 3

Entwicklungsumgebung und Werkzeuge

In diesem Kapitel werden wir auf die Entwicklungsumgebung, dass heißt die Programmiersprache und die Bibliotheken, sowie die und Werkzeuge, die wir benutzt haben, um das Projekt *SIGEL* zu realisieren, eingehen.

Im Abschnitt 3.1 wird diskutiert, warum unsere Auswahl auf *C++* fiel. Außerdem geben wir einen Überblick über die Historie von *C/C++* und den C++-Bibliotheken, die wir benutzen. Am Ende wird zwischen *C*, *C++* und *Java* im Hinblick auf unsere Entscheidung für *C++* verglichen.

In Abschnitt 3.2 wird die Bibliothek *DynaMo* kritisch betrachtet, die dazugehörige Kollisionsbibliothek *SOLID* wird im Unterkapitel 3.3 vorgestellt. Im Abschnitt 3.4 wird schließlich auf die von uns in der Simulation hauptsächlich benutzte Bibliothek *DynaMechs* eingegangen. Zur Berechnung diverser Matrixoperationen nutzen wir die im Abschnitt 3.5 beschriebene Bibliothek *Newmat*.

Die grafische Oberfläche wurde im Projekt mit der Bibliothek *Qt* (Abschnitt 3.6) und dem dazugehrigen *Qt-Designer* (Abschnitt 3.7) realisiert, für die 3D-Darstellung benutzten wir die Standardbibliothek *OpenGL* (Abschnitt 3.8). Die dreidimensionalen Oberflächen lesen wir aus Dateien im Format *VRML* unter Zuhilfenahme der Bibliothek *CV97* ein wie im Abschnitt 3.9 beschrieben.

Zur Parallelisierung der GP nutzen wir die unter 3.10 beschriebenen Bibliothek *PVM*, ein Paket, das es uns ermöglicht, mehrere Rechner in einem Netzwerk zu einer virtuellen Rechenmaschine zusammenzufassen.

Abschließend werden wir noch auf die beiden von uns am meisten genutzten Tools, *CVS* (Abschnitt 3.11) und *Make* (Abschnitt 3.12) eingehen.

3.1 Die Programmiersprache C++

C wurde in den Siebziger-Jahren von Kernighan und Ritchie entworfen. Die Verbreitung von *C* basiert auf dem Erfolg des *UNIX*-Betriebssystem, zu dessen Implementierung *C* benutzt wurde. *C* kombiniert die Vorteile von assemblerähnlichen

und höheren, strukturierten Programmiersprachen.

Bjarne Stroustrup stellte 1985 erstmals die Programmiersprache *C++* vor. Sie hieß damals noch *C with classes* und war genau das: Eine objektorientierte Obermenge von *C*. Seitdem ist *C++* gewachsen und weitere Konstrukte, die insbesondere das Programmieren im Großen vereinfachen sollen, sind hinzugekommen, z.B. Templates, Namespaces, Exceptions (vgl. auch [18]).

Mitte der Neunziger-Jahre veröffentlichte die Firma *Sun Microsystems* die in ihren Reihen entwickelte Programmiersprache *Java*. Mit dem Aufkommen des Internets wuchs der Wunsch nach einer plattform- und betriebssystemunabhängigen Programmierung, die *Sun* durch einen Byte-Code-Interpreter umsetzte. Obwohl Java wegen seiner ähnlichen Syntax häufig mit *C* und *C++* verglichen wird, handelt es sich doch um eine gänzlich andere Sprache. So macht der Verzicht auf Zeigerarithmetik Garbage Collection möglich und führt damit zu einem völlig anderen Programmierstil. Die Tatsache, daß das Programm nicht mehr zu Maschinencode, sondern zu Bytecode compiliert wird, führt dazu, daß Aspekte der maschinennahen Programmierung und Optimierung völlig ausgeblendet werden. Die im Folgenden genannten Kriterien waren zum größten Teil entscheidend für unsere Wahl von *C++*.

3.1.1 Bibliotheken

Da von vornherein feststand, dass *SIGEL* ein so großes Gebilde wird, dass es von uns alleine nicht bewältigt werden kann, war die Einbeziehung von Bibliotheken eine Notwendigkeit. Die wichtigsten Bibliotheken sind die GUI-Bibliothek, die Dynamiksimulation, die Kollisionserkennung, die Grafikausgabe und das GP-System. Allerdings haben wir uns später für ein eigenes GP-System entschieden.

Im Falle der GUI-Bibliothek wurden wir in der Wahl nicht weiter eingeschränkt. Für *C* steht *gtk* zur Verfügung, für *C++* gibt es *Qt*. Mit den Wrappern *Gtk–* und *QtC* lassen sich die Bibliotheken auch auf jeweils anderen Sprachen verwenden.

Dynamiksimulationen sind, wie wir feststellen mussten, entweder teuer oder rar. Mit *Dynamo* und *DynaMechs* haben wir freie *C++*-Bibliotheken gefunden. Zur Kollisionserkennung gibt es die Bibliotheken *RAPID, V-COLLIDE* und *SOLID*. Allesamt verfügen sie über ein *C*-Interface und sind damit sowohl von *C*, als auch von *C++* ansprechbar. Für die Grafikausgabe ist die Standardbibliothek *OpenGL*, eine *C*-Bibliothek. Als GP-System ist uns nur *SYSGP*, eine *C++*-Bibliothek, bekannt.

3.1.2 Paradigma

Der Programmierstil in allen Sprache ist imperativ. Deklarative[1] Sprachen sind wegen ihrer geringen praktischen Relevanz gar nicht erst zu Kandidaten für die Auswahl geworden[2].

C ist systemorientiert mit problemorientierten Elementen. Es gibt keine Sprachkonstrukte, die die Softwareerstellung im Team besonders unterstützen. Üblicherweise gibt es Schwierigkeiten mit Bezeichnern, weil der Namensraum sehr flach ist. Die Verdeckung von Bezeichnern erfolgt teilweise erst im Linker.

C++ bietet neben der Verschachtelung des Namensraums, die erst in einer späteren Version hinzukam, die Unterstützung des objektorientierten Entwurfs. Dabei kann mit *C++* prozeduraler Code auf vielfache Weise mit objektorientierter Programmierung verquickt werden. Ein weiterer Vorteil von *C++* ist das *generische Programmieren*. Damit sind im Wesentlichen Templates gemeint, durch die Klassen geschrieben werden können, deren Eigenschaften erst bei der Benutzung endgültig festgelegt werden. Häufigste Anwendung dieser Technik sind Datencontainer, denen erst zur Instanziierung der Datentyp der Elemente bekanntgemacht wird.

Java verkörpert den objektorientierten Ansatz noch vollständiger als *C++*, ist aber in der Bereitstellung von Sprachmitteln dennoch sparsamer. So sind die Mehrfachvererbung, deren Notwendigkeit von zahlreichen Experten angezweifelt wird, als auch das Überladen von Operatoren, was, richtig angewendet, zu gut lesbarem Code führt, nicht möglich. Jeglicher Code befindet sich in *Java* innerhalb von Methoden einer Klasse. Rein prozedurales Programmieren ist behelfsweise innerhalb einer Klasse möglich, aber mühsam. Ebenso wie *C++* unterstützt *Java* zahlreiche Möglichkeiten, die Sichtbarkeit von Attributen einzuschränken.

3.1.3 Effizienz

Im Punkt Effizienz ist *Java* gegenüber *C* und *C++* eindeutig im Nachteil. Der *Java*-Compiler generiert keinen Maschinencode, sondern Bytecode, der durch die *Java Virtual Machine* interpretiert werden muß. Dadurch verlängert sich die Laufzeit je nach Problem bis auf das Zehnfache. Neuere Interpreter übersetzen den Bytecode unmittelbar vor der ersten Ausführung in Maschinencode (Just-In-Time-Compilation) und mildern damit das Problem.

Für *C* dagegen existieren mehrere Compiler, die bereits eine lange Geschichte hinter sich haben und in die zahlreiche Optimierungen eingebaut wurden. Die Manualseite des *gcc* gibt Auskunft über die verwendeten Optimierungen. Auch gibt es in *C* wegen des Hintergrundes, dass *C* zur Betriebssystemimplementation benutzt

[1] Seit funktionale und logische Programmiersprachen zu funktional-logischen Programmiersprachen integriert werden konnten (z.B. in [19]), werden all diese Sprachen als *deklarative* Sprachen bezeichnet, im Gegensatz zu imperativen Sprachen. Ein Beispiel einer funktional-logischen Sprache ist *Curry* (siehe auch [20]).

[2] Man begebe sich auf die Suche nach einer Dynamiksimulationsbibliothek für SML.

wird, zahlreiche Sprachkonstrukte, die es dem Programmierer einfach machen, den Code zu optimieren bzw. den Compiler bei der Optimierung zu unterstützen.

3.1.4 Plattformunabhängigkeit

Als Programmiersprache für das Internet, eine höchst heterogene Umgebung, hat *Java* in der Betrachtung der Plattformunabhängigkeit die Nase vorn. Der Bytecode, der sich in der Effizienzbetrachtung als hinderlich erwies, sorgt hier dafür, dass *Java*-Programme überall da laufen, wo eine virtuelle Maschine vorhanden ist (also praktisch überall).

Nicht so gut sieht es bei *C* aus, und noch schlechter bei *C++*. Dass die Programme auf jedem Rechner neu compiliert werden müssen, ist eine Selbstverständlichkeit, jedoch können verschiedene Versionen der Standardbibliothek der Portabilität entgegenwirken.

Die Ausgaben verschiedener Compiler sind im Allgemeinen miteinander unverträglich, Assembler und Linker müssen ebenfalls passen. Bei *C++* kommt noch das Problem des Name Mangling hinzu. Durch spezielle Namensanhängsel werden überladene Funktionen und Operatoren unterscheidbar gemacht. Diese Anhängsel können sich gar von Compilerversion zu -version unterscheiden.

Ferner muss sichergestellt werden, dass die verwendeten Bibliotheken ebenfalls portabel sind.

3.1.5 Diskussion

Es ist zu fragen, welche Kriterien wichtig sind. Das Vorhandensein von Bibliotheken, die es uns ersparen, das Rad neu zu erfinden, ist insofern wichtig, als das wir Zeit und Arbeitskraft sparen, die wir auf das Kernproblem verwenden können. Das Entwerfen und Implementieren einer Dynamiksimulation ist zum einen als sehr schwierig einzustufen und zum anderen nicht Gegenstand der Projektgruppe. Das gleiche gilt für Bibliotheken für Benutzungsschnittstellen. Die Ersparnis durch die Verwendung vorhandener Bibliotheken ist immens.

Da *SIGEL* ein sehr großes Projekt zu werden schien, ist es natürlich sinnvoll, eine Sprache zu haben, die Mittel bereitstellt, um damit umzugehen. Stand der Softwaretechnik ist der objektorientierte Entwurf. Obwohl ein objektorientierter Entwurf auch in rein prozeduralen Sprachen implementiert werden kann (siehe z.B. bei *gtk*), ist eine Sprache, in der man den Entwurf direkt ausdrücken kann, sehr wünschenswert.

Das Programm *SIGEL* hat viele Berechnungen zu tätigen. Zum einen läuft die ganze Zeit die Evolutionsschleife, zum anderen erfordert die Dynamiksimulation viel Rechenleistung. Das Compilat sollte also möglichst viel Rechenzeit sinnvoll nutzen.

Wir haben mit *SIGEL* in erster Linie ein Problem zu lösen. Das Finden einer Lösung ist also primär. Wenn wir das Programm portabel halten, erleichtern wir es anderen Menschen, unsere Arbeit fortzusetzen. Weil wir jedoch im Wesentlichen

	C	*C++*	*Java*	Wichtigkeit
Bibliotheken	√	√		eher wichtig, 3
Paradigma		√	√	neutral, 2
Effizienz	√	√		eher wichtig, 3
Portabilität	(√)		√	eher unwichtig, 1
Ergebnis	6,5	8	3	von 9

Tabelle 3.1: Entscheidungsmatrix für die Programmiersprachen

an Solaris-Maschinen im Rechnerpool gebunden sind, bringt uns die Portabilität keine besonderen Vorteile.

Daneben ist noch festzuhalten, dass die Erfahrungen der Teilnehmer mit den Programmiersprachen ebenfalls von Interesse waren. Es ergaben sich jedoch keine nennenswerten Präferenzen.

In Tabelle 3.1 ist die Entscheidungsmatrix für die Programmiersprache abgebildet. Man sieht, daß *C++* alle Kriterien außer der eher unwichtigen Portabilität erfüllt. Es ist demnach die Sprache unserer Wahl.

3.2 Die Bibliothek DYNAMO

Auf der Suche nach einer Bibliothek zur Dynamiksimulation stießen wir zu Beginn des Projektes auf die Bibliothek DYNAMO, die Bart Barenbrug an der Unversität Eindhoven im Rahmen seiner Master Thesis entwickelt hat. DYNAMO schien allen Anforderungen zu genügen und passte zudem mit der ebenfalls in Eindhoven entwickelten Bibliothek zur Kollisionserkennung SOLID hervorragend zusammen.

Beim Entwurf des Projekts wurde der Simulator auf DYNAMO zugeschnitten, und als Datentypen für Vektoren und Matrizen wurde in anderen Modulen, vor allem in Robot und RobotIO, die Typen von DYNAMO verwand.

Als die Simulation fast fertig war, wurde uns klar, dass das Lösungsverfahren von DYNAMO für unsere Zwecke leider numerisch nicht stabil genug ist. Um möglichst genaue Lösungen zu bekommen, errechnet die Bibliothek einen riesigen Vektor, der alle Abweichungen von den sogenannten *constraints* in einer sehr großen Jacobi-Matrix in Gegenkräfte umwandelt. Bei sehr starken und harten Kräften, wie sie zum Beipiel durch Kollisionen oder aber zufällig errechnete Motorbewegungen durch die GP entstehen können, bricht DYNAMO mit einer Exception ab, die mitteilt, das die Contraints nicht mehr eingehalten werden können.

Für die Simulation von Laufrobotern ist es in unserem Sinne besser, mitunter auf physikalische Genauigkeit zu verzichten, dafür aber dem Verfahren eine absolute Stabilität zu geben. DYNAMO verfolgt aber leider den anderen Ansatz. Bei kleinen Weltmodellen oder aber bei sehr kleinen Berechnungsschritten erzeugt DYNAMO eine sehr reale Physik, bricht aber im praktischen Gebrauch zu häufig ab.

Trotz vieler Versuche mit „weicheren“ Kräften entschieden wir uns kurz vor Schluss der Projektgruppe, doch noch auf eine andere Bibliothek umzusteigen (vgl. auch [21]).

3.3 Die Bibliothek SOLID

Obwohl wir Solid inzwischen nicht mehr benutzen, sei trotzdem im Rahmen eines roten Fadens kurz die Funktionsweise, und der Zusammenhang mit DYNAMO skizziert. SOLID lässt sich zwar theoretisch auch mit anderen Bibliotheken, wie zum Beispiel der unten genannten DYNAMECHS verwenden, der Einbau war aber zu komplex und in der kurzen Zeit nicht zu bewerkstelligen.

3.3.1 Dynamiksimulation mit DYNAMO und SOLID

Beide Bibliotheken sind von der „Technische Universität Eindhoven“ entwickelt worden und empfehlen sich gegenseitig. Das hat den Vorteil, dass sich die Interfaces von erkannten Kollisionen (in SOLID) und einer Kollisionsgegenkraft (in DYNAMO) sehr ähneln.

3.3.2 Die Funktionsweise von SOLID

SOLID steht für „Software Library for Interference Detection“, eine Bibliothek zur Erkennung von Kollisionen. Das Interface von SOLID ähnelt dabei dem von OpenGL. Das hat den Nachteil, dass man nicht mehrere verschiedene Instanzen von SOLID gleichzeitig in einem Programm geöffnet haben kann, was unserem objektorientierten Ansatz wiederspricht. Dadurch, dass die Simulation sowohl bei der Visualisierung als auch bei der Berechnung der Fitness im Slave und damit in einem eigenen Programm gestartet wird, ist das Problem nicht mehr so relevant.

Die Funktionsweise von SOLID ist recht einfach. Zuerst baut man sogenannte *shapes* aus einzelnen Polygonen auf. Aus diesen erzeugt man dann *objects*. So ist es also möglich mehrere gleich aussehende Objekte in einer Erkennung zu haben.

Nun kann man in jedem Frame die aktuelle Position der einzelnen Objekte aktualisieren. Dazu kann man diese anhand einer bei der Erschaffung übergebenen Referenz auswählen und dann verschieben bzw. drehen.

Damit die Kollisionserkennung vernünftige Ergebnisse erzielt, ist es sinnvoll, mit den geschätzten Positionen des nächsten Frames zu rechnen und nur dann eine Gegenkraft in der Dynamik zu erzeugen, wenn beide Objekte sich immer noch aufeinander zubewegen.

Die Rückmeldung der Kollision funktioniert bei SOLID über eine Callback-Funktion. Diese Funktion bekommt von SOLID die Referenzen der beiden Objekte, sowie die lokalen Koordinaten der Kollision mitgeliefert. Dazu erhält man einen Normalenvektor der Kollision. Das sind genau die Werte, die DYNAMO für Kollisionsgegenkräfte braucht (vgl. auch [22]).

3.4 Die Bibliothek DYNAMECHS

Es handelt sich bei DYNAMECHS um eine Bibliothek zur Dynamiksimulation in Echtzeit und zur Hydrodynamiksimulation. Ursprünglich wurde sie von *Dr. Scott McMilian* an der Universität in Ohio entwickelt und sollte für Unterwasserroboter zum Einsatz kommen. Die Bibliothek ist komplett in *C++* geschrieben und ist von daher für uns besonders interessant. Das sie auch eine Quasi-Kollisionserkennung anbietet und im Gegensatz zu anderen Simulationen *funktioniert* und *getestet* ist, ist sie für uns ein guter Ersatz für *DynaMo* geworden. Es können simple kinematische Ketten bis hin zu komplizierten Baumstrukturen aufgebaut werden. Diverse Gelenktypen werden unterstützt und sind sehr effizient implementiert. *DynaMechs* setzt komplett auf *OpenGL*, dazu wird eine „workalike"-Version angeboten. *DynaMechs* ist frei erhältlich und verfügt über einen aktuellen Fehlerreport. Auch die Beispiele zu dieser Bibliothek konnten uns hinreichend überzeugen (vgl. [23]).

3.4.1 Grobe Klassenübersicht

Alle hier benannten Klassen leiten sich von der Superbasisklasse *dmObject* ab. Zu den in erster Instanz davon abgeleiteten Klassen gehört auch die Klasse *dmSystem*. Dies ist die abstrakte Basisklasse für alle simulierten Systeme. Hier stellt man Gelenkvariablen und verschiedene Simulationsparameter, u.a. das Referenzkoordinatensystem unter wahlweiser Benutzung eines Quaternions oder kartesischen Vektors ein. Ferner befinden sich hier Methoden zur Vorwärtskinematik via homogener Koordinatentransformation. Die Methode *ABDynamics* ist dabei das Herz der Bibliothek. Hier werden alle relevanten Berechnungen für die Multikörperdynamik durchgeführt.

Von dieser Systemklasse leiten sich *dmEnvironment*, *dmArticulation* und auch andere direkt ab. Letztere ist die Klasse für die Darstellung und Administration von Gelenken in topologischen Baumstrukturen. Geschlossene Ketten sind in der Subklasse *dmClosedArticulation* zugelassen. In *dmEnvironment* werden umgebungsspezifische Parameter wie z.B. Gravitation, Reibungskoeffizienten, Dämpfung, Elastizität und Fluiduumdichte eingestellt. Ebenfalls an dieser Stelle wird der Filename des „Terrains", auf welchem der Roboter sich bewegt eingestellt. Hier befinden sich alle Informationen für die Interaktion, wie Kraftberechnung von Körpern und die Umwelt. Die Umwelt wird dabei als prismenähnliches Terrain (Gitterflächen) modelliert. In in allen Klassen befinden sich auch zwei Klassenmethoden *draw* und *drawInit*, welche nicht in der Bibliothek definiert sind. Der Programmierer selbst muss dies tun, damit er die 3D API seiner Wahl benutzen kann. Beispiele für *OpenGL* befinden sich auch in der Distribution.

Von *dmObject* wird ebenfalls in 1. Instanz *dmLink* abgeleitet. Hier handelt es sich um die abstrakte Basisklasse für alle davon abgeleiteten Gelenkklassen. Hier können im Gelenkeingangsvektor physikalische Motorgrößen etc. eingegeben werden. Man stellt hier auch die Coulombreibung ein. Von dieser Klasse wird *dmRigidBody* abgeleitet: Dort werden alle Dynamikparameter für Gelenke für die Dy-

namiksimulation gelagert. Darüber werden mehrere Klassen, wie z.B. *dmSphericalLink*, oder *dmMDHLink* abgeleitet, die die verschiedene Gelenkarten und deren spezielle Transformationen bereit stellen.

Da es in *DynaMechs* keine echte Kollisionserkennung gibt, behilft man sich mit ausgewählten Kontaktpunkten, die zur Quasikollision herangezogen werden. Ihre Definition erfolgt in *dmContactModel*, welche von *dmForce* abgeleitet wird, die wiederum von *dmObject* abgeleitet wurde. In *dmContactModel* gibt es die Methode *dmContactModel::computeContactForce()*, welche die erste Anlaufstelle für Kollisionen von Objektkontaktpunkten und dem Terrain ist. Über Abstandprüfungen entlang des errechneten Normalenvektors des Terrains werden dann die Kollisionen erkannt.

Von der von *dmObject* abgeleiteten Klasse *dmIntegrator* werden verschiedene Integratorklassen abgeleitet. Mit ihnen werden die nummerischen Integrationen der in der Simulation entstehenden Bewegungsgleichungen (Differentialgleichungssysteme 2. Ordnung) gelöst. Zur Zeit werden die Verfahren Eulervorwärts, Runge-Kutta 4.Ordnung und ein Derivat davon mit adaptiven Schrittweitenvektor angeboten.

In der Klasse *dmRevDCMotor* befinden sich die für Gleichstrommotoren üblichen Konstanten wie Drehmoment, Trägheitsmoment, Reibungskoeffizient, Viskositätsreibungskonstante etc. um einen Gleichstrommotor zu simulieren. Allerdings machen wir im Rahmen unseres Projekts keinen Gebrauch davon.

3.4.2 Mathematische Erläuterungen

Um die Funktionen dieser Bibliothek zu verstehen, werden hier nun die wichtigsten theoretischen Aspekte von *DynaMechs* erörtert.

- Für die Einstellung der Coulombschen Reibung in *dmLink* wird ein skalarer Faktor $\mu_{Coulomb}$ mit dem Gelenkgeschwindigkeitsvektor multipliziert, um ein Drehmoment entgegen der Bewegungsrichtung zu erhalten (τ entspricht dem Drehmoment bei einem *dmRevoluteLink* und einer Kraft bei einem *dmPrismaticLink*):

  $$\tau = \mu_{Coulomb} \cdot \dot{q}. \tag{3.1}$$

  Wobei q der Positionsvektor in allen 3 kartesischen Richtungen und drei räumlichen Drehwinkeln ist (6x1 Auslenkungsvektor, siehe Abschnitt 2.5). Im Falle von nur einem Freiheitsgrad handelt es sich um einen skalaren Wert q (z.B. stellvertretend für die Wegvariable x).

- Generell wird bei Überschreitung der Gelenkauslenkungslimits um Δq sofort ein Drehmoment $\tau = \beta_{Limit} \cdot \Delta q$ entgegen der Überauslenkung ausgeführt, während in den anderen Richtungen ein Dämpfungsdrehmoment $\tau = -\alpha_{Limit} \cdot \dot{q}$ durchgeführt wird, um unnötige Schwingungen aufgrund von Wiederüberauslenkung auszugleichen.

- Die Klassenmethode *dmMDHLink::setJointLimits* spezifiziert den Bereich der Auslenkung von Gelenken (hinsichtlich ihrer rotatorischen/translatorischen Eigenschaften) für den Gelenkpositionsvektor q. Bei Überschreitung dieses Bereichs um Δq werden Gegendrehmomente via Elastizitätskonstante β und Dämpfungskonstante α nach $\tau = \beta \cdot \Delta q - \alpha \cdot \dot{q}$ überführt.

- In *dmRigidBody*, abgeleitet von *dmLink*, werden alle Inertialparameter gesetzt und abgefragt. Dazu wird intern in *DynaMechs* eine *DynaMechs*-spezifische 6x6 Inertialmatrix

$$\mathbf{I} = \begin{pmatrix} \tilde{I} & m \cdot P_{cog} \\ m \cdot P_{cog}^T & m \cdot E_3 \end{pmatrix} \tag{3.2}$$

  aufgestellt. Der Programmierer übergibt dazu aber nur

$$\tilde{I} = R^T I_{cog} R + m P_{cog} P_{cog}^T, \tag{3.3}$$

  den Trägheitstensor bezüglich der Körperhauptachsen, I_{cog}, den Trägheitstensor bezüglich der Körperschwerpunktachse, m, die Masse des Körpers, P_{cog}, den Körperschwerpunktvektor und E_3, den dreidimensionaler Einheitstensor.

3.4.3 Abschließende Gedanken

Wie gesagt erfolgen die Kollisionen mit dem Boden durch ausgewählte Kontaktpunkte der Objekte. Dagegen gibt es keinerlei Kollisionen zwischen zwei oder mehreren Objektkörpern. Man muss also mit Hilfe von minimal und maximal zulässigen Auslenkungen, die je Objekt spezifisch vom Benutzer eingestellt werden, Kollisionen vermeiden. Alle Objektkoordinatensysteme befinden sich im System der sogenannten Denavit-Hartenberg Parameter (siehe Grundlagenkapitel *Roboterkinematik 2.2*).

3.5 Die Bibliothek NEWMAT

Diese C++-Bibliothek ist für Wissenschaftler und Ingenieure gedacht, die eine Vielzahl von Standardmatrixoperationen, aber auch numerische Besonderheiten bezüglich der Matrizenrechnung und Lösung von Gleichungssystemen brauchen. Das Hauptgewicht liegt in speziellen Berechnungen wie Fehlerquadratminimierungen oder dem Lösen von Ricatti-Differentialgleichungen via numerischer Gauss-Seidel Rückwärtsiteration. Die Berechnung von Eigenwerten mit Standardverfahren wie *Singular Value Decomposition* ist ein weiterer Schwerpunkt

NEWMAT unterstützt die diverse Matrizenarten, darunter auch diverse Darstellungen für spärlich besetzte Matrizen. Die Bibliothek umfasst u.a. die Operationen * , + , - sowie die in der Naturwissenschaft gängigen Matrizenoperationen wie z.B. Inversenbildung oder auch die Berechnung von Eigenwerten. Der moderne Gebrauch von *Namespaces* ist auch in dieser Bibliothek konsequent umgesetzt.

Es befindet sich auch eine Schnittstelle für das Standardwerk in der numerischen Mathematik *Numerical Recipes in C*. Daher auch die reichhaltigen numerischen Matrizenoperationen, wie zum Beispiel *Hessenbergtransformation* .

NEWMAT ist für Matrizen der dimensionalen Größe 15 x 15 bis zur maximalen Größe, die die benutzte Rechenmaschine schafft, gedacht. Für Unix-Maschinen oder PCs, die eine interne 32-Bit Modellierung von Zahlen benutzen, wird die Begrenzung durch die Größe des vorhandenen Speichers physikalisch begrenzt. Die Zahl der Elemente in einer Zeile/Spalte kann nicht die maximale Größe von einem *int*-Wert überschreiten.

NEWMAT arbeitet zwar auch für sehr kleine Matrizen, aber wird dann ziemlich wirkungslos und ineffizient. Eine sogenannte *lazy evaluation* nähert numerisch die Matrixausdrücke nur an, um Leistungsfähigkeit und Effizient zu verbessern und den Gebrauch von Zwischenspeicherressourcen zu verringern.

3.6 Die Bibliothek QT

Dieses Kapitel enthält eine Einführung in die grafische Benutzeroberfläche QT. Schon in einer frühen Phase des Projekts musste entschieden werden, mit Hilfe welcher Bibliothek die grafische Benutzeroberfläche erstellt werden soll. Zwar könnte das Programm rein theoretisch auch komplett über die Kommandozeile laufen, aus Gründen der Benutzerfreundlichkeit entschied man sich jedoch schon zu Beginn der Projektgruppe, dass das Programm über eine grafische Benutzeroberfläche steuerbar sein soll.

Zur Auswahl standen TCL/TK, XFORMS, GTK und QT. Die Wahl fiel sehr schnell auf QT, da es die Vorteile aller anderen Bibliotheken in sich vereint und auch weitere eigene Vorteile bietet. QT besitzt zwar auch einige Nachteile, diese sind jedoch nicht sehr schwerwiegend bzw. werden durch einige für QT erhältliche Tools wieder kompensiert.

3.6.1 Einleitung

QT ist eine Bibliothek zur Erstellung grafischer Benutzeroberflächen und wurde von der norwegischen Firma *Trolltech* entwickelt. Sie ist für Unix, Linux und Windows erhältlich. Es wird zwischen der Professional Edition und der Free Edition unterschieden. Die Professional Edition ist kostenpflichtig, schließt jedoch Support mit ein, während die Free Edition unter der Free Edition License frei erhältlich ist, jedoch keinen Support bietet. Ab Version 2.2 ist die Bibliothek zusätzlich unter der *GPL (GNU Public License)* erhältlich.

QT wurde in C++ geschrieben und ist daher vollständig objektorientiert. Es werden zahlreiche Klassen zur Verfügung gestellt, die entweder sofort genutzt oder nach eigenen Wünschen erweitert werden können. Die Kommunikation der Widgets mit der Applikation und untereinander findet über den sogenannten Signal/Slot-Mechanismus statt, auf den später noch genauer eingegangen wird.

Einige hervorzuhebende Features sind Unterstützung von Drag and Drop, der Layout Manager, Internationalisierung, sowie OPENGL-Support über die QT-Extensions. Ab Version 2.2 gibt es zusätzlich den QT DESIGNER, mit dem man interaktiv grafische Benutzeroberflächen entwerfen kann. Auf den QT DESIGNER wird in Kapitel 3.7 näher eingegangen.

Eines der bekanntesten Projekte, welches mit QT entwickelt wird, ist die Desktop-Umgebung KDE (K Desktop Environment). Ausführlichere Informationen zum Thema QT finden sich unter [24].

Im Folgenden wird nun anhand einiger Beispiele auf die QT-Bibliothek eingegangen um einige Prinzipien zu verdeutlichen. Begonnen wird mit dem obligatorischen „Hello World"-Beispiel, welches nicht mehr macht, als einen Button mit dem Label „Hello World" anzuzeigen. Dieses wird dann im Zusammenhang mit dem Signal/Slot-Mechanismus so erweitert, dass das Programm terminiert, wenn der Button gedrückt wird.

3.6.2 Hello World in QT

Das folgende „Hello World"-Beispiel wurde dem Tutorial entnommen, welches in der QT-Dokumentation enthalten ist. Das Programm erzeugt ein Fenster, das komplett von einem Button mit der Aufschrift „Hello world!" gefüllt wird. Abbildung 3.1 zeigt das Ergebnis dieses Programms, während Abbildung 3.2 den Code zur Erzeugung des Programms anzeigt.

Abbildung 3.1: „Hello world!" mit QT.

Über die `include`-Zeilen binden wir die Header-Dateien für die beiden Objekte, die benötigt werden, ein. Im `main`-Teil passieren nun der wichtige Teil des Programmes.

Zuerst erzeugen wir eine Instanz eines Objektes vom Typ `QApplication`. In diesem Falle ist sie statisch. Jede Anwendung, die die QT-Bibliothek benutzt, muss *genau eine* Instanz von `QApplication` besitzen. `QApplication` kümmert sich unter anderem um anwendungsweite Resourcen, wie Standardschrift und Standardcursor, sowie um das Kommandozeilen-Parsing. Unter X11 werden zum Beispiel Argumente wie `-display` erkannt. Jedes erkannte Argument wird dabei aus der Liste der Argumente gestrichen und `argc` entsprechend um eins dekrementiert.

In der nächsten Zeile wird der Button erzeugt. Es handelt sich um einen normalen `QPushButton`. Die meisten Widgets in QT haben mehrere Konstruktoren. In diesem Fall benutzen wir einen Konstruktor, dem drei Argumente übergeben

```
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
    QApplication a( argc, argv );

    QPushButton hello( "Hello world!" , 0 , "Hello-Button" );
    hello.resize( 100, 30 );

    a.setMainWidget( &hello );
    hello.show();
    return a.exec();
}
```

Abbildung 3.2: Der Code zu „Hello world!“

werden. Über das erste Argument “Hello world!“ wird einfach die Aufschrift des Buttons festgelegt. Das zweite Argument gibt bei diesem Konstruktor das Eltern-Widget (Parent-Widget) des Buttons an.

In QT gibt es ähnlich wie bei TCL/TK eine Eltern-Kind-Beziehung zwischen den Widgets. Zu einem gewissen Grad kümmert sich ein Eltern-Widget um seine Kinder-Widgets. Wird zum Beispiel das Eltern-Widget über die `show()`-Methode angezeigt, so werden auch automatisch alle Kinder angezeigt, sofern nicht eines oder mehrere explizit ausgeblendet werden. Wird ein Eltern-Widget zerstört, sorgt es dafür, dass auch alle seine Kinder zerstört werden. Dies erscheint zwar nicht sehr fürsorglich, ist jedoch programmiertechnisch äußerst angenehm. Ein Widget wird immer in seinem Eltern-Widget angezeigt. Wird kein Eltern-Widget angegeben, also ein *NULL*-Zeiger übergeben, so bekommt das Widget ein eigenes Fenster. Dies ist auch in unserem Beispiel der Fall.

Das dritte Argument des benutzten `QPushButton`-Konstruktors ist ein QT-interner Name für das Widget. Diese können zum Beispiel bei der Verwendung des Signal/Slot-Mechanismus sehr hilfreich sein, da sie helfen, Fehlerquellen schneller zu finden.

Mit dem Aufruf der Methode `hello.resize( 100 , 30 )` ändern wir die Breite des Buttons auf 100 Pixel und die Höhe auf 30. Die Methode `resize()` ist dabei keine Button-spezifische Methode, sondern QPushButton erbt sie von der sehr allgemeinen Klasse QWidget. Allerdings überschreibt `QPushButton` diese.

Nun wird über `a.setMainWidget( &hello )` der Button als Hauptwidget der Applikation festgelegt. Wird das Hauptwidget der Applikation zerstört, so wird auch die Applikation beendet. Es muss kein Hauptwidget festgelegt werden, bei den meisten Applikationen ist es jedoch der Fall.

Wenn ein Widget erzeugt wird, ist es noch nicht auf dem Bildschirm sichtbar.

Daher muss es erst über die Methode `show()` sichtbar gemacht werden.

Über `a.exec()` gibt die Funktion `main` die Kontrolle an QT ab, so dass Ereignisse erkannt und entsprechend behandelt werden können. Nach Beendigung der Applikation gibt `exec()` einen Rückgabewert zurück, welcher über `return` nach außen weitergereicht wird.

3.6.3 Signals und Slots

Wird der Button im „Hello World"-Beispiel gedrückt, passiert nichts. Im Folgenden wollen wir nun dafür sorgen, dass die Appliktion beendet wird, wenn der Button geklickt wird. Bei XFORMS wurde dies mit Hilfe von Callback-Funktionen gemacht. QT benutzt dafür den Signal/Slot-Mechanismus.

In QT kann jedes Objekt Signale senden und über Slots Signale empfangen. Signale werden meist ausgesendet, wenn sich das entsprechende Objekt in irgendeiner Art ändert, die für die Außenwelt interessant sein könnte. Zum Beispiel sendet ein Button das Signal `clicked()`, wenn er angeklickt wird oder ein Eingabefeld das Signal `textChanged()`, wenn sich der Text geändert hat.

Dass ein Objekt Signale aussenden kann, ist ja schon schön und gut, es würde jedoch nicht viel bringen, wenn es niemand mitbekommt. Hier kommen nun die Slots ins Spiel. Denn ein Signal, welches von einem Objekt ausgesendet wird, kann mit einem Slot eines anderen Objekt verbunden werden. Diese Slots sind dabei nichts anderes als Member-Funktionen der Objekte. Wenn ein Signal mit einem Slot verbunden ist, führt das Signal zur Ausführung der entsprechenden Methode.

Momentan könnte man den Eindruck bekommen, dass dies eigentlich wieder die altbekannten Callback-Funktionen sind. Der Signal/Slot-Mechanismus ist jedoch um einiges flexibler. So können Signale auch mit anderen Signalen verbunden werden oder das Signal eines Objektes mit einem Slot desselben Objektes. Außerdem kann ein Signal mit beliebig vielen Slots verbunden werden und ein Slot mit beliebig vielen Signalen. Im Vergleich zu XFORMS Callback-Funktionen ist der Signal/Slot-Mechanismus auch um einiges differenzierter. Bei XForms kann ein Objekt nur *eine* Funktion auslösen, wenn *irgendwas* mit dem Objekt geschieht, während bei QT für unterschiedliche Ereignisse unterschiedliche Slots aufgerufen werden können. Ein Button besitzt z.B. die Signale `pressed()`, `released()`, `clicked()`, `toggled()` und `stateChanged()`. Auf jedes dieser Ereignisse kann also in unterschiedlicher Weise reagiert werden.

Viel wichtiger ist jedoch, dass dadurch die Entwicklung echter Software-Komponenten möglich wird. Es ist nämlich so, dass ein Objekt, welches ein Signal aussendet, gar nicht weiß, ob ein anderes Objekt dieses Signal hört oder nicht. Im „Hello World"-Beispiel sendet der Button in Wirklichkeit jedesmal das Signal `clicked()`, wenn er geklickt wird. Es ist nur so, dass es niemand hört.

Da wir möchten, dass die Applikation beendet wird, wenn der Button gedrückt wird, müssen wir also dafür sorgen, dass die Applikation das Signal `clicked()` empfängt. Es muss in diesem Fall nur noch der richtige Slot gefunden werden. Und tatsächlich besitzt ein Objekt vom Typ QApplication einen Slot namens `quit()`.

Jetzt müssen wir das Signal nur noch mit dem Slot verbinden. Dies geht über folgende Methode:

```
QObject::connect( &hello , SIGNAL( clicked() ) ,
                  &a , SLOT( quit() ) );
```

Sie kann zum Beispiel zwischen die `resize()`- und `setMainWidget()`-Methode im ursprünglichen „Hello World“-Code eingefügt werden. Das Signal `clicked()` des Objektes `hello` wird mit dem Slot `quit()` des Objektes `a` verbunden. Bei dem oben benutzten Verfahren gibt es nur ein Problem. Was ist, wenn wir eine Software-Komponente entwickeln, und diese zum Beispiel einen Button besitzen soll, der die Applikation beendet? Die Software-Komponente kann ja gar nicht wissen, wie die Instanz der QApplication heißt. Für diesen Fall stellt QT einen globalen Zeiger namens `qApp` zur Verfügung, der auf die Instanz zeigt. Daher sollte man die Methode bevorzugt wie folgt verwenden:

```
QObject::connect( &hello , SIGNAL( clicked() ) ,
                  qApp , SLOT( quit() ) );
```

3.6.4 OPENGL-Einbindung

Bei QT gibt es standardmäßig verschiedene Klassen, die eine OPENGL-Einbindung ermöglichen. So zum Beispiel QGLWidget, welche ein Widget zur Verfügung stellt, in dem eine OPENGL-Szene gerendert werden kann oder QGLContext, in der ein OPENGL-Context gekapselt wird.

3.7 Das Werkzeug QT DESIGNER

Der QT DESIGNER ist ein Werkzeug, das es einem ermöglicht, grafisch Benutzeroberflächen zu erstellen. Er ist in etwa mit dem Form-Designer der GUI-Bibliothek XFORMS zu vergleichen, ist diesem jedoch bei Weitem überlegen.

Im Folgenden wird der QT DESIGNER nun etwas ausführlicher beschrieben, da viele GUI-Klassen des Projektes mit seiner Hilfe entstanden sind und es so einfacher ist, nachzuvollziehen, warum einige Klassen so aussehen, wie sie aussehen.

3.7.1 Die Arbeitsumgebung des QT DESIGNER

In Abbildung 3.3 kann man das Hauptfenster des QT DESIGNERS sehen. Es besteht aus einem Menu, einer Toolbar und einer großen Arbeitsfläche, auf der man seine Widgets zusammenstellen kann. Es wird nun kurz beschrieben, wie man ein neues Widget zusammenstellt und wie dieses dann in die Applikation kommt.

Nachdem der QT DESIGNER gestartet wurde, sieht der Bildschirm in etwa so wie in Abbildung 3.3 aus, nur das Fenster zwischen den beiden anderen Fenstern auf der Arbeitsfläche ist noch nicht vorhanden. Zunächst muss dafür gesorgt werden, dass ein Fenster zur Bearbeitung des Widgets vorhanden ist. Dies wird erreicht, indem man entweder im Hauptmenu unter „Datei“ den Punkt „Neu“ anwählt

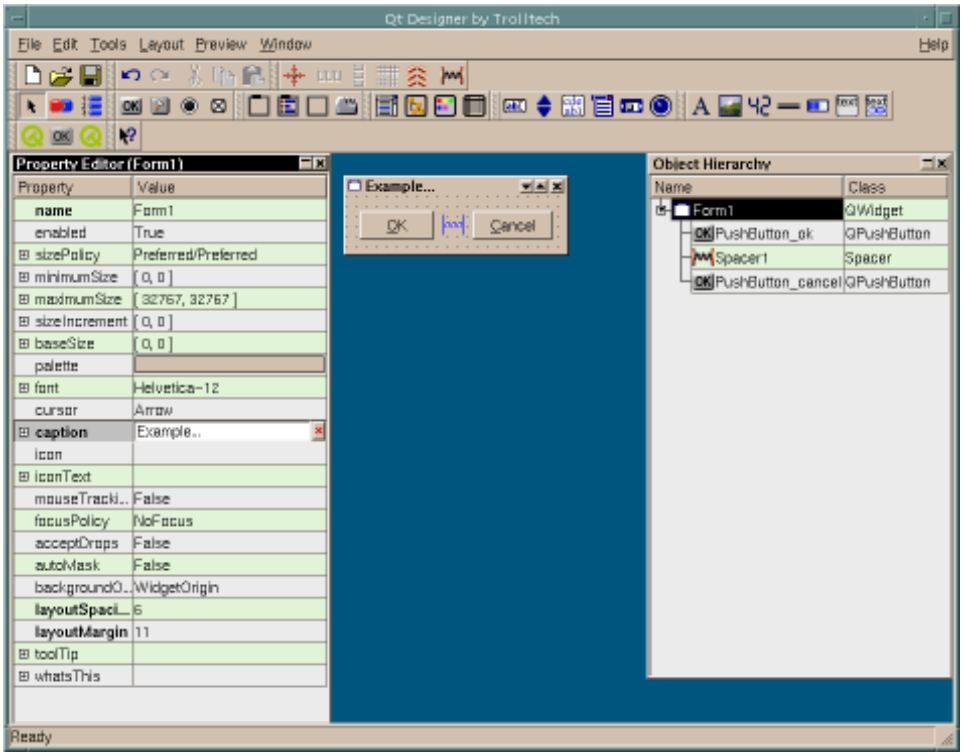


Abbildung 3.3: Der QT DESIGNER.

oder aber, indem man auf den entsprechenden Punkt in der Toolbar klickt. Ist dies geschehen, wird man in einem Dialog gefragt, welches Template man benutzen möchte. Wir entscheiden uns hier für „Widget“. Andere oft genutzte, praktische Templates sind „Dialog“ und „Wizard“.

Nachdem „Widget“ gewählt wurde, erscheint ein neues leeres Fenster auf der Arbeitsfläche namens „Form1“, auf welches nun Widgets und Layouts platziert werden können. Das Fenster auf der linken Seite des Arbeitsplatzes ist der sogenannte „Property Editor“. In diesem können für fast alle Widgets wichtige Eigenschaften geändert werden (z.B. der Text, der auf einem Button steht, etc.). Wird ein neues Widget auf die Form gesetzt, so werden dessen Eigenschaften sofort im „Property Editor“ angezeigt, so dass diese geändert werden können.

Das Fenster auf der rechten Seite der Arbeitsfläche ist die „Object Hierarchy“. Hier wird die Objekthierarchie angezeigt, also die verschiedenen Parent/Child-Beziehungen.

Abbildung 3.5 zeigt wie eine mit dem QT DESIGNER erstellte Form beispielsweise aussehen könnte. Die Kästen, welche um einige Widgets gezogen sind, markieren Layouts.

Zusätzlich bietet der QT DESIGNER eine Möglichkeit, verschiedene Signale der Widgets mit Slots zu verbinden, sowie neue Slots zur Form hinzuzufügen. Dar-

auf wird in diesem Zusammenhang jedoch nicht weiter eingegangen, da dies den Rahmen dieses Kapitels sprengen würde.

3.7.2 Die Nutzung der mit dem QT DESIGNER erstellten Klassen

Ist das Widget bzw. die Form zufriedenstellend und möchte man es in seinem Code benutzen, muss es erst einmal abgespeichert werden. Der QT DESIGNER speichert Forms im `.ui`-Format. Die Forms werden in diesem Format als XML-Code gespeichert. Mit diesem XML-Code kann man im C++-Code natürlich nichts anfangen. Daher muss die `.ui`-Datei zuerst mit einem Programm namens `uic` bearbeitet werden. Die Hauptaufgabe des `uic` besteht darin, aus den `.ui`-Dateien entsprechende Header- und Implementierungs-Dateien zu erstellen. Zu jeder Form wird eine Klasse erstellt, dessen Konstruktor dafür sorgt, dass die Widgets erzeugt werden und eventuell auch schon Signale mit Slots verbunden werden.

Der `uic` kann jedoch noch mehr. Er kann nämlich auch Header- und Implementierungs-Dateien von Klassen erstellen, die von der ursprünglichen Klasse erben. Konkret heißt dies, dass wenn man eine Form namens „FormBase“ im QT DESIGNER erstellt hat, der `uic` Header- und Implementierungs-Dateien erzeugen kann, die die Form „FormBase“ realisieren. Diese Klasse heißt dann ebenfalls „FormBase“. Aus diesen beiden Dateien kann er jedoch auch noch Dateien für eine weitere Klasse generieren, die dann zum Beispiel einfach nur „Form“ heißen kann und von „FormBase“ erbt. Ein Diagramm dazu ist in Abbildung 3.4 dargestellt. Und hier zeigt sich nun der große Vorteil des QT DESIGNERS gegenüber

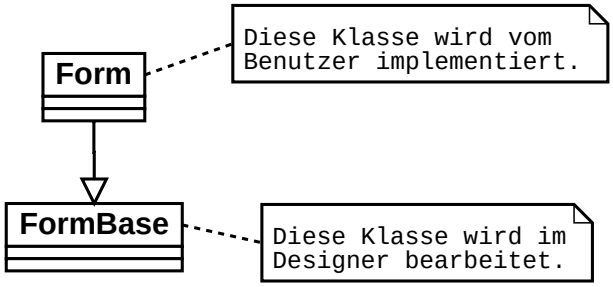


Abbildung 3.4: Das Klassendiagramm des Beispiels.

dem Form Designer und der Vorteil der Objektorientierung. Denn die selbst programmierte Funktionalität der Form wird nicht in den Basis-Klassen implementiert, sondern in den Klassen, die von den Basis-Klassen erben. Dieses Vorgehen mag auf den ersten Blick umständlich erscheinen, bietet jedoch den großen Vorteil, dass man nachträglich Änderungen an den Forms bzw. Widgets vornehmen kann. Denn der Code, in dem die eigentliche, eigene Implementierung stattfindet ist davon garnicht betroffen, im Gegensatz zum Form Designer, wo einmal implementierter Code verloren ist, wenn man nachträglich Änderungen an den Forms vornimmt.

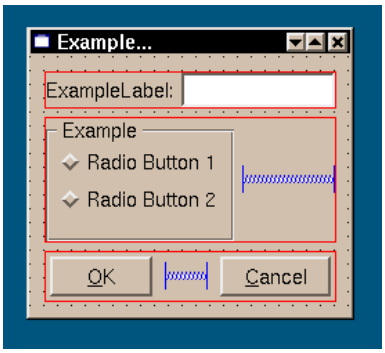


Abbildung 3.5: Beispiel einer Form des QT DESIGNER.

Dies ist auch der Grund, warum viele GUI-Klassen des Projekts gar nicht bzw. wenig dokumentiert sind. Denn teilweise werden viele Widgets in den Basis-Klassen realisiert und da diese sich öfter ändern, würde es nichts bringen, diese zu dokumentieren.

Wie wahrscheinlich aus dieser kurzen Zusammenfassung ersichtlich ist, ist der QT DESIGNER ein weiteres starkes Argument für die Nutzung von QT gewesen.

3.8 Die Bibliothek OPENGL

OpenGL bedeutet Open Graphic Library und ist eine Grafikbibliothek die mittlerweile in fast allen gängigen Progammiersprachen für die meisten Platformen übersetzt wurde. Für die Entwicklung der OpenGL ist das sogenannte „Architectural Review Board“ (ARB) verantwortlich. Mitglieder des ARB sind Hersteller der High-End-Workstations wie Silicon Graphics und DEC, sowie Microsoft. Das ARB kümmert sich um Dinge, welche die Implementierung von OpenGL betreffen und um die Erweiterung der OpenGL-Architektur (vgl. auch [25]).

3.8.1 Prinzipien der OpenGL

OpenGL ist ein Software-Interface zur Hardware. Zweck ist die Darstellung von zwei- und dreidimensionalen Objekten mittels eines Bildspeichers. Diese Objekte sind entweder Images, die aus Pixeln zusammengesetzt sind, oder geometrische Objekte, die durch Vertices (Raumpunkte) beschrieben sind.

Anmerkungen zur Definition von OpenGL: Der größte Teil von OpenGL benötigt als Teil der grafischen Hardware einen Frame-Buffer. Viele Aufrufe von OpenGL-Funktionen bewirken das Zeichnen von Objekten wie Punkte, Linien, Polygone und Bitmaps, wie das Zeichnens erfolgt (wenn z.B. Antialiasing oder Textur-Mapping eingeschaltet sind), hängt dabei von der Existenz eines Frame-Buffers ab. Darüberhinaus sind einige Teile von OpenGL unmittelbar mit der Manipulation des Frame-Buffers befasst. Derzeit sind die Einbindungen des ARB für

C und FORTRAN 77 fertig, für ADA ist sie in Arbeit.

3.8.2 Funktionalität

In der 3D-Ausgabe unterstützt die OpenGL unter Anderem schattierte Punkt, Linien und Polygone, Stacks für Attribute und Matrizen sowie die Verwendung von Display-Listen zum puffern wiederholt auftretender Zeichenbefehlsketten. Durch Depth Buffering werden verdeckte Objekte ausgeblendet und die Anzeige wird durch Antialiasing, also Glätten der Linien und Polygonkanten verfeinert. Des weiteren unterstützt die OpenGL einige Spezialeffekte und diverse Materialeigenschaften, wie Texturen, Bump Mapping etc.

3.8.3 Fragmentoperationen

Fragmente entstehen bei der Rasterung und bestehen aus Pixelinformationen. Diese beinhalten Koordinaten, Farbe sowie die Texturkoordiaten. Befehle, die sich auf Fragmente beziehen enthalten daher diese Informationen.

3.8.4 Die Buffer von OpenGL (Der Bildspeicher)

In OpenGL werden einige Buffer benutzt, um beim Rendern maximal mögliche Qualität und Geschwindigkeit zu erzeugen. So gibt es zwei Buffer, die die Farbe speichern, einen sogenannten Z-Buffer für die Tiefenwerte um keine verdeckten Objekte anzuzeigen, einen Akkumulationsbuffer für akkumulierte Werte, die mit dem Bildschirmspeicher verknüpft werden können, einen Stenzilbuffer, damit Masken, also unregelmäßige Fenster beachtet werden und schließlich einen Hilfsbuffer für die sonstigen Daten.

3.8.5 Immediate Mode und Display-List Mode

Grafiksysteme können in zwei verschiedenen Ausgabemodi arbeiten. Diese sind:

- Immediate Mode
  Die grafischen Ausgabeelemente durchlaufen nach ihrer Spezifizierung sofort die gesamte Pipeline und werden zur Anzeige gebracht.

- Display-List Mode
  Die Ausgabeelemente gelangen zunächst in einen Zwischenspeicher. Von dort werden sie zu einem späteren Zeitpunkt „abgeholt“, weiterverarbeitet und angezeigt.

3.8.6 OpenGL als Zustandsmaschine

Die OpenGL-Bibliothek arbeitet wie eine Zustandsmaschine. Eine Zustandsmaschine ist ein Objekt mit einem definierten Zustand. Es verbleibt in diesem Zustand,

bis ein Ereignis eine Zustandsänderung hervorruft. Viele der im folgenden vorgestellten OpenGL-Funktionen ändern den aktuellen Zustand der OpenGL-Maschine und bleiben wirksam, bis sie verändert werden. Mit dem glColor-Befehl wird oftmals die aktuelle Farbe eingestellt, wie im Folgenden Beispiel:

```
glColor3d( 1.0, 0.0, 0.0 );
```

Wenn die Farbe mit glColor gesetzt wurde, verwenden alle mit der OpenGL-Bibliothek erstellten Objekte diese Farbe, bis sie geändert wird. In ähnlicher Weise bleiben Funktionen gültig, die z.B. eine Drehung oder eine Position beschreiben.

3.8.7 Arbeitsweise von OpenGL

OpenGL ist ausschließlich mit der Übergabe in den Framebuffer (und Auslesen aus dem Frame-Buffer) befasst. Andere Peripheriegeräte, die manchmal zur grafischen Hardware gerechnet werden, wie Mäuse und Tastaturen werden zunächst nicht unterstützt. Auf die Möglichkeiten verschiedener Zusatzlibraries wurde bereits hingewiesen. Der Programmierer muss sich selbst um Mechanismen zur Benutzereingabe kümmern.

3.8.8 OpenGL-Datentypen

Um die Portierung von OpenGL-Code von einer Plattform auf eine andere zu erleichtern, werden in den OpenGL-Programmen normalerweise spezielle typedefs anstatt der definierten Datentypen eines bestimmten Rechnertyps verwendet. Statt eine Variable als double zu vereinbaren, soll man in einem OpenGL-Programm OpenGLdouble verwenden. Die Tabelle 3.8.8 zeigt eine Liste der möglichen OpenGL-Datentypen.

Pointer und Felder werden nicht gesondert betrachtet. Ein Feld von zehn GLshort Variablen wird durch

```
GLshort kurzfeld[10];
```

deklariert, ein Feld von zehn Pointern auf Variable vom Typ GLdouble durch

```
GLdouble  *doubles[10];
```

3.8.9 Namenskonventionen

OpenGL liegt normalerweise als C Bibliothek vor. Vielfach existieren für ein Kommando mehrere Funktionen, die sich nur in der Anzahl der Parameter oder im verwendeten Datentyp unterscheiden. Der Funktionsname setzt sich in diesem Fall aus einem einleitenden „gl“ für Open Graphics Library, dem eigentlichen Namen, der Anzahl der Parameter als Ziffer, einem Kürzel für den Datentyp und optional einem „v“ zusammen, falls ein Zeiger auf ein Array anstatt einzelner Parameter erwartet wird. So ergeben sich Funktionsnamen wie glNormal3f zur Spezifikation

OpenGL-Datentyp	C-Datentyp
GLbyte	signed char
GLbitfield	unsigned long
GLboolean	unsigned char
GLclampd	double
GLclampf	float
GLdouble	double
GLenum	unsigned long
GLfloat	float
GLint	long
GLshort	short
GLseizei	long
GLubyte	unsigned char
GLuint	unsigned long
GLushort	unsigned short
GLvoid	void

Tabelle 3.2: Die OpenGl-Datentypen

einer Normalen durch drei Fließkommazahlen, oder glColor3bv für eine Funktion, die die ersten drei Bytes des übergebenen Arrays als Rot-, Grün- und Blauwert interpretiert. Im Folgenden wird stellvertretend für alle Ausführungen solcher Funktionen nur der Basisname angeführt, also etwa glColor.

OpenGL definiert eine Vielzahl von Konstanten, die mit einem großen „GL“ beginnen. Die Konstanten sind komplett in Großbuchstaben gehalten, einzelne Wörter sind durch Unterstriche abgetrennt, (z.B. GL_POLYGON, GL_LIGHTING).

3.8.10 Transformationen

Die Abbildung von Objektkoordinaten in Bildschirmkoordinaten verläuft bei OpenGL in nur zwei Schritten. Die Transformation von Objekt- in Weltkoordinaten und von Welt- in Kamerakoordinaten wird in einer einzigen Transformationsmatrix, der „Model-View“ Matrix, zusammengefasst. Zur weiteren Abbildung in Bildschirmkoordinaten wird eine zweite, „Projection“ genannte, Matrix herangezogen. Jede dieser beiden Matrizen kann unabhängig von der anderen angesprochen und unter anderem mit Translationen, Rotationen und Skalierungen verändert werden. Außerdem stellt OpenGL für beide Matrizen eigene Stacks bereit, die das Sichern und Wiederherstellen der aktuellen Transformationen erlauben.

3.8.11 Erstellen eines OpenGL-Programms

Die Benutzeroberfläche eines modernen Betriebssystems ist in der Regel so konzipiert, dass einer Applikation ein oder mehrere Fenster zugeordnet sind, über die alle Ein- und Ausgaben erfolgen. Die mit OpenGL erzeugten Grafiken müssten also auch in einem solchen Fenster erscheinen. Aufgrund der Plattformunabhängigkeit kann OpenGL keine Funktionen für das Management und die Konfiguration von Fenstern enthalten. Der Programmierer muss also für folgende Dinge selbst sorgen:

- Festlegung von Größe und Position des Ausgabefensters.
- Zuordnung der gewünschten Ressourcen zum Fenster (Bit pro Pixel, Buffer).
- Spezifizierung des Ausgabemodus.
- Öffnen und Schließen von Fenstern.
- Auswahl und Abfrage des aktuellen Fensters.
- Behandlung von Ereignissen

Nachdem das Fenster initialisiert und OpenGL eine Zeichenfläche mitgeteilt wurde ist der erste echte OpenGL-Funktionsaufruf die Anweisung

```
glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
```

Diese Funktion löscht das Fenster und initialisiert mit der angegebene Farbe. Der Prototyp lautet:

```
void glClearColor(GLfloat rot, GLfloat grün,
                  GLfloat blau, GLfloat alpha);
```

Beispiel: Jede Komponente kann 32bit Gleitkommazahlen zwischen 0 und 1 annehmen. Auf die Komponente alpha wird später eingegangen. Sie kann als Maß für die Lichtdurchlässigkeit gesehen werden. Mit dem obigen Kommando wird genau genommen das Löschen des Fensters erst vorbereitet. Das aktuelle Löschen erfolgt mit:

```
glClear(GL COLOR BUFFER BIT);
```

Mit den Befehlen

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

wird eine simple Transformation (siehe auch 3.8.10) erstellt.

Nun beginnt das eigentliche Zeichnen, in unserem Beispiel wird ein simples Dreieck erzeugt:

```
glBegin(GL_POLYGON);
 glColor3f(1.0, 0.0, 0.0);
 glVertex3f(-0.75, -0.75, 0.0);
 glColor3f(0.0, 1.0, 0.0);
 glVertex3f(0.75, -0.75, 0.0);
 glColor3f(1.0, 1.0, 1.0);
 glVertex3f(0.0, 0.75, 0.0);
glEnd();
```

Der Befehl

```
glFlush();
```

ist der letzte OpenGL-Funktionsaufruf. Er bewirkt, dass alle noch nicht ausgeführten OpenGL-Kommandos zur Ausführung gelangen.

3.8.12 Geometrische Primitive

Als hardwareorientierte Graphikbibliothek konzentriert sich OpenGL neben Punkten und Linien auf Primitive, die zur einfacheren Berechnung unmittelbar in Dreiecke zerlegbar sind. Es stehen insgesamt zehn derartige geometrische Objekte zur Verfügung: Punkte, Strecken, offene Streckenzüge, geschlossene Streckenzüge, Dreiecke, Dreiecksstreifen, Dreiecksfächer, Vierecke, Vierecksstreifen sowie beliebige konvexe Polygone. Die Verwendung von Streifen und Fächern hat dabei den Vorteil, dass gemeinsame Eckpunkte mehrerer Polygone nur ein einziges Mal transformiert werden müssen. Beim Beginn einer Objektdefinition wird der Funktion glBegin der Typ des gewünschten Objekts übergeben, z.B. GL_TRIANGLES. Darauf folgen Aufrufe von glVertex zur Spezifikation der Eckpunkte, denen Änderungen der aktuellen Farbe, Normalen oder Texturkoordinaten vorangehen können, die dann auf diesen Punkt Einfluss haben. Das Ende wird durch die Funktion glEnd gekennzeichnet.

Weiterhin lassen sich mit OpenGL auch gekrümmte Oberflächen erzeugen, die aber völlig anders gehandhabt werden, wie die soeben beschriebenen. Da sie im vorliegenden Projekt nicht zum Einsatz kamen, sollte sie der geneigte Leser in einer der zahlreichen OpenGL-Dokumentationen nachlesen.

3.8.13 Vertex Arrays

Müssen viele Attribute für jeden Punkt geändert oder einzelne Punkte als Begrenzung mehrerer Primitive wiederholt angesprochen werden, so ist ein Funktionsaufruf für jeden Parameter unter Umständen recht ineffizient. Deshalb wurden mit der Version 1.1 von OpenGL Vertex Arrays eingeführt. Dabei handelt es sich um Felder mit beispielsweise Farben, Normalen, Texturkoordinaten und Punktkoordinaten für beliebig viele Punkte. Diese werden vor den Beschreibungen der Primitive übergeben und ihre Elemente anschließend über einen Index angesprochen.

Dabei genügt dann ein einziger Aufruf von glArrayElement, um alle aktiven Vertex Arrays auszulesen, wodurch dann z. B. neben den Punktkoordinaten auch die Texturkoordinaten und die Normale an diesem Punkt gleichzeitig gesetzt werden. Andere Attribute bleiben unbeeinflusst, so dass für ein einfarbiges Objekt kein Array mit identischen Farbdaten gefüllt werden muß, sondern wie gewohnt auf die mittels glColor gesetzte aktuelle Farbe zurückgegriffen wird.

Unter Verwendung von glArrayElement lässt sich die Zahl der Funktionsaufrufe bereits auf einen pro Punkt verringern, bei Objekten, die neben den Punktkoordinaten auch noch Farben, Normalen und Texturkoordinaten an jedem Punkt benötigen immerhin nur noch ein Viertel der ursprünglichen Aufrufe. Besteht das Objekt nur aus einer einzigen Art von geometrischen Primitiven, so ist es sogar möglich, es komplett mit nur einem einzigen Aufruf zu beschreiben. Die Funktion glDrawElements nimmt hierzu neben dem Typ des Primitivs ein weiteres Array als Parameter entgegen, welches die Indizes der Punkte enthält. Dabei werden die Aufrufe zum Setzen und Aktivieren der Vertex Arrays aber ebensowenig berücksichtigt, wie der Aufbau des Arrays selbst. Müssen die Felder also ausschließlich zur Verwendung als Vertex Arrays angelegt werden, ist nicht unbedingt ein Vorteil gegenüber den direkten Funktionsaufrufen zu erwarten.

3.8.14 Display Listen

Um mehrfach vorkommende Funktionssequenzen einzusparen, besteht eine andere Möglichkeit. Öffnet man eine Display Liste, so werden nachfolgende Funktionsaufrufe in dieser Liste gespeichert. Nach ihrem Schließen kann man sie dann beliebig oft mit einem einzigen Aufruf von glCallList abarbeiten lassen. So lassen sich z. B. komplexere Objekte aus einfachen Primitiven zusammenbauen. Selbst ganze Szenen kann man so aufzeichnen, um sie etwa nach einer Änderung der Betrachterposition erneut abzuspielen. Neben diesen Annehmlichkeiten in der Handhabung von Display Listen haben sie auch auf die Rechengeschwindigkeit einen großen Einfluss. So muss der erzeugende Programmteil, der die Parameter der gespeicherten Funktionen möglicherweise aufwendig berechnen muss, nicht wiederholt durchlaufen werden. Vor allem aber können die Listen bereits bei der Erzeugung von OpenGL optimiert werden, beispielsweise ließen sich mehrere aufeinanderfolgende Transformationen in einer einzigen Transformationsmatrix zusammenfassen. Auf der anderen Seite haben Display Listen den Nachteil, daß sie sehr viel Speicher belegen können, was im Extremfall, wenn Hauptspeicher ausgelagert werden muß, den erhofften Gewinn wieder zunichte machen kann. Ein sehr einfaches Beispiel für die Anwendung von Display Listen ist ein rotierendes Objekt. Angenommen, in der Liste Nummer 1 seien alle beschreibenden Funktionen für einen Würfel enthalten. Dann würde schon der folgende Programmcode genügen um eine Animation zu erzeugen:

```
for(;;)
{
```

```
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glRotated(5.0, 0.0, 1.0, 0.0);
   glCallList(1);
}
```

3.9 Die Bibliothek CV97

CYBERVRML97 für C++ ist eine Entwicklungsbibliothek für VRML97/2.0 Applikationen. Bei der Benutzung dieser Bibliothek kann man VRML-Dateien (siehe unten) lesen und schreiben. Ferner ist das Setzen und Auslesen von Szenen/Graph-Information, Zeichnen der Geometrien und die Verhaltenskontrolle oder *behaviors*, möglich. Diese Bibliothek kann aus dem Internet heruntergeladen werden (z.B. unter http://www.cybergarage.org/vrml/cv97/cv97cc/download/ (11.09.2001)), liegt aber dem Source des Projektes auch bei.

3.9.1 VRML

VRML ist die Abkürzung für “Virtual Reality Modeling Language”. Auch hierbei handelt es sich um ein sehr populäres Format. Selbst wenn die verwendete 3D-Software nicht einen Export nach VRML unterstützen sollte, so gibt es immer noch eine Fülle an frei erhältlichen Programmen, die zwischen VRML und anderen Standard-Formaten konvertieren können. VRML ist nicht nur zur Beschreibung von Geometrien ausgelegt. Man soll damit virtuelle Szenen beschreiben können, was ebenso Materialeigenschaften, Beleuchtungseigenschaften, Audio-Informationen sowie Interaktionsmöglichkeiten eines virtuellen Betrachters mit der Szene umfasst. Trotzdem ist es kein Problem, dieses Format in eingeschränkter Funktionalität zu nutzen. Beispielsweise werden die Glieder unserer Roboter ebenfalls Materialeigenschaften besitzen. Diese werden mittels unserer eigenen Roboterbeschreibungssprache spezifiziert, während wir das VRML-Format lediglich zum Import von Geometrien nutzen. Es existieren drei Versionen von VRML: VRML 1.0, VRML 2.0 und VRML97. Viele Programme wie z.B. das von uns verwendete Paket Blender können lediglich nach VRML 1.0 exportieren. Allerdings gibt es frei erhältliche Konvertierungsprogramme, die VRML 1.0-Dateien nach VRML 2.0 exportieren. Diese Version ist wiederum größtenteil kompatibel zu VRML97. Bei letztgenannter Version handelt es sich um einen ISO-Standard.

3.10 Die Bibliothek PVM

Parallele Verarbeitung, eine Methode, in der *kleine*, in der Regel unabhängige Tasks ein *großes* Problem lösen, ist eine der Schlüsselmethoden der modernen Computertechnologie. Algorithmen, die eine parallele Verarbeitung unterstützen, können durch den Einsatz einer parallelen Verarbeitung einen enormen Performancegewinn verbuchen. Da sich populationsbasierte Verfahren wie die Evolutionären

Algorithmen und damit auch die Genetische Programmierung sehr gut für Parallelisierungen eignen, soll an dieser Stelle ein Tool, welches eine Parallelisierung in einem heterogenen Netzwerk unterstützt, vorgestellt werden: PVM, was soviel heißt wie Parallel Virtual Machine. Auch an dieser Stelle kann nur eine kompakte Einführung mehr im Sinne einer Vorstellung dieses (komplexen) Tools gegeben werden. Zur Vertiefung sei weiterführende Literatur wie [26] empfohlen.

Einführung

PVM 3 ist ein Software-System, welches erlaubt ein heterogenes (UNIX-) Computernetzwerk als einen *Parallelrechner* zu verwenden. Die Entwicklung der PVM startete im Sommer 1989 am Oak Ridge National Laboratory. Die Entwicklung erfolgte in der Programmiersprache C. Unter PVM erscheint eine Benutzer-definierte Ansammlung von seriellen, parallelen oder sogar Vektor-Computern als ein einziger großer *speicher-verteilter* Computer. Die Gesamtheit aller Rechner, also der simulierte Parallelrechner, soll im Folgenden *virtueller Rechner* genannt werden, während ein Mitglied dieser Computersammlung als Host bezeichnet wird. PVM unterstützt

- das automatische Starten von Tasks auf dem virtuellen Rechner
- die Kommunikation zwischen den Tasks
- die Synchronisation von Tasks untereinander

Ein *Task* ist im Rahmen der PVM analog zu dem eines UNIX Tasks als Berechnungseinheit definiert. Oft ist es sogar ein UNIX Prozess, das ist jedoch nicht notwendigerweise so. Applikationen, welche in Fortran77 oder C geschrieben werden, können durch die Verwendung der *Message-Passing-Methode* parallelisiert werden, in der multiple Tasks einer Anwendung durch das Empfangen und Senden von Nachrichten (sog. *Messages*) kooperieren, um ein Problem parallel zu lösen. PVM unterstützt Heterogenität sogar auf der Ebene der Applikation, des Rechners und des Netzwerks.

Es erlaubt dadurch die bestmögliche Anpassung an das Problem und behandelt (wenn notwendig) die Datenkonvertierung, wenn z.B. zwei Rechner unterschiedliche Floating Point oder Integer Repräsentationen voraussetzen. Wie bereits oben erwähnt, unterstützt PVM die Verbindung unterschiedlicher Netzwerktypen.

3.10.1 PVM Aufbau und Eigenschaften

PVM besitzt also im Wesentlichen die folgenden Eigenschaften:

- Benutzerkonfigurierbarer Host-Pool
- Prozessbasierte Berechnungen
- Explizites Message-Passing-Modell

- Heterogene Unterstützung
- Multiprozessor-Unterstützung

Das PVM-System besteht im Wesentlichen aus zwei Komponenten:

- Daemon, genannt *pvmd3*, jedoch meist abgekürzt durch *pvmd*.
  Dieser Daemon muss vor dem Start einer PVM Anwendung gestartet werden. Dieser startet (selbstständig) auf jedem Rechner, der als Host des virtuellen Rechners registriert ist, ebenfalls diesen Daemon, wodurch eine PVM Anwendung von jedem PVM Host aus gestartet werden kann.
- PVM Library (libpvm3.a), welche die Funktionsschnittstellen der PVM Methoden enthält, u.a. Routinen für
  - Message Passing
  - das sogenannte „spawning“ von Prozessen
  - die Koordination von Tasks
  - die Modifikation des virtuellen Rechners

  Applikationen müssen mit dieser Bibliothek gelinkt werden, um PVM in dieser Applikation nutzen zu können.

PVM (3) bietet unter anderem die folgenden (detaillierteren) neuen Funktionalitäten (bezogen auf PVM (2), welche in [26] ausführlich erläutert werden):

- Im Gegensatz zu PVM v2.x ein verbessertes User Interface
- Integer Task Identifier
- User Process Control
- Erweiterte Fehlertoleranz
- Signaling
- Erweiterte Kommunikation der Tasks
- Unterstützung von Multiprozessor-Systemen

Auf die Installationsanleitung soll an dieser Stelle verzichtet werden, sie wird jedoch in [26] sehr ausführlich beschrieben.

3.10.2 Start und Konfiguration von PVM

Der Start von PVM schließt die Ausführung von *pvm* oder *pvmd3* mit ein. Ohne Argumente wird pvmd3 auf dem lokalen Host durchgeführt. Die PVM-Konsole ermöglicht das interaktive Hinzufügen oder Löschen von Hosts auf einem virtuellem Rechner, ebenso wie das Starten oder Löschen von PVM-Prozessen. Die PVM Konsole empfängt den Benutzer mit dem folgenden Prompt:

```
pvm>
```

Viele Kommandos können nun ausgeführt werden. Eine Auswahl sei im Folgenden erwähnt:

- help oder ?
  Information über jeden interaktiven Befehl. Diesem Kommando kann auch der explizite Name eines Kommandos folgen, über den weitere Informationen erwünscht sind.
- conf
  Listet die aktuelle Konfiguration des virtuellen Rechners auf mit Hostname, pvmd Task ID, Architekturtyp, ...
- add
  Gefolgt von einem (oder mehreren) Rechnernamen, fügt einen oder mehrere Rechner dem virtuellen Rechner hinzu.
- delete
  Gefolgt von einem (oder mehreren) Rechnernamen, löscht einen oder mehrere Rechner aus dem virtuellen Rechner.
- spawn
  Startet eine PVM Applikation

Es gibt natürlich noch weitaus mehr Kommandos, die in der Konsole ausgeführt werden können. Diese Befehle werden ebenfalls in [26] vorgestellt.

Beim Start von pvmd3 kann ein Konfigurationsfile als Argument übergeben werden. Diese Datei listet alle die Hosts auf, welche den virtuellen Rechner definieren. Alternativ können alle Hosts über das *add* Kommando sequentiell dem virtuellen Rechner über die Konsole hinzugefügt werden. Nur ein Benutzer muss PVM installieren, aber jede PVM sollte ihr eigenes Hostfile besitzen. Das einfachste zu verwendende Hostfile enthält nur die Hostnamen und kann das folgende (einfache) Aussehen haben, wobei Blanks ignoriert und mit # eingeleitete Zeilen als Kommentar interpretiert werden:

```
#My used Configuration
```

```
MySun
sun1
sun2
sun3
```

Der erste Hostname muß der Name des lokalen Hosts sein. Es folgen die weiteren zu verwendenden Hosts. Dem Hostnamen können weitere (optionale) Optionen folgen (s. [26]), welche ebenfalls als Standard definiert werden können.

Die Standardausgabe eines PVM Programms erfolgt nach /tmp/pvml.<uid> auf dem Host, auf dem PVM gestartet wurde. Wenn die Ausgaben auf dem Bildschirm erfolgen sollen, so kann tail -f /tmp/pvml.<uid> ausgeführt werden, wodurch *stdout* und *stderr* auf dem Bildschirm für aktuelle PVM Tasks erscheinen.

3.10.3 Die Benutzerschnittstelle

Die Schnittstelle wird ausführlich in [26] beschrieben. Eine detaillierte Beschreibung in Form einer Auflistung aller Befehle ohne direkten Kontext wäre an dieser Stelle nicht besonders sinnvoll, da dies eine detailliertere Vorstellung des PVM Tools voraussetzt. Daher wird im Folgenden eher auf mögliche Grundsätze im allgmeinen Umgang mit PVM eingegangen.

3.10.4 Allgemeine (Geschwindigkeits-) Betrachtungen

Es bestehen keine Einschränkungen bzgl. eines Programmierparadigmas, welches ein PVM Benutzer ggf. wählt. Jede spezielle Kontroll- und Abhängigkeitsstruktur kann bei zweckmäßigem Einsatz der PVM implementiert werden, aber der Entwickler sollte sich bei der Benutzung der Message-Passing-Methode über einige Dinge im Klaren sein:

3.10.5 Task Granularity

Dieser Wert wird typischerweise als Verhältnis der Anzahl von empfangenen Bytes (durch einen Prozess) zur Anzahl der berechneten Floating Point Operationen berechnet. Durch einige einfache Berechnungen bzgl. der Berechnungsgeschwindigkeit einzelner Hosts und der Netzwerkverbindungen zwischen den Hosts kann bereits eine grobe Schwelle errechnet werden, die als Maß dienen kann. Je höher die Granularität, je höher ist die Verarbeitungsgeschwindigkeit, was leider in den meisten Fällen eine Reduktion des Parallelismus zur Folge hat. An dieser Stelle muß ein Trade-Off gefunden werden.

3.10.6 Anzahl der gesendeten Nachrichten

Die Anzahl der empfangenen Bytes könnte in vielen kleinen Nachrichten, oder ebenfalls in einer Anzahl von kleineren Nachrichtenpaketen gesendet werden. Solange kleine Nachrichtenpakete verwendet werden, reduziert dies die Messagever-

arbeitungszeit, was aber nicht die Gesamtverarbeitungszeit reduzieren muss. Es besteht die Möglichkeit, Kommunikation mit Berechnungen zu *überdecken*, was natürlich eine Berechnung der Anzahl von Nachrichten, die überdeckt werden können, voraussetzt.

3.10.7 Funktionale oder datenbezogene Parallelität

Es ist durch den Benutzer von PVM zu überlegen, worauf sich die Parallelität beziehen soll:

- Bei der *funktionalen Parallelität* wird die parallele Eigenschaft auf die *Aufteilung* der Funktionen bezogen, indem jeder Host eine (funktional) andere Aufgabe berechnet, aber vielleicht mit den selben Daten.
- Bei der *datenbezogenen Parallelität* werden die Daten partitioniert und auf alle Hosts verteilt. Die Operationen (oftmals gleich) werden auf jede Teilmenge der entsprechenden Hosts ausgeführt, wobei einzelne Prozesse durchaus miteinander kommunizieren dürfen.

In PVM können alle Ansätze in Form eines hybriden Ansatzes gemischt werden, eine Festlegung auf einen Ansatz ist nicht notwendig.

3.10.8 Netzwerkspezifische Betrachtungen

Natürlich sollte der Benutzer auch bedenken, dass das PVM-Programm - im Falle eines Einsatzes im Netzwerk - auch andere Netzwerkbenutzer (geschwindigkeitstechnisch) beeinflussen kann:

1. Jedem Host stehen unterschiedliche Speicher- und CPU-Ressourcen zur Verfügung.
2. Je nach Netzwerktechnologie kann die Länge einer Nachricht einen unterschiedlichen Einfluss auf den Netzwerkdurchsatz haben.
3. Die Performance eines Netzwerkes ist dynamischer Natur. Der Durchsatz kann je nach Auslastung und Art der Verwendung (teilweise sogar stark) schwanken.

Daher ist eine gute Balance bzgl. des Daten-Durchsatzes für die vorteilhafte Anwendung von PVM anzuraten.

3.11 Das Werkzeug CONCURRENT VERSION SYSTEM

Bei einem Projekt unserer Größenordnung macht eine Versionsverwaltung selbstverständlich Sinn. Möglichkeiten wie die Archivierung älterer Versionen von Dateien, “tagging” von Dateiversionen sowie eine Zugriffskontrolle um gleichzeitiges

Editieren mehrerer Entwickler an einer Datei zu vermeiden hielten wir für wichtig(vgl. auch [27], [28],[29] und [30]). Allerdings fanden wir es wünschenwert, die CVS-Funktionalität aus folgenden Gründen nur in spezieller, eingeschränkter Weise zu nutzen:

- CVS Befehle haben eine umständliche Form, da sie alle mit `cvs` beginnen und die genaue gewünschte Aktion wie z.B. `update` als Parameter erhalten.

- Normalerweise ist es CVS-Benutzern gestattet, parallel an der gleichen Datei zu arbeiten. Arbeiten also beispielsweise zwei Programmierer *A* und *B* an der gleichen Datei `code.cpp`, und führt *A* zuerst ein `cvs commit code.cpp` aus, erhält *B* bei seinem anschließenden `commit` einen Fehler und muss zuerst `cvs update code.cpp` ausführen. Dabei werden die Versionen von *A* und *B* vermischt (merge).

  Da PG-Teilnehmer damit schlechte Erfahrungen gemacht haben, weil offensichtlich die Semantik von Programmcode auch an unterschiedlichen Stellen voneinander abhängig sein kann, sollte dieser Fall vermieden werden!

  CVS bietet allerdings zum einen die Möglichkeit, durch Anwenden des Befehls `cvs edit code.cpp` sich selber in eine Liste von Editoren einzutragen, die momentan diese Datei bearbeiten. Diese lässt sich durch den Befehl `cvs editors` anzeigen. Zusätzlich besteht die Möglichkeit, bei Ausführen von `cvs edit` eine spezifizierte Aktion automatisch ausführen zu lassen (z.B. automatische Benachrichtigung aller anderen Entwickler per email: “Markus Muster bearbeitet zur Zeit die Datei code.cpp!”). Durch `unedit` wird man aus der Liste der Editoren entfernt. Zusätzlich ermöglichen diese Befehle ein Aktivieren/Deaktivieren von Schreibrechten dieser Datei in der lokalen Arbeitskopie.

  Die Übersicht über die momentanen Editoren scheint sehr praktisch. Die automatische Benachrichtigung per email hielten wir für keine gute Idee. Außerdem ist man letztendlich doch von der verantwortungsvollen Nutzung dieser Funktionen durch die Benutzer abhängig.

  CVS bietet dafür die Möglichkeit an, einen Lock auf eine Datei zu setzen. Dies geschieht durch `cvs admin -l code.cpp`. Dieser Befehl ist jedoch nur dann erfolgreich, wenn kein anderer Benutzer bereits einen Lock auf dieser Datei hat. Durch `cvs admin -u code.cpp` gibt man einen Lock wieder frei.

  Es ist nun wünschenswert, `commit`/`update`, `edit` und Locks miteinander zu verbinden, um den Datenaustausch zwischen CVS-Archiv und lokaler Arbeitskopie, Übersicht über die Editoren von Dateien und exklusive Dateischreibrechte gemeinsam nutzen zu können.

- Manche Befehle sollen nur in Kombination mit anderen oder speziellen Parametern genutzt werden.

Aus diesen Gründen haben wir Shellskripte erstellt, die diese spezialisierte Funktionsweise bereitstellen, und uns darauf geeinigt, wesentliche Operationen auf dem CVS nur noch über sie abzuwickeln. Im folgenden werden die wichtigsten Skripten kurz beschrieben.

lock: Updated die lokale Version einer Datei, trägt den Anwender in die Liste der Editoren ein und versieht die lokale Arbeitskopie mit Schreibrechten. Außerdem erhält man einen Lock auf diese Datei, so dass kein anderer erfolgreich das `lock` Skript auf diese Datei anwenden kann.

unlock: Das Gegenstück zu `lock`. Stellt die neue, lokal veränderte Version der Datei ins CVS und macht die Arbeitskopie schreibgeschützt. Entfernt den Anwender von der Liste der Editoren. Entfernt den Lock auf die Datei.

add: Fügt eine neue Datei dem CVS hinzu. Die Datei muss schon vorhanden sein. Das Skript veranlasst lediglich, dass sie auch über das CVS verwaltet wird.

Dabei darf es sich nicht um ein Verzeichnis handeln. Dafür gibt es den Befehl `diradd` (siehe unten).

remove: Entfernt eine Datei aus dem CVS. Verzeichnisse dürfen nur entfernt werden, wenn sie leer sind. In diesem Fall werden sie automatisch bei einem `update` entfernt.

diradd: Fügt ein neues Verzeichnis dem CVS hinzu. Auch hier gilt, dass das Verzeichnis schon vorhanden sein muss.

update: Aktualisiert die lokale Arbeitskopie der Datei entsprechend der aktuellen Version im CVS.

undo: Wirkt im Prinzip wie unlock, mit dem Unterschied, dass die eigenen Änderungen an der Datei rückgängig gemacht werden.

release: Erhält als Argument einen Modulnamen. Führt `unlock` auf alle Dateien des Moduls aus und löscht die lokale Arbeitskopie.

checkout: Erhält als Argument einen Modulnamen. Erstellt eine lokale Arbeitskopie dieses Moduls.

Bis auf die Skripte `release` und `checkout` erhalten alle einen Dateinamen als Argument.

3.12 Das Werkzeug MAKE

Im folgenden wird das Werkzeug Make vorgestellt und Gründe genannt, warum wir es für unser Projekt benutzen.

3.12.1 Was ist Make

Make ist ein Werkzeug, das typischerweise für größere Programmierprojekte benutzt wird, um automatisch entscheiden zu lassen, welche Teile bei Änderung des Quellcodes neu kompiliert oder gelinkt werden müssen (siehe [31]).

Im Allgemeinen werden aus vielen Quelldateien durch einen immergleichen Prozess wenige Ergebnisdateien erzeugt. Vielfach ist man auch nur an einer einzigen Datei von den Erzeugten interessiert. Dies gilt nicht nur für das Übersetzen von Programmen, sondern auch für die automatische Erzeugung von Programmcode durch Werkzeuge wie Bison, moc (QT) und uic (ebenfalls QT), oder mit LaTeX erzeugte Dokumente.

Es ergibt sich dabei ein Baum von Abhängigkeiten zwischen den beteiligten Dateien. Abhängigkeit bedeutet in diesem Zusammenhang, dass eine Veränderung einer Datei bewirkt, dass eine oder mehrere andere Dateien nicht mehr aktuell sind und erneut aus den Dateien, von denen sie abhängen, erzeugt werden müssen:

Eine `.dvi` Datei hängt beispielsweise von einer oder mehreren `.tex`-Dateien ab, sowie evtl. von `.eps`-Dateien und einer `.bib`-Datei.

Eine ausführbare Datei kann von mehreren `.o`-, `.a`- oder `.so`-Dateien abhängen. Diese können wiederum von mehreren `.cpp`- und `.h`-Dateien abhängen. Diese können wiederum von `.y`-, `.ui` oder wiederum von C++-Quelldateien abhängen.

Um die gewünschten Ergebnisdateien in einer aktuellen Version zu erhalten, könnte man jedesmal den kompletten Abhängigkeitsbaum in einer Art Postorder-Durchlauf bearbeiten. Dies würde im Programmierbeispiel bedeuten, dass man alle Quelldateien neu übersetzen müßte, um eine aktuelle ausführbare Datei zu erhalten. Dabei würde aber im allgemeinen Fall überflüssige Arbeit getan. *Make* kümmert sich selbständig darum, welche Dateien neu erzeugt werden müssen.

Ist die letzte Änderung einer Datei *A* neuer als die von einer Datei *B*, und ist *B* von *A* abhängig, so veranlasst *Make* automatisch die Neuerzeugung von *B*.

Diese Abhängigkeiten können nun in einem sogenannten *makefile* beschrieben werden. Außerdem werden dort Befehle aufgelistet, um eine Datei aus den Quelldateien, von denen sie abhängt, neu zu erzeugen.

3.12.2 Wozu benutzen wir Make

Wir benutzen eine spezielle Variante mit dem Namen *GNU make* für unser Projekt. Bei dem Projekt bestehen folgende Anwendungsfälle bzw. Abhängigkeitsverhältnisse zwischen Dateien, die ein Werkzeug wie *Make* unverzichtbar machen:

- Wir programmieren in C++. Das Ergebnis einer Kompilierung ist meistens eine `.o`-Objektdatei. Eine `.cpp`-Quellcodedatei bindet mindestens eine `.h`-Headerdatei ein. So ist eine `.o`-Datei mindestens von zwei anderen Dateien abhängig. Bindet der Programmierer weitere Headerdateien ein, können sich diese Abhängigkeiten ändern.

- Wir benutzen den QT Designer. Mit diesem Werkzeug lassen sich graphische Oberflächen erstellen. Das Speicherformat dieses Programms hat die Endung `.ui`.

  Mit dem per Kommandozeile gesteuerten Programm `uic` (User Interface Compiler) werden diese Dateien in C++-Quelldateien und -Headerdateien umgewandelt. Diese hängen also von den entsprechenden `.uic`-Dateien ab.

- Will man bestimmte Möglichkeiten von QT nutzen, müssen einige Klassendefinitionen das Makro `Q_OBJECT` enthalten. Außerdem müssen die Headerdateien, die diese Definitionen enthalten, von dem sogenannten Meta Object Compiler (moc) eingelesen werden, der eine weitere C++-Quelldatei ausgibt. Diese muß ebenfalls kompiliert und zur letztendlichen ausführbaren Datei gelinkt werden. Es handelt sich also wieder um eine Abhängigkeit.

Dies sind die Hauptabhängigkeiten unseres Projektes. Wir brauchen uns nicht mehr bei jedem Kompiliervorgang um sie zu kümmern, da *Make* das für uns erledigt.

Wie oben bereits erwähnt, können sich aber auch die Abhängigkeiten leicht beim Programmieren ändern, wenn sich die Menge der in eine Quelldatei eingebundenen Headerdateien ändert. Dort helfen spezielle Optionen des von uns verwendeteten GNU C++ Compilers, die für eine Quelldatei automatisch die Abhängigkeiten in einem von *Make* verarbeitbaren Format ermittelt.

Zu fast jeder C++-Quelldatei wird auf diese Weise eine gleichnamige Datei mit der Endung `.d` erzeugt, die diese Abhängigkeiten enthält. Diese werden ins *makefile* eingefügt.

Außerdem enthält unser *makefile* Möglichkeiten, um automatisch eine Klassendokumentation mit Hilfe des Werkzeugs *Doxygen* zu erzeugen, temporäre Dateien zu entfernen oder spezielle Testprogramme zu kompilieren, die jedem Teilnehmer der PG zur Verfügung stehen.

Kapitel 4

Das System SIGEL

In diesem Kapitel wird das System SIGEL beschrieben und auf einzelne wichtige Teile des Systems genau eingegangen. Eingangs geben wir einen Überblick über die Architektur des Systems unter dem Abschnitt 4.1. Unter dem Abschnitt 4.2 werden die Repräsentation und das Einlesen von Robotermodellen vorgestellt, es wird die Roboterbeschreibungssprache und die Idee, die dahinter steht genau erklärt. Die Sprache für die Robotersteuerungsprogramme wird unter dem Punkt 4.3 näher erläutert. Im Abschnitt 4.4 stellen wir den Dynamiksimulator vor, den wir zur Berechnung der Fitness und zur Visualisierung benötigen. Mit dem in Abschnitt 4.5 beschriebenen GP-System werden unsere Programme evolviert und mit den in Abschnitt 4.6 genauer erklärten Fitnessfunktionen bewertet.

4.1 Architektur

Die Architektur des SIGEL-Systems besteht aus mehreren Teilen, die die Bearbeitung der einzelnen Teilaufgaben der Simulation genetisch evolvierten Laufrobotersteuerungsprogrammen übernehmen. Es folgt nun ein Überblick, der kurz alle Komponenten des SIGEL-Systems beschreibt. In den darauf folgenden Unterkapiteln werden dann besondere Bereiche hervorgehoben und eingehender erklärt.

Das Erste, was den Benutzer des SIGEL-Systems nach dem Programmstart erwartet, ist die grafische Benutzeroberfläche. Ihre Programmdateien sind programmtechnisch in drei Namesräume aufgeteilt. Die Dateien der graphischen Benutzeroberfläche des Hauptprogramms sind in einem Namensraum zusammengefasst, so wie die des Dienstprogramms auch ihren eigenen Namensraum haben. Programmdateien, die Bestandteil beider Benutzeroberflächen sind, befinden sich im dritten Namensraum. Mit der graphische Benutzeroberfläche steuert der Benutzer des Programm, nimmt die nötigen Einstellungen bezüglich seiner Experimente vor und kann sich vom Erfolg seiner Evolution überzeugen. Die Namensräume der grafischen Benutzeroberfläche spiegeln die Teilung des SIGEL-Systems wieder. Es gibt ein Hauptprogramm und ein Dienstprogramm, welche beide verschiedene Funktionen ausüben. Das Hauptprogramm des SIGEL-Systems steuert die Dienst-

programme, verwaltet die Experimente des Benutzers und organisiert die genetische Evolution. Ein Dienstprogramme beinhaltet immer einen Simulator. Durch ihn kann das Dienstprogramm ein Robotermodell anzeigen oder einen Fitnesswert berechnen. Beide Zustände des Dienstprogramms unterscheiden sich hauptsächlich darin, ob die grafische Benutzeroberfläche zugeschaltet ist oder nicht, was nichts anderes bedeutet, als dass mit oder ohne Visualisierung simuliert werden soll.

Die Visualisierung der Simulation der genetisch evolvierten Laufrobotersteuerungsprogramme ist in die Benutzeroberfläche des Dienstprogrammes eingearbeitet. Programmtechnisch ist die Visualisierung natürlich ein eigenständiger Namensraum, der alle die Funktionen in sich vereint, die zum Darstellen eines simulierten Robotermodells benötigt werden. Diese Funktionen werden vom Dienstprogramm des SIGEL-Systems, je nach Anwendungsfall genutzt. So beinhalten sie nicht nur die Funktionen, die aus den errechneten Daten des Simulators eine grafische, dreidimensionale Darstellung machen, sondern auch Zusatzfunktionen. Diese werden dazu benutzt, um einzelne Sequenzen der Visualisierung abspeichern zu können und um die Visualisierung zu steuern.

Auch die Bestandteile einer Simulation sind in ihrem eigenen Namensbereich zusammengefasst. So gibt es einen Namensraum für die Simulationsumgebung, das Laufrobotersteuerungsprogramm, sowie für den Roboter und den Simulator selbst. Die Programmteile, die im Namensraum des Simulators zusammengefasst sind, realisieren nicht nur die kinematische Simulation, sondern auch die Simulation der Robotersteuereinheit, den Interpreter des Laufrobotersteuerungsprogrammes und den Datenaufzeichnungsfunktionen. Da der Simulator im Dienstprogramm des SIGEL-Systems enthalten ist, müssen alle relevanten Daten nach dem Aufruf des Dienstprogrammes vom Hauptprogramm an den Simulator weitergeben werden. Während jeder Namensraum, der Programmteile enthält, die Daten an die Simulation übertragen müssen, seine eigenen Datenübertragungsfunktionen beinhaltet, wurden die Funktionen, die das Robotermodell zur Übertragung vorbereiten, übertragen und wieder in eine dem Simulator gebräuchliche Form bringen, in einem eigenen Namensraum zusammengefasst.

Der Namensraum der Laufrobotersteuerungsprogramme umfasst die Funktionen, die zum Aufbau eines solchen Programmes gebraucht werden. Diese werden von verschiedenen Seiten genutzt. Der Simulator, der die Funktionen zum Interpretieren eines Laufrobotersteuerungsprogramms besitzt, muss den Namensraum der Roboterprogramme benutzen, um den Programmablauf des Steuerprogramms zu simulieren. Aber auch die automatische Erzeugung eines Laufroboterstuerungsprogramms, wie sie im GP-System des SIGEL-Systems passiert, benutzt Funktionen aus dem Namensraum der Roboterprogramme.

Natürlich hat das GP-System des SIGEL-Systems auch seinen eigenen Namesraum. In ihm werden die Programmteile zusammengefasst, die für den gesamten Aufbau und Ablauf der Genetischen Programmierung verantwortlich sind. Aber auch über die Standardfunktionen eines GP-Systems hinaus sind hier Programmteile zur Steuerung des parallelisierten Ablaufs der Evolution zu finden. Diese Funktionen realisieren die verteilte Fitnesswertberechnung auf mehreren

Computern.

Die weiteren Namensräume sind die des realen Roboterinterfaces und der Werkzeugfunktionen des SIGEL-Systems. Der Namensraum, der die Schnittstelle zum realen Roboter realisieren sollte, ist zwar vorhanden, aber nicht ausgefüllt, weil dieser sehr stark von der reale Roboterarchitektur abhängt. Leider ist kein realer Laufroboter als Beispiel aufgenommen worden, weil kein reales Robotermodell zur Betrachtung zur Verfügung stand. Zu den Werkzeugfunktionen zählen der Zufallszahlengenerator, die Ein-/Ausgabefunktionen und die Ausnahmebehandlung. Diese Funktionen werden von allen anderen Programmteilen benutzt und stellen Lösungen für Standardverfahren da, die der Einfachheit halber in diesem Namensraum zusammengefasst wurden.

4.2 Repräsentation und Einlesen von Robotermodellen

In diesem Kapitel geht es darum, wie die Roboterbeschreibung aus der Konfigurationsdatei eingelesen wird, wie die Modelle intern gespeichert werden und wie sie zwischen den vernetzten Rechnern transportiert werden.

4.2.1 Die Roboterbeschreibungssprache

In diesem Abschnitt wird die Roboterbeschreibungssprache von einem eher formalen Blickpunkt aus betrachtet. Für praktische Aspekte der Modellierung und Beschreibung mit dieser Sprache sei auf Kapitel 6.3 verwiesen.

Grundsätzlich berücksichtigt der Parser Groß- und Kleinschreibung. Reservierte Wörter werden grundsätzlich klein geschrieben. Zur Beschreibung der Grammatik wird die EBNF (erweiterte Backus-Naur-Form) verwendet. Die folgenden Regeln beschreiben die grundlegenden Spracheinheiten.

digit = ’0’ | . . . | ’9’ .

char = ’a’ | . . . | ’z’ | ’A’ | . . . | ’Z’ .

name = char { char | digit | ’_’ } .

number = [’-’] (’.’ digit { digit } |
digit { digit } [’.’ digit { digit }]) .

stringcontents = *concatenations of all ASCII-characters except quote, new line, and carriage return* .

string = ’“’ stringcontents ’´´’ .

filepath = string .

Leerzeichen sind die ASCII-Zeichen 9 (Tabulator), 10 (neue Zeile), 13 (Wagenrücklauf) und 32 (gewöhnliches Leerzeichen). Zwischen den Bestandteilen der oben angegebenen Regeln sind die Leerzeichen nicht erlaubt. Die in den folgenden Abschnitten aufgeführten Regeln basieren nicht mehr auf Zeichen, sondern auf Symbolen und benötigen zwischen zwei Symbolen mindestens ein Leerzeichen zur Trennung. Dies betrifft insbesondere Bezeichner und Zahlen. Symbole, die nur aus Sonderzeichen bestehen, lassen sich auch ohne Trennzeichen isolieren und benötigen i.A. weder vorher noch nachher ein Leerzeichen.

Die Angabe eines Dateinamens ist technisch eine Stringeingabe. Der Aliasname *filepath* dient nur der Erinnerung, dass dieser String an besondere, vom Betriebssystem vorgegebene syntaktische Regeln gebunden ist, die der Parser für Roboterbeschreibungen nicht überprüft.

4.2.1.1 Materialien

Eine Materialbeschreibung umfasst die physikalischen Daten und die Farbe für die grafische Darstellung. Die physikalischen Daten bestehen zum einen aus der Dichte, angegeben in Kilogramm pro Kubikmeter, den Reibungskoeffizienten und einem Elastizitätswert.

Der Elastizitätswert ist eine Hilfsgröße, die eine Dynamikbibliothek benutzen kann, wenn sie keine Coulomb'sche Reibung unterstützt. Sie soll die Erhaltung kinetischer Energie im Falle eines Stoßes angeben.

material = 'material' name '{' materialattribute '}' .

materialattributes = materialdensity ';' { materialfriction ';' '}'
[materialelasticity ';'] [materialcolour ';'] .

materialdensity = 'density' number .

materialfriction = 'friction' number 'on' name .

materialelasticity = 'elasticity' number .

materialcolour = 'colour' 'red' number 'green' number 'blue' number .

4.2.1.2 Glieder

Die geometrischen Daten eines Gliedes können über ein CAD-Programm oder über die Konfigurationsdatei eingeben werden. Bei der Benutzung eines CAD-Programms ist zu beachten, dass das VRML-Dateiformat dazu benutzt wird, die Daten vom CAD-Programm nach SIGEL zu bewegen. Die Dateien müssen darüber hinaus den Standard VRML'97 erfüllen und dürfen keine USE- oder DEF-Statements aufweisen. Ferner muss sichergestellt sein, dass alle Punkte von Polygonen gegen den Uhrzeigersinn, von außen betrachtet, definiert sind.

Die geschlossene Form wird mit einem Material ausgefüllt. Damit werden Dichte, Masse und äußere Erscheinung festgelegt. Ein Glied kann mit dem Schlüsselwort `torso` zum Rumpfglied erklärt werden. Ein so ausgezeichnetes Glied dient als Aufhänger zur Platzierung und Fitnessmessung. Die Benennung von Punkten relativ zum Glied dient der Verkettung von Gliedern über Gelenke. Ferner kann die Kollisionserkennung zwischen bestimmten Gliedern aufgehoben werden.

link = 'link' name '{' linkattributes '}' .

linkattributes = ['torso' ';'] linkgeometry ';' linkmaterial ';'
{ linkpoint ';' } [linknc ';'] .

linkgeometry = 'geometry' filepath .

linkmaterial = 'material' name .

linkpoint = 'point' name '=' '(' number ',' number ',' number ')' .

linknc = 'no_collision' name { ',' name } .

4.2.1.3 Gelenke

Die Syntax variiert ein wenig, je nachdem welchen der vier Gelenktypen Drehgelenk, Schubgelenk, Zylindergelenk oder Ungelenk man verwendet. Dreh- und Schubgelenk sind Gelenke mit einem, das Zylindergelenk mit zwei und das Ungelenk mit null Freiheitsgraden.

Grundsätzlich werden Achsen und Richtungen nicht durch Vektoren, sondern durch Punkte angegeben. Der Weg von einem Punkt zu einem anderen entspricht dann der Achse oder Richtung. Punkte werden auch nicht direkt als Koordinaten angegeben, sondern lediglich symbolisch. Die Symbole müssen in den Gliedern definiert sein. Die genaue Funktionsweise der Berechnung der relativen Initialpositionen, welche durch die Spezifikation der Gelenke bedingt ist, entnehmen Sie bitte Abschnitt 6.3.4.2.

joint = 'joint' (rotatejoint — translatejoint — cylinderjoint) | 'glue' unjoint .

Drehgelenke Bei Drehgelenken können Glieder an einer gemeinsamen Achse gegeneinander rotieren.

rotatejoint = 'rotational' name
'{' rotateconsts ';' rotateminimal ';'
rotatemaximal ';' rotateinitial ';' '}' .

rotateconsts = 'between' name '(' name ',' name ',' name ')'
'and' name '(' name ',' name ',' name ')' .

rotateminimal = ’minimal’ number .

rotatemaximal = ’maximal’ number .

rotateinit = ’init’ number .

Schubgelenke An Schubgelenken können die beteiligten Glieder an einer gemeinsamen Achse entlang verschoben, aber nicht verdreht werden.

translatejoint = ’translational’ name
’{’ translateconsts ’;’ translateminimal ’;’
translatemaximal ’;’ translateinit ’;’ ’}’ .

translateconsts = ’between’ name ’(’ name ’,’ name ’,’ name ’)’
’and’ name ’(’ name ’,’ name ’,’ name ’)’ .

translateminimal = ’minimal’ number .

translatemaximal = ’maximal’ number .

translateinit = ’init’ number .

Zylindergelenke Das Zylindergelenk ist eine Kombination von Drehgelenk und Schubgelenk. Die beteiligten Glieder können entlang der selben Achse verschoben und verdreht werden.

cylinderjoint = ’cylindrical’ name
’{’ cylinderconsts ’;’ cylinderrotate cylindertranslate ’}’ .

cylinderconsts = ’between’ name ’(’ name ’,’ name ’,’ name ’)’
’and’ name ’(’ name ’,’ name ’,’ name ’)’ .

cylinderrotate = ’minimal_rot’ number ’;’ ’maximal_rot’ number ’;’
’init_rot’ number ’;’ .

cylindertranslate = ’minimal_trans’ number ’;’ ’maximal_trans’ number ’;’
’init_trans’ number ’;’ .

Ungelenke Ungelenke sind technisch gesehen Gelenke, aber aufgrund ihres Charakters, keine Bewegung zuzulassen, wurde die Syntax etwas abweichend gestaltet.

unjoint = name ’{’ unjointconsts ’;’ ’}’ .

unjointconsts = ’between’ name ’(’ name ’,’ name ’,’ name ’)’
’and’ name ’(’ name ’,’ name ’,’ name ’)’ .

4.2.1.4 Antriebe

Ein Gelenk kann mit einem Antrieb versehen werden. Es kann dabei festgelegt werden, welche Kraft der Motor maximal und minimal ausüben kann wenn er in Betrieb ist, diese sind als absolute Beträge der Kraft in beide Richtungen anzusehen und sollten deshalb nicht negativ sein. Ferner ist festzulegen, wie der Wert des Steuerprogramms zu interpretieren ist, also als absolute Positionsangabe, als relative Positionsangabe oder als Kraftangabe.

drive = 'drive' name '{' driveattributes '}' .

driveattributes = drivejoint ';' driveminf ';' drivemaxf ';' .

drivejoint = ('force' | 'relativ' | 'absolute') name .

driveminf = 'minimalforce' number .

drivemaxf = 'maximalforce' number .

4.2.1.5 Sensoren

Unterscheiden kann man zwischen Gelenksensoren, die die Stellungen der Gelenke messen und damit den Roboter über seinen eigenen Zustand informieren, und sonstige Sensoren, die die Umwelt des Roboters erfassen. Implementiert sind derzeit nur Gelenksensoren.

sensor = 'sensor' name '{' sensorattributes '}' .

sensorattributes = jointsensorattr .

Gelenksensoren Ein Gelenksensor wird mit einem Gelenk verbunden und funktioniert dann ohne weiteres Zutun.

jointsensorattr = 'joint' name ';' .

4.2.1.6 Oberflächenform

Neben der Eingabe über eine VRML-Datei besteht die Möglichkeit, geometrische Daten über die Oberfläche von Gliedern direkt in die Konfigurationsdatei einzubetten. Dies wurde nachträglich eingefügt, um eine Möglichkeit zu schaffen, simple Oberflächenformen direkt ohne CAD-Programm einzugeben.

surface = 'surface' 'for' filepath { polygon } ';' .

polygon = '(' point { '/' point } ')' .

point = number ',' number ',' number .

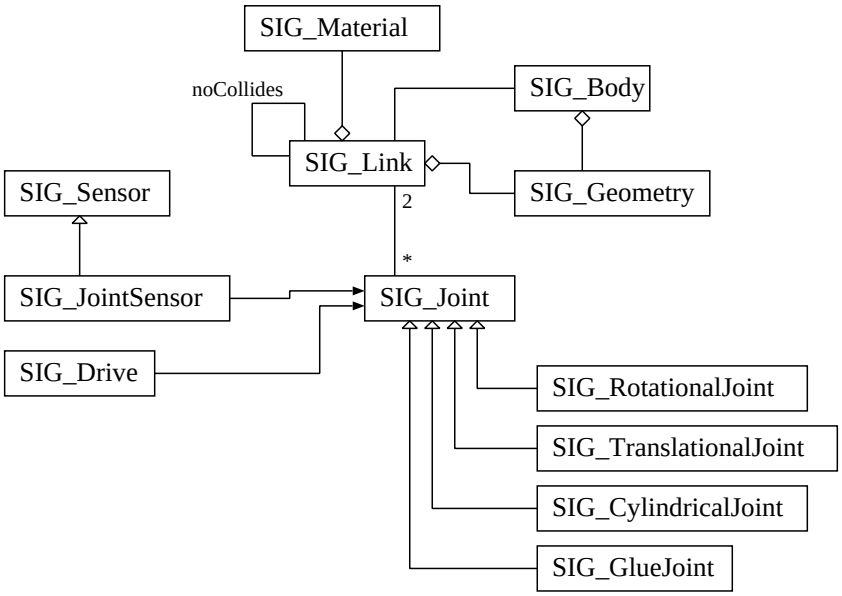


Abbildung 4.1: Vereinfachtes Strukturdiagramm des internen Robotermodells.

4.2.1.7 Das gesamte Dateiformat

Aus diesen Einzelteilen muss noch das komplette Dateiformat zusammengesetzt werden, wie im Folgenden geschehen:

robot = { material | link | joint | drive | sensor } { surface } .

4.2.2 Robotermodelle intern

In diesem Abschnitt geht es um die Speicherung des Robotermodells innerhalb von SIGEL. Abbildung 4.1 zeigt ein vereinfachtes Strukturdiagramm. Im Folgenden sind die einzelnen Klassen dieses Modells beschrieben. Neben der allgemeinen Modellierung haben die Klassen unterschiedliche Aufgaben in der Berechnung der für die Simulation notwendigen Daten.

SIG_Body Die Klasse *SIG_Body* ist die Repräsentation einer Datei mit Informationen zur Oberflächengeometrie. Bestandteil ist *SIG_Geometry*, die die Daten tatsächlich enthält. Hauptaufgaben von *SIG_Body* ist das Einlesen der Daten aus der VRML-Datei.

SIG_Geometry Diese Klasse dient der Speicherung der Oberflächeninformation in Form von Polygonen. Die Verwaltung erfolgt derart, dass ein Feld mit Punkten existiert und eines mit Polygon-Objekten. Die Eckpunkte eines Polygons werden als Indizes dargestellt, die auf die Elemente in einem Punkte-Feld zeigen. In Verbindung mit dieser Klasse stehen die Hilfsklassen *SIG_Polygon*

und *SIG_GeometryIterator*, die es erlauben, die Oberfläche für Grafik und Kollisionserkennung abzufragen.

SIG_Link Dies ist die Repräsentation eines Gliedes. Die Oberflächendaten sind anfangs über *SIG_Body* erreichbar und werden kopiert, wenn die physikalischen Attribute errechnet werden, weil die Daten eventuell für jeden Link, auch wenn er gleich aufgebaut ist, unterschiedlich transformiert werden müssen. Hauptaufgabe dieser Klasse ist die Berechnung der physikalischen Daten.

SIG_Material Hierin werden alle Daten gespeichert, die für die Spezifikation eines Materials wichtig sind.

SIG_Joint Diese Klasse repräsentiert ein Gelenk. Jedes Gelenk hat genau ein Glied auf jeder Seite. Die verschiedenen Gelenktypen sind Spezialisierungen dieser Klasse. Aufgabe von *SIG_Joint* und ihren Unterklassen ist die Berechnung der relativen initialen Position der Glieder zueinander.

SIG_Sensor Diese Klasse steht für einen allgemeinen Sensor. Ihre Unterklasse *SIG_JointSensor* repräsentiert einen Sensor, der die Gelenkstellung misst.

SIG_Drive Hierbei handelt es sich um die Repräsentation eines Motors.

Neben den oben aufgeführten Klassen gibt es noch weitere, von denen wahrscheinlich *SIG_Robot* die wichtigste ist. Sie ist der Container für alle bereits genannten Objekte und stellt das Interface für andere Teile der Applikation zur Verfügung. Dazu kommen noch die Klassen *SIG_LanguageParameters* und *SIG_CommandParameters*, die Informationen über die Steuerungssprache zum Interpreter und Simulator transportieren, aber für die Modellierung nicht wichtig sind.

4.2.3 Der Prozess des Compilierens

Das Compilieren ist der Prozess, mit dem aus einer Konfigurationsdatei ein Objektmodell aus den oben angegebenen Klassen konstruiert wird. Der dafür geschriebene Parser benötigt für diesen Prozess zwei Durchgänge. Im ersten Durchgang werden die Objekte des Modells erzeugt und in einem *SIG_Robot* gespeichert. Ferner werden Daten, die nicht von anderen Objekten abhängen, eingetragen (z.B. alle nummerischen Daten). Der zweite Durchgang dient der Vernetzung der erzeugten Objekte, z.B. werden die rechte und linke Seite der Gelenke mit den Gliedern verbunden.

4.2.3.1 Der Aufbau des Compilers

In Abbildung 4.2 ist die Struktur des Compilers für Robotermodelle zu sehen. Im folgenden werden die einzelnen Klassen erläutert.

Es ist zu beachten, dass von *allen* folgend aufgeführten Klassen Assoziationen zu *allen* Klassen des Modellierungsmoduls gezogen werden müssten, um ein

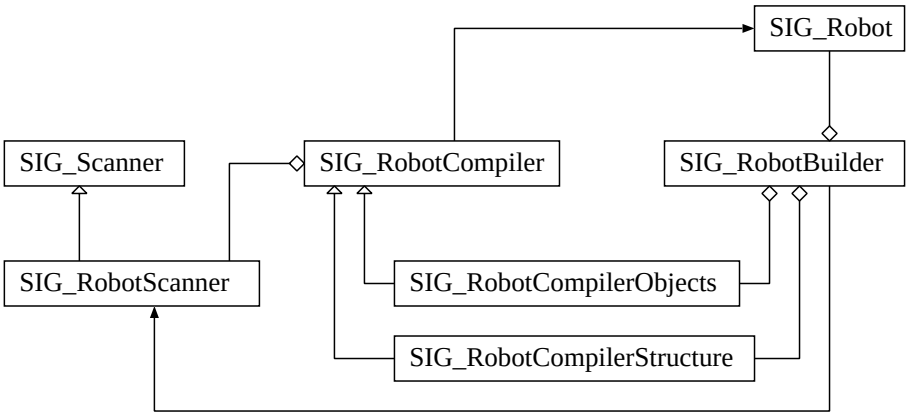


Abbildung 4.2: Die Klassenstruktur des Compilers für Robotermodelle.

korrektes Entwurfsdiagramm zu erhalten. Wir beschränken uns hier auf die wesentlichen Bestandteile.

SIG_Robot Diese Klasse ist der Container für das gesamte Robotermodell und steht hier stellvertretend für alle Klassen des Modells.

SIG_RobotBuilder Der RobotBuilder ist die „zentrale Schaltstelle“ für den Compiler. Diese Klasse steuert den groben Ablauf des Compiliervorgangs.

SIG_Scanner Die Klasse *SIG_Scanner* enthält einige grundlegende Methoden, um Texte lexikalisch zu analysieren.

SIG_RobotScanner Diese Klasse spezialisiert *SIG_Scanner* derart, dass ein Scanner speziell für Roboterbeschreibungen zur Verfügung steht.

SIG_RobotCompiler Diese Klasse enthält Analysemethoden für die syntaktische Struktur von Roboterbeschreibungen und leere virtuelle Methoden für jede Funktionalität.

SIG_RobotCompilerObjects Hier werden die virtuellen Methoden überschrieben, so dass sie die Funktionalität des ersten Compilerdurchgangs beinhalten. Die syntaktischen Analyse bleibt gleich.

SIG_RobotCompilerStructure Es werden hier die virtuellen Methoden von *SIG_RobotCompiler* überschrieben, so das sie die Funktionalität des zweiten Compilerdurchgangs beinhalten.

4.2.3.2 Der Ablauf des Compilierens

Der Benutzer, in diesem Fall handelt es sich dabei um das GUI-Modul, erzeugt ein Objekt der Klasse *SIG_RobotBuilder* und initialisiert es mit dem Namen der

Konfigurationsdatei. Anschließend ruft er *build* auf. Von nun an läuft der Prozess ohne weitere Eingriffe ab. *SIG_RobotBuilder* ruft nun die Methoden *firstPass* und *secondPass* von selbst auf. Zum Schluss wird beim fertig geladenen Roboter *loadGeometries* aufgerufen, damit die geometrischen Daten eingelesen werden.

Der erste und der zweite Durchgang sind vom Ablauf nahezu gleich. Zunächst wird die Konfigurationsdatei geladen und deren Inhalt einem frischen *SIG_RobotScanner*-Objekt zur lexikalischen Analyse übergeben. Dann wird ein Compilerobjekt, im ersten Durchgang von *SIG_RobotCompilerObjects*, im zweiten von *SIG_RobotCompilerStructure*, erzeugt und mit dem Scanner und dem Zielroboterobjekt initialisiert. Dann wird der Compiler angewiesen, die syntaktische Analyse durchzuführen. In deren Folge werden die unterschiedlich überschriebenen Methoden ausgeführt.

Der erste Durchgang Während des ersten Durchgangs werden alle notwendigen Objekte der Klassen *SIG_Body*, *SIG_Link*, *SIG_Joint* und dessen Spezialisierungen, *SIG_Material*, *SIG_Drive* und *SIG_Sensor* und dessen Spezialisierungen erzeugt und alle Daten, die nicht von anderen Objekten anhängen werden initialisiert. Ausnahme davon ist die Liste aller Glieder, die die gleiche Form haben. Diese Liste wird bereits jetzt aufgebaut.

Am Ende des ersten Durchgangs werden auch eventuell angegebene geometrische Daten aus der Konfigurationsdatei gelesen und in die entsprechenden *SIG_Body*-Objekte eingetragen.[1]

Der zweite Durchgang Im zweiten Durchgang werden Verbindungen zwischen den Objekten hergestellt. Dabei handelt es sich im einzelnen um

- die Liste der Gelenke, an denen ein Glied hängt,
- die Glieder, die an der rechten und linken Seite eines Gelenks hängen,
- die Liste der Materialien, für die eine Reibungskonstante zu einem bestimmten Material angegeben wurde,
- die Liste der Glieder, die mit einem bestimmten Glied nicht kollidieren,
- das Gelenk, das ein Sensor beobachtet und
- das Gelenk, das ein Motor bewegt.

Nachdem nach den beiden Durchgängen auch die geometrischen Daten eingelesen wurden, liegt das fertige Robotermodell in einer Rohfassung vor. Der Compiler wird nicht mehr benötigt. Bevor der Roboter aber einer Dynamiksimulation zugeführt werden kann, muß er einige Bedingungen erfüllen, die jetzt noch nicht sichergestellt sind.

[1] Das ist der Grund, warum sie sich immer am Ende der Datei befinden müssen. Sonst würde man Gefahr laufen, für eine noch nicht deklarierte Datei die Daten laden zu müssen.

4.2.4 Serialisierung und Deserialisierung von Robotermodellen

In diesem Abschnitt geht es darum, wie ein im Speicher aufgebautes Robotermodell in einen Text serialisiert, an einen anderen Ort gebracht und dort wieder in ein Modell deserialisiert werden kann. Es werden ein Serialisierungs- und ein Deserialisierungsverfahren beschrieben, die den Roboter zwar nicht absolut identisch, aber semantisch äquivalent reproduzieren. Dies ist wichtig, um ein Robotermodell via PVM (siehe auch Abschnitt 3.10) über das Netzwerk zu einem anderen Rechner übertragen. Der Algorithmus zum Einlesen ist einpässig. Für eine identische Reproduktion wäre ein zweipässiges Vorgehen von Nöten.

4.2.4.1 Was wird abgedeckt?

Von der Serialisierung sind alle Klassen des Pakets *SIGEL_Robot* betroffen, mit Ausnahme der Spezialisierungen von *SIG_Exception* und der Klasse *SIG_GeometryIterator*. Zusätzlich werden die Klassen *SIG_LanguageParameters* und *SIG_CommandParameters* übertragen, um z.B. die Befehlsdauer der einzelnen Befehle für den Interpreter bekannt zu machen.

4.2.4.2 Serialisierung

Zur Serialisierung muss man sich Gedanken darüber machen, wo die den Roboter beschreibende Datenstruktur Zyklen aufweist, und wie man diese Zyklen ohne wesentlichen Informationsverlust „flach“ bekommt.

Wo befinden sich zyklische Referenzen?

- Grundsätzlich ist es möglich, von *SIG_Robot* aus jedes andere Objekt zu erreichen. Die meisten Objekte haben aber auch Rückreferenzen auf *SIG_Robot*. Da es pro Modell nur ein *SIG_Robot*-Objekt gibt, wird dieses eingetragen.

- Zwischen *SIG_Geometry* und *SIG_Polygon* gibt es zyklische Bezüge. Diese existieren aber immer als Paar und jedes *SIG_Polygon* referenziert genau ein *SIG_Geometry*-Objekt.

- Ein *SIG_Link* hat einen Bezug auf ein *SIG_Body*-Objekt, dass seine geometrischen und physikalischen Daten enthält. Andererseits sind in *SIG_Body* sämtliche Links eingetragen, die dieselbe äußere Form haben. Auch hier gilt, dass sämtliche Verweise paarweise auftreten und jedes *SIG_Link*-Objekt genau ein *SIG_Body*-Objekt referenziert.

- Jedes *SIG_Joint*-Objekt referenziert genau zwei *SIG_Link*-Objekte, aber jedes *SIG_Link*-Objekt kann auf beliebig viele *SIG_Joint*-Objekte verweisen. Auch hier treten die Verweise grundsätzlich paarweise auf.

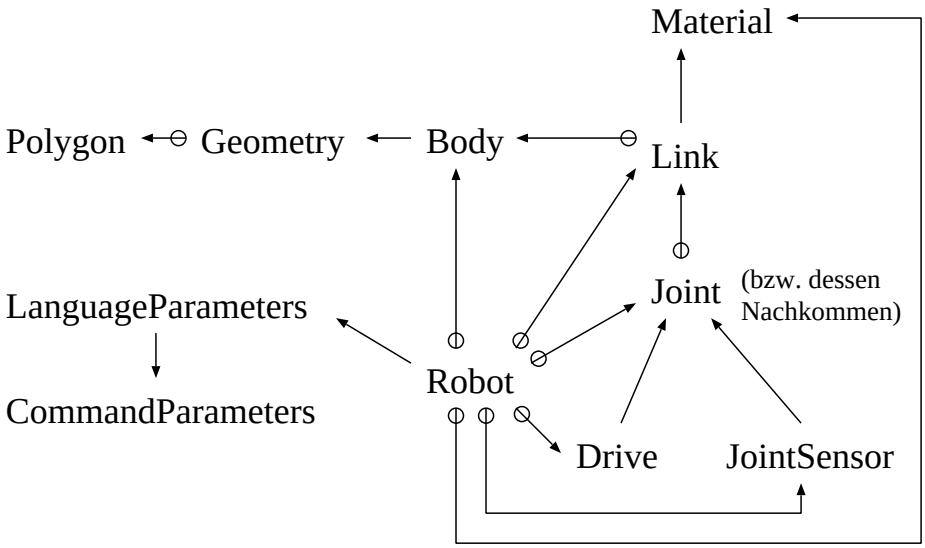


Abbildung 4.3: Abhängigkeiten im Robotermodell. Gestrichene Abhängigkeiten sind mit einem Kreis gekennzeichnet.

- Innerhalb von *SIG_Material* wird zur Festlegung von Reibungskoeffizienten auf andere *SIG_Material*-Objekte verwiesen. Auch diese Referenzen treten nur paarweise auf. Die annotierten Reibungskoeffizienten sind jeweils gleich.

- Innerhalb von *SIG_Link* werden Links verwaltet, die keine Kollision mit dem Glied verursachen können. Diese Eigenschaft ist paarweise vorhanden, demnach auch die Referenzen.

Sämtliche Mengen von Referenzen eines Objekttyps auf Objekte eines anderen Typs sind nicht geordnet.

Reihenfolge Da Referenzenmengen nicht geordnet sind, spielt die Reihenfolge der Eintragung in Listen, Dictionaries, etc. keine Rolle. Zyklische Referenzen, die paarweise auftreten, können vereinfacht werden, indem eine Referenz gestrichen wird. Wenn die übrig gebliebene Referenz eingetragen wird, wird die Gegenrichtung auch eingetragen.

In Abbildung 4.3 sind alle Abhängigkeiten des Robotermodells dargestellt. Die Klasse *SIG_Sensor* wurde weggelassen, da sie abstrakt ist und deshalb nicht auftritt. Gleiches trifft im Prinzip auf die Klasse *SIG_Joint* zu. Da alle Spezialisierungen von *SIG_Joint* jedoch die gleichen Abhängigkeiten haben, wurden sie wieder zu *SIG_Joint* zusammengefasst. Eventuelle Ableitungen von *SIG_Sensor* können andere Abhängigkeiten aufweisen.

Abhängigkeiten, die wegen der Singularität des *SIG_Robot*-Objekts oder durch paarweises Auftreten gestrichen wurden, sind mit einem Kreis anstelle eines Pfeils markiert. Es ist direkt einsichtig, daß der Graph keine gerichteten Zyklen mehr enthält.

Technik In der Klasse *SIG_Robot* gibt es sechs Dictionaries, deren Elemente serialisiert werden müssen. Zusätzlich kommt der Zweig mit *SIG_LanguageParameters* und *SIG_CommandParameters*. ("schreiben" heißt im Folgenden in den Serialisierungsstrom, die -datei schreiben).

1. Wir beginnen mit den *SIG_Body*-Objekten. Es wird für jedes Objekt folgendes durchgeführt:

   (a) Es werden die Attribute *geometryFile*, *centreOfGravity*, *majorInertiaAxes* und *inertiaTensor* geschrieben.

   (b) Es wird das in *geometry* eingetragene *SIG_Geometry*-Objekt geschrieben, indem zuerst reihenfolgeerhaltend alle Eckpunkte im Feld *vertices* geschrieben werden.

   (c) Es werden alle im Feld *polygons* gespeicherten *SIG_Polygon*-Objekte geschrieben, indem die darin im Feld *vertices* enthaltenen Indizes reihenfolgeerhaltend geschrieben werden.

2. An zweiter Stelle schreiben wir die *SIG_Material*-Objekte. Es werden die Attribute *name*, *elasticity*, *density*, *friction* und *colour* geschrieben. Im Falle *friction* wird das gegenüberliegende Material durch seinen Namen beschrieben.

3. Jetzt werden die *SIG_Link*-Objekte geschrieben. Dazu werden die Attribute *name, number, points, noCollide, initialPosition, initialOrientation, initiated* geschrieben. Im Falle *noCollide* wird das gegenüberliegende Glied durch seinen Namen beschrieben. Ferner werden die Referenzen *body* in Form des Namens der zugehörigen DXF-Datei und *material* in Form des Namens des Materials geschrieben.

4. Nach den Gliedern werden die Gelenke, d.h. alle *SIG_Joint*-Objekte oder deren abgeleitete Objekte geschrieben. Zunächst wird festgelegt, um welchen Gelenktypen es sich handelt. Geschrieben werden die Attribute *name, number, leftLink, rightLink*. Die Verweise auf die Links werden durch die Namen der Glieder beschrieben. Anschließend werden die Zahlenwerte aus der Spezialisierung des Gelenks (siehe Klassendiagramm) geschrieben.

5. Jetzt werden Objekte des Typs *SIG_Drive* geschrieben, indem die Attribute *name, number, minforce, maxforce* und *theJoint*, letzteres bezeichnet durch den Namen des Gelenks, geschrieben werden.

6. Zu guter Letzt werden die Sensoren gespeichert. Zur Zeit gibt es nur Gelenksensoren. Geschrieben werden die Attribute *name*, *number* und *theJoint*, letzteres bezeichnet durch den Namen des Gelenks.

7. Abschließend wird aus dem *SIG_Robot*-Objekt das Attribut *rootlink* geschrieben. Diese Referenz auf ein Glied wird durch den Namen des Gliedes beschrieben.

8. Zum Schluss werden die *SIG_LanguageParameters*- und *SIG_CommandParameters*-Objekte geschrieben. Die Vorgehensweise entspricht in etwa der von *SIG_Geometry* und *SIG_Polygon*.

Damit befinden sich, wie wir im nachfolgenden Abschnitt sehen werden, alle benötigten Roboterdaten im Stream.

4.2.4.3 Deserialisierung

Die Reihenfolge der Daten im Stream ist durch die oben beschriebene Serialisierung festgelegt. Zu zeigen ist jetzt, daß sich daraus ein semantisch gleichwertiges Robotermodell erzeugen lässt. Das Modell ist mit dem Ausgangsmodell in dem Sinne nicht identisch, dass die Reihenfolge, in der Referenzen auf andere Objekte in Listen und Feldern stehen, nicht mit der ursprünglichen Reihenfolge übereinstimmt. Für die Interpretation des Modells ist das jedoch belanglos. In der folgenden Beschreibung wird nur dargestellt, wie die Referenzen zwischen Objekten wiederhergestellt werden. Das Auslesen einfacher Attribute ergibt sich direkt aus der oben aufgeführten Serialisierung.

1. Als erstes wird ein *SIG_Robot*-Objekt erzeugt.

2. Die folgenden *SIG_Body*-Objekte werden in das *SIG_Robot*-Objekt unter ihrem Namen, der dem gelesenen Attribut *geometryFile* entspricht, eingetragen.

   Jedem *SIG_Body*-Objekt folgen direkt ein *SIG_Geometry*-Objekt und mehrere *SIG_Polygon*-Objekte. Das *SIG_Geometry*-Objekt ist im Attribut *geometry* einzutragen, die *SIG_Polygon*-Objekte mit dem *SIG_Geometry*-Objekt zu verlinken.

3. Es werden jetzt die *SIG_Material*-Objekte gelesen. Sie werden jeweils sofort mit ihrem Namen im *SIG_Robot*-Objekt eingetragen.

   Schwierigkeiten macht hier das Attribut *friction*. Nehmen wir an, bei Material *A* wird festgelegt, dass die Reibungskonstante zu Material *B* dem Wert *c* entspricht. Nun kann in *A* dieser Wert nicht eingetragen werden, weil das *SIG_Material*-Objekt für *B* noch nicht existiert. Die Konstante fällt also unter den Tisch. In *B* ist jedoch festgelegt, dass als Reibungskonstante zu *A* der Wert *c* zu benutzen ist. Dieser Wert kann eingetragen werden, da das

SIG_Material-Objekt von *A* bereits existiert. Jetzt wird *c* sowohl in *B* als Konstante zu *A*, als auch in *A* als Konstante zu *B* eingetragen. Zum Auffinden der Objekte werden die Einträge im *SIG_Robot*-Objekt benutzt.

4. Jetzt werden sämtliche *SIG_Link*-Objekte gelesen und unter ihrem Namen im *SIG_Robot*-Objekt eingetragen. Die Attribute *material* und *body* werden über die Einträge im *SIG_Robot*-Objekt initialisiert. Ferner wird das *SIG_Link*-Objekt im entsprechenden *SIG_Body*-Objekt in der Liste *usedBy-Links* eingetragen. *adjacentJoints* bleibt vorläufig leer, für *noCollide* wird das gleiche Verfahren wie für *Material::friction* angewandt, jedoch ohne eine Konstante.

5. Danach folgen alle *SIG_Joint*-Objekte, die unter ihrem Namen im *SIG_Robot*-Objekt eingetragen werden. Die Verweise *leftLink* und *rightLink* können sofort aufgelöst werden, weil alle *SIG_Link*-Objekte bereits erzeugt wurden. In diesen Gliedern wird das gelesene Gelenk im Feld *adjacentJoints* eingetragen.

6. Jetzt werden alle *SIG_Drive*-Objekte gelesen. Der Verweise auf das Gelenk kann aufgelöst werden, weil alle Gelenke bereits gelesen wurden.

7. Gleiches gilt für Gelenksensoren.

8. Zum Schluß werden die Objekte der Typen *SIG_LanguageParameters* und *SIG_CommandParameters* gelesen.

4.3 Aufbau der Robotersteuerungsprogramme

Im Rahmen unser Projektarbeit wurde eine eigene Sprache für die Robotersteuerungsprogramme entwickelt, welche im Folgenden näher erläutert wird. Die hier vorgestellte Sprache besteht lediglich aus sehr einfachen Maschinenbefehlen, was eine schnelle Implementation des Interpreters und auch eine schnelle Interpretation zulässt. Komplexes und in einem gewissen Sinne sinnvolles Verhalten wird nur durch die Kombination vieler Befehle möglich.

Das Gegenteil dazu wäre eine Architektur auf Basis komplizierter, auf die Anwendung zugeschnittener Befehle. Die Idee dahinter ist, den Suchraum zu verkleinern um schneller an Programme zu gelangen, die eine Bewegung des Roboters veranlassen. Beide Prinzipien haben ihre Vor- und Nachteile. Beim Vorschlag einfacher Befehle (i.F. Mikrobefehlssatz) ist der große Suchraum und damit die wahrscheinlich sehr lange Evolution zu bemängeln. Vorteil ist die mögliche Feinheit der Programme, was sehr schnelle und elegante Bewegungen zulässt.

Komplizierte Befehle (i.F. Makrobefehle) haben den Vorteile, bereits ein sinnvolles Verhalten zu implizieren. Evolution ist in diesem Fall nur die Suche nach sinnvollen Kombinationen, nicht aber die Konstruktion von effizienten Bewegungen. Es gibt zwei Möglichkeiten, diese Vorschläge zu integrieren.

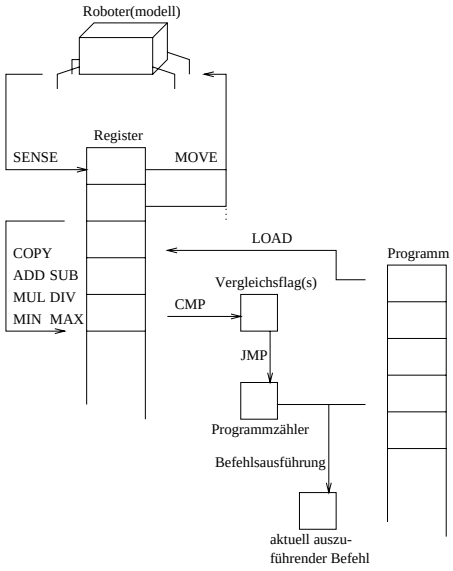


Abbildung 4.4: Der grundlegende Aufbau des Interpreters und die Wirkungsweise der Befehle.

1. Einem Mikrobefehlssatz können Makrobefehle hinzugefügt werden, in der Hoffnung die Produktion von Ausschusscode am Anfang zu vermindern.

2. Zunächst wird eine Evolution mit Makrobefehlen durchgeführt. Hat sich dabei eine einigermaßen flüssige Bewegung herauskristallisiert, können die Makrobefehle in Mikrobefehlssequenzen überführt werden, ähnlich wie eine Hochsprache in eine Assemblersprache übersetzt wird. Um die Bewegung zu vervollkommnen, wird die Evolution nun auf Mikrobefehlen fortgesetzt.

Variante 2 hätte dabei den Vorteil, dass man kein Interface zwischen Mikro- und Makrobefehlen konstruieren muss. Wegen diesen Gründen haben wir uns in der PG für Mikroanweisungen entschlossen.

4.3.1 Grundaufbau

Im Folgenden wird ein Mikrobefehlssatz beschrieben. Er basiert auf einer 2-Adress-Maschine. In Abbildung 4.4 ist der grobe Aufbau des Interpreters und seine Handhabung der Befehle wiedergegeben. Im weiteren wird versucht, das verwendete Registermaschinenmodell so detailliert aufzuschreiben, dass es ohne weiteres Zutun implementiert werden kann.

4.3.1.1 Definitionen

Im Folgenden sei also $R(X)$ der Inhalt des Registers Nummer X. Weiter sei $:=$ die Zuweisung, d.h. in $Y := X$ sind nach der Ausführung die Werte von X und Y gleich

dem Wert von X vor der Ausführung. Ferner sei $[X]$ die größte ganze Zahl, die kleiner als X ist, falls X positiv ist, und die kleinste ganze Zahl, die größer als X ist, falls X negativ ist. Die eckigen Klammern runden also immer in Richtung 0.

val ist die Operation, die ihr Argument in den gültigen Wertebereich der Register abbildet. Wir definieren diesen Wertebereich durch die Anzahl der Bits der Register. Werden n Bits verwendet, dann reicht der Wertebereich von -2^{n-1} bis $2^{n-1}-1$. Die Funktion sei folgendermaßen definiert:

$$\text{val } X = \begin{cases} \text{val}(X+2^n) & \text{falls } X < -2^{n-1} \\ X & \text{falls } -2^{n-1} \leq X \leq 2^{n-1}-1 \\ \text{val}(X-2^n) & \text{falls } 2^{n-1}-1 < X \end{cases} \tag{4.1}$$

Mit P wird das Programm bezeichnet, d.h. $P(X)$ ist der Befehl Nummer X des Programms. Das Vergleichsflag heißt VF und der Befehlszähler BZ. Zusätzlich sei AB der gerade auszuführende Befehl. Ist $L(P)$ die Anzahl der Befehle des Programms, so sind sie von 0 bis $L(P)-1$ durchnumeriert. Analoges gilt für die Register.

4.3.1.2 Ablauf

Zunächst werden Befehlszähler und Vergleichsflag gleich 0 gesetzt. Bei der Ausführung werden folgende Schritte durchlaufen:

1. Der Befehlszähler wird auf einen gültigen Wert gesetzt.

$$BZ := \begin{cases} BZ & \text{falls } BZ \geq 0 \wedge BZ < L(P) \\ 0 & \text{sonst} \end{cases}$$

2. Der aktuelle Befehl wird ausgelesen.

$$AB := P(BZ)$$

3. Der Befehlszähler wird hochgezählt.

$$BZ := BZ + 1$$

4. Der aktuelle Befehl AB wird ausgeführt. Siehe dazu die Auflistung des Befehlssatzes.

5. Falls die dem Programm zur Verfügung stehende Zeit noch nicht abgelaufen ist, mache weiter bei Punkt 1.

4.3.2 Befehlssatz

Im Befehlssatz der Steuerungseinheit befinden sich vier Klassen von Befehlen. Datentransportbefehle dienen dem Übertragen von Daten im Speicher und der Berechnung von Werten. Programmsteuerungsbefehle ermöglichen Verzweigungen und Verzögerungen im Programmfluss. Interaktionsbefehle ermöglichen Eingaben über Sensoren und Ausgaben über Motoren. Unter den sonstigen Befehlen befindet sich der Befehl zum Nichtstun.

4.3.2.1 Datentransportbefehle

Hier sind Befehle aufgelistet, die Daten aus Registern auslesen, sie manipulieren und Register beschreiben.

COPY X,Y

$$R(X) := R(Y)$$

LOAD X,Y

$$R(X) := Y$$

ADD X,Y

$$R(X) := \text{val}(R(X) + R(Y))$$

SUB X,Y

$$R(X) := \text{val}(R(X) - R(Y))$$

MUL X,Y

$$R(X) := \text{val}(R(X) \cdot R(Y))$$

DIV X,Y

$$R(X) := \begin{cases} \text{val}\left(\left[\frac{R(X)}{R(Y)}\right]\right) & \text{falls } R(Y) \neq 0 \\ 0 & \text{sonst} \end{cases}$$

MOD X,Y

$$R(X) := \begin{cases} \text{val}\,(R(X) \mod R(Y)) & \text{falls } R(Y) \neq 0 \\ 0 & \text{sonst} \end{cases}$$

MIN X,Y

$$R(X) := \begin{cases} R(X) & \text{falls } R(X) \leq R(Y) \\ R(Y) & \text{sonst} \end{cases}$$

MAX X,Y

$$R(X) := \begin{cases} R(Y) & \text{falls } R(X) \leq R(Y) \\ R(X) & \text{sonst} \end{cases}$$

4.3.2.2 Programmsteuerungsbefehle

DELAY X Der Programmablauf wird um $R(X)$ Millisekunden verzögert.

CMP X,Y

$$VF := \begin{cases} 1 & \text{falls } R(X) \leq R(Y) \\ 0 & \text{sonst} \end{cases}$$

JMP X

$$BZ := \begin{cases} BZ + X & \text{falls } VF = 1 \\ BZ & \text{sonst} \end{cases}$$

4.3.2.3 Interaktionsbefehle

Das Steuerprogramm kommuniziert mit dem Roboter und der Umwelt über die ersten Register des ihm zur Verfügung stehenden Speichers. Es sei hiermit festgelegt, dass der gesamte Zahlenbereich eines Registers für Messungen und Steuerungen ausgenutzt wird.

Im Falle von Motorsteuerungen ist dies besonders wichtig, weil das Programm dann keine Werte erzeugen kann, die den Antrieb über seine Extrembereiche hinaus bewegen würden.

SENSE X Dies ist die Eingabefunktion, mit der das Programm Informationen über die Außenwelt und den Zustand des Roboters einholen kann.

$$R(n) := \text{Sensor}_{R(X)^n}$$

Diese Definition bestimmt den Sensor indirekt über ein Register. Stattdessen könnte auch eine direkte Adressierung erfolgen. Der Parameter n iteriert über die Dimension des Sensors. So kann z.B. ein Sensor, der die Stellung eines Gelenks mit zwei Freiheitsgraden misst, zwei Werte gleichzeitig liefern.

MOVE X Dies ist die Ausgabefunktion, mit der das Programm auf die Gelenke des Roboters Einfluss nehmen kann.

$$\text{Gelenk}^n_{R(X)} := R(n)$$

Auch hier gibt es die Möglichkeit, direkt zu adressieren. Der Parameter n iteriert über die Freiheitsgrade, die das Gelenk bietet. Diese seien von 0 an durchgezählt.

Es gibt verschiedene Arten von Motoren und verschiedene Arten, einen Motor anzusteuern. In der Außenwelt tun wir gut daran, davon auszugehen, dass das Programm “weiß”, wie ein Motor anzusteuern ist und dass es dies mit dem MOVE-Befehl tut. Wir interpretieren diesen Wert gemäß unseren Wissens und unserer Vorstellungen.

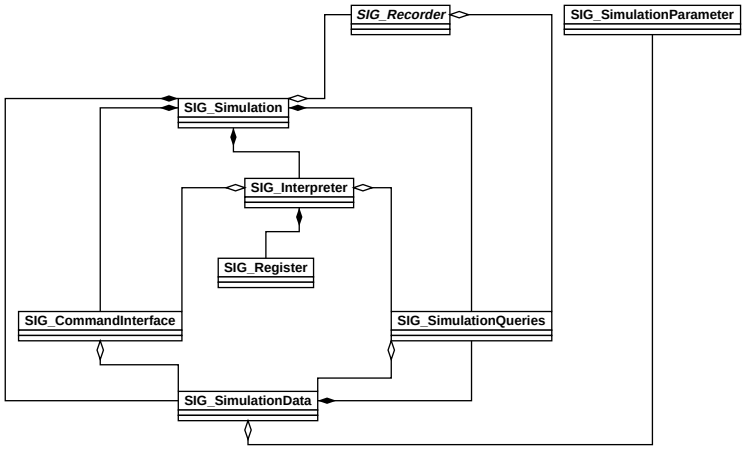


Abbildung 4.5: Grobes Klassendiagramm des Simulators

4.3.2.4 Sonstige Befehle

NOP Dieser Befehl macht gar nichts. Er belegt lediglich eine Programmstelle.

4.4 Der Dynamiksimulator

In diesem Kapitel werden nur die konkreten Systemteile der Dynamiksimulationsbibliothek *DynaMechs* betrachtet, die für das System *Sigel* von Relevanz sind. Dazu werden auch gewisse theoretische Aspekte hinsichtlich der vorkommenden Transformationen und Umrechnungen angegangen. Ferner wird ein kurzer Einblick in *DynaMechs*-spezifische mathematische Grundlagen gegeben.

Die Dynamiksimulation unterteilt (wie in Abbildung 4.4 zu sehen) sich in den eigentlichen Simulator, der mitlerweile mit zwei Dynamikbibliotheken implementiert worden ist und die Physik berechnet, den Interpreter, der für die Roboterprogramme zuständig ist und den Recorder, der ein Interface besitzt, um Daten aus der laufenden Simulation auszulesen.

4.4.1 Recorder

Der Recorder kann sowohl von der Fitnessfunktion als auch von der Visualisierung genutzt werden. Dafür ist es extrem wichtig, ein für beide nutzbares Interface zu erzeugen, mit dem zwar die Daten der laufenden Simulation jedes Frames auslesbar sind, aber die Simulation nicht beeinflusst werden kann. Also bietet sich Vererbung an. Beim *SIG_Recorder* werden zwar die virtuell zu erweiternden Funktionen aufgerufen, doch sie achten ausschließlich auf die korrekte Aufrufreihenfolge der Recorderfunktionen. Eine Erweiterung kann in jedem Frame in der *record* Metho-

de alle für sie wichtigen Daten auslesen, was zum Beispiel für die verschiedenen Fitnessfunktionen sehr wichtig ist.

4.4.2 Interpreter

Der Interpreter soll die von uns entworfene Roboterprogrammiersprache verstehen und in die Simulation einfließen lassen. Während für die meisten Kommandos Zugriffe auf die *SIG_Register*-Klasse reichen, muss für ein *SENSE*-Kommando ein Zugriff auf die (auch im Recorder genutzten) *SIG_SimulationQueries* erfolgen, für einen *MOVE*-Befehl muss sogar auf die Simulation Einfluss genommen werden, wozu die Klasse *SIG_CommandInterface* erschaffen wurde, um Klassen der Simulation vor unbefugten Zugriffen zu bewahren.

Ein typischer Ablauf sieht so aus, dass der aktuelle zu interpretierende Befehl geholt wird, dann geschaut wird, ob der Befehl erlaubt ist und wenn dies der Fall ist dieser ausgeführt wird. Anschließend wird die Zeit, die der Befehl benötigt, vom zur Verfügung stehenden Zeitkonto abgezogen und der Programmzähler um eins erhöht.

4.4.3 Simulation mit DynaMo und SOLID

Die beiden Bibliotheken funktionieren mit unserem Projekt nicht, diese Simulation ist also hauptsächlich aus historischen Gründen noch im Programm. Grundlegend benutzt die Simulation die Klasse *SIG_DynaSystem* und für die einzelne Teile des Roboters eine „dynamische“ Repräsentation wiederum als Klasse. Leider waren diese Bibliotheken eine Sackgasse, die sehr viel Zeit gekostet hat, näheres dazu auch in den Kapiteln 3.2 und 3.3.

4.4.4 Simulation mit DynaMechs

Als Ersatz für die oben genannten Bibliotheken hat sich die PG für die Bibliothek DynaMechs entschieden. Der Vorteil ist, dass die Bibliothek explizit für die Simulation von Robotern geschrieben wurde. Das Interface von DynaMechs ist zudem viel einfacher zu handhaben als das von DynaMo. Dafür mussten wir leider auf eine komplexe Kollisionserkennung verzichten.

Von der allgemeinen Basisklasse *dmSystem* wird die Klasse *dmArticulation* abgeleitet. Sie ist nun hauptverantwortlich für kinematische Bäume. *dmSystem* kann als Datencontainer für die gesamte Roboterrepräsentation gesehen werden, während *dmArticulation* nur die Roboter repräsentiert, die kinematische Bäume strukturell beinhalten.

Die Klasse *dmEnvironment* verfügt über ein statisches Klassenattribut, dass *dmArticulation* die Information gibt, um welches Environment es sich gerade handelt.

Von besonderer Wichtigkeit für uns sind die Klassen *dmLink* und die davon abgeleiteten Klassen *dmZScrewTxLink, dmRevoluteLink, dmMobileBaseLink* und

dmPrismaticLink. Darauf wird nun näher eingegangen.

4.4.5 Transformationen

Für jeden Roboter wird nun unmittelbar für die Simulierbarkeit und Darstellung eine Reihe von Transformationen gliedweise getätigt:

1. Koordinatensystemtransformation des Roboters aus dem zu *Sigel* konformen System, die der Modellierer (siehe Modellbeschreibung 6.3) festlegt, in die für DynaMechs spezifische Form. Dabei wird die Achse $\vec{Z}_i$ vom jeweiligen Glied i zur Schubachse erklärt. Der Vektor $\vec{X}_i$ wird dabei gemeinsame Normalenachse zwischen dem Gelenk i und i+1, siehe Bild 2.14 im Kapitel 2.2.

2. Nun werden im Rahmen einer Vorwärtsrechnung diese Achsen alle in ein (globales) Koordinatensystem umgerechnet. D.h., die lokalen Translations-/Rotations-Matrizen, die man stets benötigt um zwischen den lokalen Systemen hin-und her zurechnen. Oder anders gesagt, um z.B. von Gelenk i die Koordinaten bezüglich i-1 zu berechnen.

3. Der ganze Aufwand wurde getrieben, um nun die Denavit-Hartenberg Parameter (siehe Bild 2.14) zu bestimmen. Beispielsweise wird der Winkel α als Winkel zwischen den Achsen $\vec{X}_i$, $\vec{X}_{i+1}$ gemessen etc.

Anmerkungen: Zur schnellen Bestimmung der Richtung der Achse $\vec{X}_i$ im globalen Koordinatensystem, liest man einfach die erste Spalte der im Rahmen der Vorwärtskinematik bis zum Glied i aufmultiplizierten Transformationsmatrix. D.h. man erhält so schnell den *aktuellen* Basiseinheitsvektor $\vec{e}_x$. Mit Transformationsmatrix ist die Zusammenfassung aller Rotationen und Translationen (bis zum Glied i) in die Form homogener Koordinaten (siehe Kapitel 2.5), gemeint.

Softwaretechnisch wird in den drei vorher genannten Durchläufen jeweils *Pre-Order*-Baumdurchläufe durch den Robotergeometriegraphen vollzogen. Dies geschieht wohlgemerkt im *SIGEL-Master*, nur die transformierten Daten werden anschließend an die *SIGEL-Slaves* weiter gerreicht. Dies ist wesentlich effizienter, man spart die Wiederholung obiger Transformationen.

4.4.6 Die Behandlung der Links

Beim *dmZScrewTxLink*-Link (genauer: bei Instanzen dieser Klasse) handelt es sich um masselose Gelenke mit null Freiheitsgraden. Hier kann man nun eine konstante Winkelauslenkung und Verschiebung einstellen. Diese können nicht weiter verändert werden.

Die *dmRevoluteLink*-Links und *dmPrismaticLink*-Links haben jeweils einen Freiheitsgrad, und auch eine festgelegte Masse. Das *dmMobileBaseLink*-Link verfügt über sechs Freiheitsgrade: Drei rotatorische, und drei translatorische. Dieses

Gelenk dient nun als *Torso* (siehe 6.3), bzw. als "Rootlink". Von dort aus verzweigen die anderen Links. In *DynaMechs* werden Glieder und Gelenke durch, mit Ausnahme des *Torso*s, ein und das selbe Objekt modelliert.

dmZScrewTxLink hat nun eine Sonderrolle: Bei kinematischen Bäumen besteht nun das Problem, das ein Glied i gleich zwei Nachfolger i+1 und i'+1 haben kann. Wie führt man nun obige Transformationsschritte durch? Man fügt eine Instanz von *dmZScrewTxLink* als quasi-imaginäres Zwischenlink ein. Dabei ist die zugehörige konstante Winkelauslenkung gleich dem Winkel zwischen dem Glied i und i'+1. Die konstante Verschiebung ergibt sich aus dem Differenzvektor der Ursprünge von i und i'+1.

Rein geometrisch hat sich nichts verändert, nur kann man jetzt nach der Einfügung eindeutig von jedem Glied i den (eindeutigen) Nachfolger bestimmen. In welchem Nachfolgerzweig man dieses Extralink einfügt wird zufällig gewählt. Es wird hierbei statt der üblichen *Denavit-Hartenberg Parameter* eine leicht modifizierte Version benutzt, welche eine besonders effiziente Darstellung von Tensoren (siehe auch 3.4) erlaubt.

4.4.7 Wichtige Link-Methoden

Besonders erwähnenswert sind die Set-und Get-Methoden für die jeweiligen Linkklassen. Sie geben je einen Zustandsvektor zurück, der die Freiheitsgrade beinhaltet. Die Dimension dieses Vektors ist abhängig vom Link: So hat ein Link der Klasse *dmZScrewTxLink* einen nulldimensionalen Zustandsvektor, ist also komponentenlos, da null Freiheitsgrade vorhanden sind, während die Links von *dmRevoluteLink*, *dmPrismaticLinks* einen Zustandsvektor der Dimension 1 (entsprechend einem Freiheitsgrad) haben.

Wichtige Ausnahme ist hier die Klasse *dmMobileBaseLink*: Liefern doch Instanzen dieser Klasse einen siebendimensionalen Zustandsvektor. Davon sind die letzten drei Positionen für den globalen Positionsvektor in karthesischen Koordinaten gedacht. Die ersten vier dagegen beinhalten die Transfomationsmatrix in quaternionaler Schreibweise. Quaternione beschreiben Transformationsmatrizen (rotatorischer Prägung) besonders effizient und kurz.

Quaternione

Wie bereits aus den Grundlagenkapiteln bekannt ist, kann man eine allgemeine Transformation mathematisch als lineare Abbildung darstellen:

$$\vec{p}' = \underline{T} \cdot \vec{p}. \tag{4.2}$$

Dabei geht der Vektor (Punkt) $\vec{p}$ in den transformierten Vektor $\vec{p}'$ über. Diese Transformationsmatrix kann man nun als ein (aus der theoretischen Physik bekannten) Quaternion darstellen:

$$\underline{T} \cdot \vec{p} = q \cdot [0, \vec{p}] \cdot q* = \vec{p}', \tag{4.3}$$

mit dem Quaternion $q = a + b \cdot i + c\cot j + d \cdot k$ (i,j,k sind komplexe Zahlen). und $q* = a - b \cdot i - c\cot j - d \cdot k$, konjugiert komplexes Quaternion zu q. Es handelt sich bei Quaternionen um die mehrdimensionale Erweiterung von „herkömmlichen“ komplexen Zahlen $a + j \cdot b, j \in \mathbf{C}$. Dabei wurde $\vec{p}$ als allgemeiner Quaternion $[0, \vec{p}] = 0 + p_x \cdot i + p_y \cdot j + p_z \cdot k$ geschrieben, d.h. der reelle skalare Anteil ist Null.

Man stellt Drehungen (Rotationen) sehr bequem dar durch

$$q = [cos(\frac{1}{2}\phi), \vec{u} \cdot sin(\frac{1}{2} \cdot \phi)], \tag{4.4}$$

mit $\vec{u}$ als Drehachseneinheitsvektor (Länge eins).

In unserer Simulation wird nun ein Algorithmus angewendet um aus dem Quaternion effizient eine Rotationsmatrix zu berechnen. Es sei angemerkt, daß man via Quaternione keine Translationen durchführen kann. Es wird nun folgende Rechnung durchgeführt:

$$\vec{p}' = (sin^2\phi - cos^2\phi) \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + 2 \cdot cos^2\phi \cdot \begin{pmatrix} u_x^2 & u_x \cdot u_y & u_x \cdot u_z \\ u_x \cdot u_y & u_y^2 & u_y \cdot u_z \\ u_x \cdot u_z & u_y \cdot u_z & u_z^2 \end{pmatrix} - 2 \cdot sin\phi \cdot cos\phi \cdot \begin{pmatrix} 0 & u_z & -uy \\ -u_z & 0 & u_x \\ u_y & -u_x & 0 \end{pmatrix} \tag{4.5}$$

4.4.8 Kräfteapplizierung

Instanzen der Klasse *dmContactModell* erben von *dmForce* und assoziieren *dmLink.addForce()*, welche ein einziges Mal initial aufgerufen wird. Dadurch wird dem Link gesagt, welches Kontaktmodell von *dmContactModel* zum jeweiligen Link gehört.

Es sei nochmals daran erinnert, dass es in *DynaMechs* keine Kollisionen gibt, sondern (wie schon im Kapitel 3.4 angedeutet) Kontaktpunkte je Objekt für Pseudokollisionen benutzt werden, um mit Hilfe von maximalen-und minimalen Auslenkungen (Winkel und Verschiebung) möglichen Durchdringungen (und Berührungen) von Körpern, also Kollisionen, a priori zu unterbinden.

Dazu werden Gegenkräfte aufgewendet um die Körper im Sollauslenkungsbereich, bezüglich ihrer Freiheitsgrade Winkel und Verschiebung, zu halten. Diesen Sollbereich stellt man mit der Klassenmethode *setJointLimits* für das jeweilige Gelenk ein.

4.5 Das GP-System

Im Rahmen unseres Projektes ist ein GP System zum Einsatz gekommen. Es wurde beschlossen, das zu verwendende System eigens zu entwickeln und auf die Evolution von Robotersteuerungsprogrammen zuzuschneiden. Die Einbettung der Genetischen Programmierung in den Kontext der Evolutionären Algorithmen wurde

bereits im Grundlagenkapitel vorgenommen (s. 2.1.3). Dieser Abschnitt beschreibt die Realisierung eines GP Systems im Rahmen der Projektgruppe SIGEL.

4.5.1 Aufbau der GP evolvierten Programme

Der folgende Abschnitt gibt einen Überblick über die Struktur und den Aufbau der im Rahmen unseres GP-Systems evolvierten Programme.

4.5.1.1 Eigenschaften

Während der Evolution von Programmen innerhalb der Evolutionsschleife werden vollständige Programmstrukturen evolviert. Aus technischen Gründen weicht die interne Darstellung dieser Strukturen von der externen Darstellung (also das, was der Benutzer sieht) ab, da die interne Darstellung möglichst die folgenden Eigenschaften unterstützen soll:

- Optimale Speicherausnutzung
  Da unterschiedliche Roboterinstruktionen evolviert werden, ist der Speicherbedarf der einzelnen Instruktionstypen durchaus unterschiedlich. Da eine Population mit vielen Programmen sehr groß werden kann, sollte die Speicherung der Programmzeilen nicht mehr Speicher benötigen als wirklich notwendig.
- Schneller (möglichst indizierter) Zugriff auf Daten
  Um die Handhabung der Programme zu vereinfachen, soll ein indizierter Zugriff auf die Programmzeilen möglich sein.

4.5.1.2 Funktionalität eines Programms

Ein Programm stellt alle notwendigen Funktionen zur Verfügung, welche zur Handhabung der internen Datenstruktur (s. 4.5.1.3) notwendig sind. Dazu gehören Funktionen für den einfachen Zugriff auf die interne Datenstruktur und ein zufälliges Erstellen eines Programms auf Basis der GP Parameter (s. 4.5.5). Programme können auch auf mehrfache Weise im- oder exportiert werden:

- Import/Export eines Programms auf Basis einer ASCII-Textdatei
  Das Erstellen von Programmdateien, welche in eine Population importiert werden können (oder umgekehrt) ist unter 4.5.8 ausführlich erläutert.
- Streambasierter Populationsinterner Programmaustausch
  Innerhalb der Population können Programme ohne größeren Aufwand ausgetauscht werden.
- Laden/Speichern von Programmen
  Programme können gespeichert werden, um später wieder geladen werden zu können.

Die Datenstruktur und der Zugriff auf diese seien im Folgenden näher erläutert.

4.5.1.3 Interne Darstellung eines Programms

Ein Programm ist eine Liste von Programmzeilen. Die Länge von Pogrammen ist (theoretisch) unbegrenzt, wird aber in der GP-Praxis beschränkt. Ein Programm ist ein Vektor, dessen Dimension sich während der Evolution verändern kann:

$$PROGRAMM = (INSTRUKTION_1, INSTRUKTION_2, \cdots, INSTRUKTION_n)$$

Das Programm *PROGRAMM* ist ein n-dimensionaler Vektor bestehend aus den Programmzeilen $INSTRUKTION_1, \cdots, INSTRUKTION_n$, auf welche mit einem Index i mit $0 < i \leq n$ zugegriffen werden kann. Die Zeile $INSTRUKTION_i$ beinhaltet schließlich die Informationen über die Roboterinstruktion i. Jedes GP evolvierte Programm besitzt eine Methode, mit welcher direkt auf die Programmzeile eines Programms zugegriffen werden kann.

4.5.2 Aufbau einer Programmzeile

Die Programmzeilen beinhalten mit den Robotinstruktionen die eigentlich Funktionalität der Programme. Es existieren mehrere Arten von Instruktionen, welche vom GP System unterstützt werden. Je nach Art der Instruktion müssen unterschiedliche Informationen über die Programmzeile gespeichert werden. Im Allgemeinen hat jede Programmzeile den folgenden formalen Aufbau:

$$INSTRUKTION_i = (TYPE, OPERAND_1, OPERAND_2, ..., OPERAND_m)$$

Bei jeder Programmzeile handelt es sich um einen $m+1$ - dimensionalen Vektor, der die Angabe des Programmzeilentyps ($TYPE$) und die Operanden der jeweiligen Instruktionen enthält. Für jeden Operanden j gilt $OPERAND_j \in [-32000, +32000]$. Die Dimension m dieses Vektors ist nicht für jede Programmzeile gleich, sondern ist abhängig vom Instruktionstyp $TYPE$. Die Semantik der Operanden ist mit dem Roboterinstruktionstyp automatisch festgelegt (s. auch 4.3.2). Für den Typ einer Instruktion t gelten alle weiter unten genannten Operationen.

Der Roboter kann eine bestimmte Anzahl von Registern, jedoch mindestens zwei besitzen, weil die Adressierung aller Befehle indirekt ist. Die Operanden werden per *modulo* auf die Anzahl der Register umgerechnet.

Die Breite der Register wird in Bit angegeben, d.h. ein Register der Breite n kann Werte von -2^{n-1} bis $2^{n-1}-1$ beinhalten. Für alle Befehle, die mit der Simulation interagieren, also SENSE und MOVE, wird der Registerbereich auf den entsprechenden Kraft- bzw. Auslenkungsbereich abgebildet (vgl. auch Abschnitt 4.3.2).

4.5.2.1 Instruktionstyp

Jede Programmzeile ist einem bestimmten Typ zugeordnet. Die Zuordnung eines Programmzeilentyps hat einen direkten Einfluss auf die Interpretation der

Programmzeile. Im Rahmen des SIGEL GP Systems werden $m \in \{0,1,2\}$ und $TYPE \in \{$COPY, LOAD, ADD, SUB, MUL, DIV, MOD, MIN, MAX, CMP, JMP, SENSE, MOVE, DELAY, NOP$\}$ unterstützt. Abschnitt 4.3.2 gibt einen genaueren Überblick über die Syntax und Semantik aller Befehle. Eine Erweiterung des Befehlssatzes ist ebenfalls möglich.

4.5.2.2 Zugriff auf Programmzeilen

Im Laufe der GP Versionen wurde der Zugriff auf die Programmzeilen eines GP evolvierten Programms mehrfach vereinfacht. Soll auf eine Programmzeile j eines Programmes i zugegriffen werden, so kann mit wenigen Befehlen auf die Informationen einer Instruktion zugegriffen werden. Ebenfalls können sehr einfach mit wenigen Befehlen Informationen einer GP evolvierten Programmzeile verändert werden, wie dies z.B. im Rahmen der genetischen Operatoren geschieht. Im Wesentlichen kann der Zugriff auf einzelne Programmzeilen auf drei Grundfunktionen beschränkt werden:

- Auslesen des Instruktionstyps
  Vor dem Zufgriff auf die Operanden einer Instruktion sollte der Typ der aktuellen Instruktion erfragt werden. Dies ist jedoch nicht zwingend erforderlich, da auch ohne diese Information gearbeitet werden kann. Das Ergebnis dieser Funktion liefert jedoch Informationen über die Semantik der aktuellen Instruktion (s. 4.5.2.1).

- Auslesen der Operatoren
  Es wird das Auslesen des n-ten Operanden unterstützt. Über die Semantik des n-ten Operanden gibt der Instruktionstyp Aufschluss.

- Setzen von Informationen
  Das Setzen von Informationen ist sehr einfach und ist bereits nur mit einem Funktionsaufruf möglich, indem dieser Funktion bereits die notwendigen Informationen über den Instruktionsstyp und ggf. den ersten und zweiten Operanden. übergeben werden. Die einzelnen Informationen können jedoch auch einzeln gesetzt werden.

4.5.2.3 Funktionalität einer Programmzeile

Eine Programmzeile stellt vorwiegend die folgenden Funktionen zur Verfügung. Es existiert eine Zugriffsfunktionen auf die Elemente einer Roboterfunktion, außerdem wird das automatische Erstellen einer solchen unter Beachtung der GP Parameter (s. 4.5.5) unterstützt. Ebenfalls in der Programmzeile enthalten sind die grundlegenden Ausgabe- und Streamingfunktionen.

4.5.3 Aufbau der Individuen

Die Individuen beinhalten die GP Programme und viele weitere Informationen, welche während der Evolution gesammelt worden sind und welche für die weitere Evolution wichtig sind (*EVOLUTIONSDATEN*):

$$INDIVIDUUM = (EVOLUTIONSDATEN, PROGRAMM)$$

Jedes Individuum führt z.B. ebenfalls eine History mit sich, welche Daten über die Entwicklung des Individuums enthält. Darunter vor allem das Datum und die Zeit der Erstellung und das Datum und die Zeit von Manipulationen. Bei Veränderung durch Mutation wird der Mutationspunkt und der Elter, beim Crossover beide Kreuzungspunkte und die Eltern angegeben.

4.5.3.1 Funktionalität des Individuums

Das Individuum stellt mehrere Funktionen zur Verfügung, so kann z.B. mit Zugriffsfunktionen auf das Individuum und das enthaltene Programm Einfluss genommen werden und dieses im Sinne der der GP Parameter (s. 4.5.5) zufällig erstellt werden. Zusätzlich können Individuen auf mehrfache Weise im- oder exportiert werden, analog zu Abschnitt 4.5.1.2.

4.5.4 Aufbau der Population

Die Population besteht aus einer Menge von k Individuen. Die hauptsächliche Aufgabe der Population P besteht in der Verwaltung dieser Individuenmenge:

$$P = (DATA, INDIVIDUUM_1, INDIVIDUUM_2, \cdots, INDIVIDUUM_k)$$

Ebenfalls enthält die Population einige Daten (*DATA*), welche zur Evolution benötigt werden. Alle notwendigen Funktionen, welche für die Verwaltung einer Individuenmenge im Rahmen von Evolutionen notwendig sind, werden durch die Population zur Verfügung gestellt. Dabei kann auf jedes Individuum zugegriffen werden und es existieren Statistikfunktionen. Es ist möglich, die gesamte Population (z.B. auf die Festplatte) zu speichern, um diese zu einem anderem Zeitpunkt wieder zu laden. Dadurch sind GP Versuche möglich, welche beliebig unterbrochen und wieder fortgesetzt werden können. Da es sich um ein textbasiertes Format handelt, kann eine gesamte Population inklusive aller Individuen und Programme mit einem Texteditor manuell erstellt oder nachbearbeitet werden.

4.5.5 GP Parameter

Die GP Parameter legen fest, welche Bedingungen für die Individuen (inklusive ihrer Programme) während der gesamten Evolution zu erfüllen sind. Weiterhin legen diese Parameter evolutionäre Parameter und Kriterien fest, die einen direkten

Einfluss auf den Ablauf der Evolution selbst haben. Damit sämtliche Einstellungen nicht stets neu eingegeben werden müssen, besteht die Möglichkeit, alle GP Parameter zu speichern und wieder zu laden. Die möglichen Einstellungen und ihre Auswirkungen seien im Folgenden nun näher beschrieben. Eine detaillierte Beschreibung der grafischen Benutzerschnittstelle erfolgt in Kapitel 6.5.

4.5.5.1 Grundlegende Einstellungen

Mit diesen Einstellungen werden die grundlegenden GP Einstellungen festgelegt. Darunter die folgenden Einstellungen für

- Random Seed
  Da GP Läufe stets reproduzierbar sein sollen, kann an dieser Stelle der Seed angegeben werden, mit welchem der GP-Lauf durchgeführt werden soll.

- Minimale und maximale Programmlänge
  Alle Funktionen des GP Systems, welche einen Einfluss auf die Länge von Programmen ausüben, wie z.B. die genetischen Operatoren, dürfen keine Programme erzeugen, welche länger als die Maximallänge oder kürzer als die Minimallänge sind. Da durch die Möglichkeit des Programmimports oder durch das zwischenzeitliche Änderung der GP Parameters die Länge eines Programms nicht mehr den Anforderungen dieser Einstellungen entsprechen kann, wird die Länge durch die genetischen Operatoren (s. auch 4.5.7) stets angepasst.

- Maximales Alter der Individuen
  Überschreiten Individuen ein bestimmtes Alter werden sie aus der aktuellen Population entfernt.

- Wahrscheinlichkeiten für die genetischen Operatoren
  Diese Einstellungen definieren die Wahrscheinlichkeit für die Anwendung der einzelnen genetischen Operatoren Mutation, Rekombination und Reproduktion.

  Je höher der eingestellte Wert w_{op} mit $0 \leq w_{op} \leq 100$ desto höher die relative Wahrscheinlichkeit. Die Operatorwahrscheinlichkeit wird relativ zu den anderen Operatorwahrscheinlichkeiten berechnet und müssen in der Summe stets 100 ergeben.

- Name der zu verwendenden Fitnessfunktion
  Das GP System unterstützt die Möglichkeit, zwischen mehreren Fitnessfunktionen zu wählen, ohne eine Neukompilierung des Systems notwendig zu machen. Durch die Eingabe des Namens dieser Funktion wird die entsprechende Fitnessfunktion festgelegt.

4.5.5.2 Wahrscheinlichkeiten von Instruktionen

Die Häufigkeit einer Instruktion kann ebenfalls festgelegt werden. Die Einstellung wird im Rahmen der Erstellung neuer Programmzeilen sowie bei den genetischen Operationen berücksichtigt. Jeder Instruktion i kann eine relative Wahrscheinlichkeit w_i mit $0 \leq w_i \leq 1000$ zugeordnet werden. Auch hier gilt, dass die Wahrscheinlichkeit w_i relativ zu allen anderen Wahrscheinlichkeiten berechnet wird. Ist eine Instruktion deaktiviert, so hat diese Einstellung kein Gewicht und wird als $w_i = 0$ angenommen. Die implementierte Auswahl von Instruktionen unter Berücksichtigung ihrer Wahrscheinlichkeiten entspricht im Wesentlichen einem *Roulette-Wheel* Algorithmus.

4.5.5.3 Technische Einstellungen

Um einen Überblick über die Entwicklung einer Population zu erhalten, ist es möglich, alle aus der Population nicht mehr berücksichtigten Individuen auf der Festplatte zu speichern (*Graveyard*). Ebenfalls besteht die Möglichkeit in regelmäßigen Abständen die gesamte Population als ein *Image* abzuspeichern. Die Frequenz der Images kann frei gewählt werden.

4.5.5.4 PVM Einstellungen

Wie bereits in den vorangegangen Kapiteln erwähnt, unterstützt das GP System die parallele Evaluierung von Individuen. Mit den PVM Einstellungen können alle Einstellungen bezüglich PVM vorgenommen werden, ohne die PVM Konfiguration manuell vornehmen zu müssen. Die PVM Parameter werden im Rahmen der GP Parameter verwaltet, zu den Einstellungen jedoch mehr im entsprechenden PVM Kapitel (s. 6.5).

4.5.5.5 Kontrolle der Evolution

Es bestehen mehrere Möglichkeiten das Abbruchkriterium für einen GP Lauf zu definieren: Abbruch nur durch den Benutzer, durch Angabe einer Laufzeit oder Generationenanzahl, oder aber Abbruch sowohl durch Laufzeit als auch Benutzer. Die zu den jeweiligen Einstellungen gehörenden Parameter werden ebenfalls erfasst.

4.5.6 Evolutionsschleife

Es wird der Algorithmus vorgestellt, der einen Individuenpool von einer Generation in die nächste überführt und somit ein wesentliches Modul in jedem GP System darstellt. Der Algorithmus ist besonders für den Fall geeignet, dass die Berechnung der Fitnessfunktion wesentlich mehr Zeit in Anspruch nimmt als die Anwendung der genetischen Operatoren selbst. Er ermöglicht deswegen eine parallele Berechnung der Fitnesswerte für verschiedene Individuen.

4.5.6.1 Was passiert bei der Evolution?

Ein Generationsübergang der Evolutionsschleife besteht aus einer Folge von Operationen auf dem Individuenpool. An jeder Operation ist eine bestimme Anzahl von Individuen beteiligt. Der Algorithmus ist hier allgemein formuliert. Momentan sind aber folgende implementiert:

Reproduktionsturniere Zwei Individuen treten gegeneinander an. Das „bessere" Individuum überschreibt den Verlierer mit einer Kopie von sich.

Mutationsturniere Zwei Individuen treten gegeneinander an. Das „bessere" Individuum überschreibt den Verlierer mit einer Kopie von sich, auf den ein Mutationsoperator angewendet wurde.

Crossoverturniere Hier sind vier Individuen beteiligt. Es werden zwei disjunkte Paare gewählt. Die zwei Individuen jeweils eines Paares treten gegeneinander an (Vergleich der Fitnesswerte). Abhängig vom Ergebnis werden zwei Individuen (die Gewinner) miteinander verschmolzen und ersetzen die beiden Verlierer.

Einige Besonderheiten unseres GP-Systems sind in diesem Zusammenhang wichtig. Der Individuenpool ist eine geordnete Menge, seine Größe ändert sich während der Evolution nicht und bei den oben genannten Turnieren ändern sich nur die Positionen des Pools, an denen sich beteiligten Individuen befinden bzw. befanden. Wird also z.B. ein Turnier zwischen den Individuen I und J an den Pool-Positionen p_I und p_J ausgetragen, darf anschließend nur eine dieser beiden Positionen mit einem (replizierten oder mutierten) Individuum überschrieben werden.

4.5.6.2 Abhängigkeiten zwischen Turnieren

Sinnvollerweise wird zur Durchführung einer Operation der aktuelle Fitnesswert aller beteiligten Individuen benötigt. Durch Mutation oder Crossover entstandene Individuen besitzen zunächst keinen solchen Wert.

Die Berechnung dieses Fitnesswertes wird in unserem Fall durch Simulation eines Roboters ermittelt und benötigt somit wesentlich mehr Zeit als die GP-Operationen an sich. Diese Simulationen können aber für mehrere Individuen parallel (z.B. auf verschiedenen Computern) geschehen, da sie voneinander unabhängig sind.

Zwischen Elementen der Operationenfolge, die letztlich den Generationsübergang darstellt, bestehen jedoch sehr wohl Abhängigkeiten. Nimmt das Individuum I an Poolposition p_I an der Operation an Position x der Operationenfolge (z.B. ein Reproduktionsturnier) und der Operation an Position y (z.B. ein Crossoverturnier) teil (und $x < y$), so darf erst nach Beendigung von x und einer evtl. Aktualisierung des Fitnesswertes von I mit y begonnen werden. Es besteht also eine partielle Ordnung $\prec$ auf der Menge der Operationen:

$$Op_x \prec Op_y :\Leftrightarrow (\exists I : I \text{ Teilnehmer von } Op_x \text{ und } Op_y \wedge x < y)$$

Dazu sei $<$ der transitive Abschluss von $\prec$. Bei der Bearbeitung von Operationen muss $<$ respektiert werden.

4.5.6.3 Ablauf der Evolution

Der Algorithmus ist im Prinzip eine topologische Sortierung der Turniere, die die $<$-Ordnung respektiert. In einem Preprocessing-Schritt wird $<$ sowie die initiale Menge der minimalen Elemente der Ordnung berechnet. Sie enthält mindestens die erste Operation der Folge, die auf jeden Fall unabhängig von anderen Schritten ausgeführt werden kann. Der Übergang von einer Generation zur nächsten besteht informell aus folgenden Schritten:

1. Für alle Individuen des Pools wird, falls noch nicht geschehen, der aktuelle Fitnesswert berechnet.

2. Es wird ein zufälliger Operationsplan aufgestellt (Folge von verschiedenen Turnieren).

3. Die eigentliche Schleife.
   Beendete Turniere werden aus der durch $<$ partiell geordneten Menge entfernt, wodurch neue minimale Elemente entstehen. Wichtig: “Beendet” bedeutet hier nicht nur, dass Fitnesswerte verglichen, Gewinner kopiert und genetische Operatoren angewendet wurden, sondern dass außerdem die Fitnesswerte der so neu entstandenen Individuen berechnet wurden. Erst dann gilt das Turnier als abgeschlossen.

   Beim herkömmlichen Algorithmus zur topologischen Sortierung einer partiell geordneten Menge wird die aktuelle Teilmenge minimaler Elemente in beliebiger Reihenfolge ausgegeben. In unserer Evolutionsschleife hingegen wird diese Teilmenge permanent durchlaufen und für jedes Turnier einzeln geprüft, ob diese durch die Fertigstellung von Fitnesswertberechnungen als abgeschlossen betrachtet werden können.

Entscheidend ist, dass bei Beginn einer Operation die Fitnesswerte der beteiligten Individuen bereits aktuell sind. Dies wird anfangs durch Schritt 1 garantiert. Neu entstandene Individuen erhalten sofort ihren Fitnesswert, bevor weitere Operationen, an denen sie teilnehmen, „freigeschaltet“ werden.

4.5.6.4 Technischer Ablauf der parallelisierten Fitnesswertberechnung

Zu Beginn der Evolution wird eine Kopie des Roboters bezüglich der zu verwendenden Simulationsbibliothek transformiert, d.h. das vom Benutzer eingegebene Modell wird in die entsprechende dynamische Darstellung umgerechnet. Zusammen mit den Simulationsparametern, dem zu bewertenden Robotersteuerungsprogramm, der Simulationsumgebung und dem Namen der zu verwendenden Fitnessfunktion bildet er (gekapselt in der Klasse *SIG_GPPVMData*) die Daten, die an

die einzelnen Slave-Prozesse geschickt werden, um dort den Fitnesswert berechnen zu lassen.

Die Klasse *SIG_FitnessTrainer* verwaltet die Verteilung von Fitnesswertberechnungen auf die einzelnen Rechner der virtuellen Maschine. An ihn kann die Anfrage gestellt werden, ein Individuum bewerten zu lassen. Der Fitnesstrainer übernimmt diese Aufgabe und gibt an den Anfragesteller eine Identifikationsnummer zurück, unabhängig davon, ob momentan eine Fitnesswertberechnung gestartet werden kann oder nicht. In regelmässigen Abständen versucht er, die Aufträge, für die noch kein Slave-Prozess erzeugt werden konnte, im Nachhinein zu bearbeiten.

Der Fitnesstrainer verteilt die Prozesse gleichmäßig auf die aktivierten Rechner der virtuellen Maschine und beachtet dabei die (in der graphischen Oberfläche einzustellende) Maximalanzahl von Slave-Prozessen, die auf einem Rechner gleichzeitig existieren dürfen.

4.5.7 Genetische Operatoren

Wie in 2.1.3 beschrieben, sind die genetischen Operatoren für die Variation der genetischen Informationen verantwortlich. Die Häufigkeit ihres Einsatzes wird über die GP Parameter (s. 4.5.5) bestimmt. Die Verfahren der implementierten Varianten seien nun im Folgenden näher erläutert.

4.5.7.1 Reproduktion

Die Reproduktion ist die einfachste Operation, welche ein Individuum nur kopiert. Da es durch den Import von Programmen und Individuen zu Inkonsistenzen bei der Programmlänge kommen kann, wird während der Reproduktion eines Individuums die definierte Mindest- und Maximallänge (s. 4.5.5.1) fortwährend geprüft. Im Fall der Unterschreitung der Minimallänge wird das Programm mit NOP Instruktionen (s. 4.3.2) bis zur Minimallänge aufgefüllt. Im Fall der Überschreitung der Maximallänge wird das Programm auf die maximale Länge gekürzt.

4.5.7.2 Mutation

Bei der Mutation als Innovationsoperator werden mehrere Mutationsvarianten unterstützt, wobei das Mutationsverfahren nur das Programm beeinflusst, nicht jedoch weitere Daten des Individuums, wie z.B. die History. Bei allen Varianten werden stets die Einstellungen der GP Parameter (s. 4.5.5) beachtet. Die folgenden Mutationsvarianten sind implementiert und sind gleichwahrscheinlich:

- Mutation einer kompletten Programmzeile
  Bei dieser Variante wird eine zufällig gewählte Programmzeile durch eine neue zufällige Programmzeile ersetzt.

- Mutation eines Programmzeilenelements
  Bei dieser Variante wird nur ein Element durch ein neues Element ersetzt, und zwar entweder der Instruktionstyp oder ggf. der 1. oder 2. Operand. Dabei ist natürlich zu prüfen (und ggf. zu korrigieren) , ob durch den Tausch des Instruktionstyps nicht irgendwelche Inkonsistenzen entstehen. Wird beispielsweise ein Instruktionstyp mit einem Operanden durch einen Instruktionstyp ersetzt, welcher jedoch zwei Operanden benötigt, so muss gleichzeitig ein zweiter Operand neu erzeugt werden, um keine ungültigen Programmzeilen zu erzeugen.

- Löschen von Programmzeilen
  Bei dieser Variante wird eine ganze (durch Zufall bestimmte) Programmzeile gelöscht, soweit die Minimallänge nicht unterschritten wird. Droht die Unterschreitung der Minimallänge, wird diese Variante nicht durchgeführt, und die folgende Variante angewendet.

- Hinzufügen von Programmzeilen
  Bei dieser Variante wird eine zufällig erzeugte Programmzeile an eine zufällig bestimmte Stelle des aktuellen Programms eingefügt. Droht die Überschreitung der zulässigen Maximallänge, wird diese Variante nicht durchgeführt, und es wird die vorherige Variante angewendet. Es erfolgt jedoch kein Tausch der Varianten, wenn die Maximal- und Minimallänge gleich sind.

Auch bei der Mutation wird stets die Länge des aktuellen Programms geprüft. Sollte ein Individuum vorliegen, welches die Kriterien der minimalen oder maximalen Länge nicht erfüllt, so wird wie bei der Reproduktion verfahren.

4.5.7.3 Rekombination

Bei der Rekombination werden elterliche Informationen gemischt und auf die Nachkommen übertragen. Nach der Wahl von zwei zufälligen Kreuzungspunkten x_1 (Kreuzungspunkt im Programm des ersten Elters) und x_2 (Kreuzungspunkt im Programm des zweiten Elters) werden die Informationen wie folgt gemischt:

Sei $P_1 = \{I1_0, I1_1, \cdots, I1_n\}$ das Programm des ersten Individuums, wobei $I1_0$ bis $I1_n$ die n Programmzeilen von P_1 sind. Für das zweite Programm gelte entsprechendes. indem $P_2 = \{I2_0, I2_1, \cdots, I2_m\}$ das Programm des ersten Individuums und $I2_0$ bis $I2_m$ die m Programmzeilen von P_2 sind. Für x_1 gelte $x_1 \in \{0, 1, \cdots, n\}$, während für $x_2 \in \{0, 1, \cdots, m\}$ gelte. Dann realisieren die folgenden Operationen die Reproduktion:

$$P_{neu1} = \{I1_0, I1_1, \cdots, I1_{x_1-1}\} \cup \{I2_{x_2}, I2_{x_2+1}, \cdots, I2_m\} \tag{4.6}$$

$$P_{neu2} = \{I2_0, I2_1, \cdots, I2_{x_2-1}\} \cup \{I1_{x_1}, I1_{x_1+1}, \cdots, I1_n\} \tag{4.7}$$

Bei der Rekombination kann es sehr schnell zu einer Überschreitung der maximalen Programmlänge kommen, daher wird auch bei der Rekombination die Länge der Programme stets überprüft.

4.5.8 Import und Export von Programmen und Individuen

Das GP System unterstützt den Im- und Export von Programmen oder ganzer Individuen. Somit besteht die Möglichkeit, ein Programm oder ein ganzes Individuum manuell mit einem Texteditor zu erstellen und in ein Experiment einzufügen. Das Format, in dem die Programme oder Individuen vom System importiert werden können, entspricht dem Speicherformat, welches verwendet wird, wenn Programme und Individuen (im Rahmen eines Experiments) auf der Festplatte gespeichert werden. Eine genauere Anleitung für das manuelle Erstellen von Programmen beinhaltet das Kapitel 6.5.

4.6 Bewertung von Individuen und Fitnessfunktion

Ein wichiger Bestandteil der Genetischen Programmierung ist die sogenannte Fitnessfunktion. Durch die Fitnessfunktion werden genetisch evolvierte Programme, die Individuen eines GP-Systems, in ihrer Güte bewertet. Den Individuen wird durch die Fitnessfunktion ein Zahlenwert zugewiesen, der aussagen soll, wie gut ein evolviertes Programm eine Lösung für das dem GP-System gestellten Problem darstellt. Mit diesem Wert können nun Individuen hinsichtlich ihrer Qualität miteinander verglichen werden, was eine Vorraussetzung für die Selektion von Individuen für die darauf folgenden genetischen Operationen darstellt.

Die Fitnessfunktion steht in engem Zusammenhang mit dem durch Genetisches Programmieren zu lösenden Problem. Um eine sinnvolle Fitnessfunktion schreiben zu können, sollten so viele Informationen wie möglich über das Problem betrachtet werden. Die Komplexität der Fitnessfunktion wird zwar nicht zwingend durch die Problemkomplexität beeinflusst, viel mehr wird die Komplexität der Fitnessfunktion durch die Anforderungen des Anwenders an die Lösung bestimmt. Diese Überlegung führte dazu, dass eine Untergruppe der PG368 einen interdisziplinären Vorstoß unternahm und bei Herrn Prof. Dr. S. Starischka, Mitglied des Fachbereich Sport der Universität Dortmund, Leiter des Arbeitsbereichs 2 *Training und Bewegung* um fachlichen Rat bat. Dort traf die Anfrage zwar auf Gegeninteresse, aber leider waren die ausgetauschten Informationen nicht von großem Nutzen. Sie bezogen sich auf Ergebnisse aus dem Forschungsgebiet der Biomechanik. Die dort gewonnenen Erkenntnisse waren aber für die PG368 nicht relevant, weil sie sich stark an einzelnen kinematischen Modellen orientierten, wie zum Beispiel dem des menschlichen Körpers.

Die Fitnessfunktionen des SIGEL GP Systems müssen sich laut den Anforderungen an das System für beliebige Robotermodelle eignen. Dies schränkt die Anzahl der möglichen Bewertungskriterien ein. Eine Fitnessfunktion, die auf ein

Robotermodell genau zugeschitten ist und mit diesem zu einem guten Evolutionsverlauf führt, kann mit einem anderen Robotermodell zu ganz anderen Ergebnissen kommen. Dies wird durch die Bewertungskriterien der Fitnessfunktion bedingt, die für das betrachtete Robotermodell speziell ausgewählt wurden. Diese speziellen Kriterien müssen sich für andere Robotermodell nicht unbedingt als sinnvoll erweisen. Allgemeine Bewertungskriterien widerum lassen dem SIGEL GP System viel Freiraum für *Nebeneffekte*, die ein eigentlich gutes Evolutionsergebnis für den Anwender unbrauchbar machen können. Es bleibt also das Problem, entweder eine Fitnessfunktion zu schreiben, die nur sinnvoll für ein Robotermodell eingesetzt werden kann oder eine Fitnessfunktion zu erstellen, die sich zwar für beliebige Robotermodelle eignet, aber in ihren Bewertungskriterien zu ungenau ist, um genau so effektiv zu sein bei einem bestimmten Robotermodell, wie eine Fitnessfunktion, die nur auf dieses bestimmte Modell zugeschnitten wurde. Die Fitnessfunktionen im SIGEL GP System sind alle auf beliebige Robotermodelle ausgelegt. So bleibt es dem Anwender überlassen eine Fitnessfunktion zu erstellen, in die er weitere Kriterien zur Individuenbewertung einfließen läßt, die nur in Verbindung mit dem von ihm betrachteten Robotermodell sinnvoll sind. Dazu kann er sich z.B. an der Klasse *SIG_SimpleFitnessFunction* orientieren, die einzelnen Daten können über den *SIG_Recorder* aufgezeichnet und ausgewertet werden.

Bevor auf die Unterschiede der verschiedenen Fitnessfuntkionen eingegangen wird, sollen zuerst noch ihre Gemeinsamkeiten beschrieben werden. Als Eingabe bekommen alle Fitnessfunktionen im SIGEL GP System die gleichen Daten:

- Ein Individuum, dessen Roboterkontrollprogramm bezüglich seiner Güte bewertet werden soll.
- Ein Robotermodell, das von dem Roboterkontrollprogramm zum Laufen gebracht werden soll.
- Die Umgebungsparameter aus dem Umgebung-Einstellungsfenster des SIGEL Systems.
- Die Simulationsparameter aus dem Simulationsparameter-Enstellungsfenster des SIGEL Systems.

Alle Fitnessfunktionen starten eine Simulation, gemäß der übergebenen Umgebungs- und Simulationsparameter, um dort ein Robotermodell durch ein Roboterkontrollprogramm steuern zu lassen. Dabei können Daten bezüglich der Position einzelner Glieder, der Gelenkausrichtungen und der ansetzenden Kräfte in jedem *Zeitschritt* gespeichert werden. Diese Daten können dann dazu benutzt werden, um die Fitness des betrachteten Individuums durch die Fitnessfunktion zu berechnen.

4.6.1 SimpleFitnessFunction

Die SIMPLEFITNESSFUNCTION ist die einfachste Variante der implementierten Fitnessfunktionen. Sie speichert die Position des Robotermodells beim Simulationsstart und seine Position am Simulationsende. Aus diesen Daten wird die Länge

der Wegstrecke errechnet, die der simulierte Roboter sozusagen in LUFTLINIE zurückgelegt hat. Dieser Wert wird durch die Dauer des Simulationslaufs geteilt und das Ergebnis dem zu bewertenden Individuum als Fitnesswert übergeben. Die SIMPLEFITNESSFUNCTION benutzt also als Bewertungskriterium die durchschnittliche Geschwindigkeit des Robotermodells während der Simulation, wobei angenommen wird, dass der Roboter sich auf der gedachten Linie zwischen Start- und Endpunkt bewegt hat. Die tatsächliche Bewegung (und damit die tatsächliche Geschwindigkeit) kann mit dieser Fitnessfunktion nicht bewertet werden, weil *geradliniges* Laufen am besten und *kurvenreiches* Laufen am schlechtesten ist.

4.6.2 RealSpeedFitnessFunktion

Die REALSPEEDFITNESSFUNKTION ist eine Weiterentwicklung der SIMPLEFITNESSFUNCTION. Im Unterschied zu ihrer einfachen Variante, legt die REALSPEEDFITNESSFUNKTION nicht den Abstand zwischen Start- und Endpunkt des Robotermodells im Simulationslauf zur Geschwindigkeitsberechnung zu Grunde, sondern den tatsächlich zurückgelegten Weg. Dazu speichert sie die Positionsveränderung des Robotermodells jede halbe Sekunde und benutzt dann die Summe dieser Wegstrecken zur Berechnung der Geschwindigkeit des Robotermodells im Simulationslauf. Die REALSPEEDFITNESSFUNKTION stellt somit eine Vergröberung der SIMPLEFITNESSFUNKTION dar. Individuen, deren Roboterkontrollprogramme das Robotermodell zwar von der Summe der Wegstrecke her gleich weit bewegen, sich aber in ihrer Gradlinigkeit unterscheiden, werden mit dem gleichen Fitnesswert bewertet. Somit wird jede *positionsverändernde* Bewegung als eine *erfolgreiche* Bewegung bewertet.

4.6.3 NiceWalkingFitnessFunction

Die NICEWALKINGFITNESSFUNKTION ist, wie die REALSPEEDFITNESSFUNKTION, auch eine Weiterentwicklung der SIMPLEFITNESSFUNCTION. Mit ihr wird versucht, Roboterkontrollprogramme zu evolvieren, die eine erwünschte Körperhaltung beim Robotermodell hervorrufen. Es werden nur Individuen mit einem Fitnesswert größer Null bewertet, deren Kontrollprogramme es schaffen, den Torso des Robotermodells auf eine bestimmte Höhe zu halten. Diese Höhe wird durch die Position des Torsos zu Beginn der Simulation festgelegt und hat einen Toleranzbereich von einem halben Meter oberhalb und unterhalb der Anfangshöhe. Verlässt der Torso im Simulationslauf auch nur für einen ZEITSCHRITT den Toleranzbereich, so wird der zurückgegebene Fitnesswert auf Null gesetzt. Bleibt aber der Torso im Toleranzbereich, so wird die Fitnesswert, wie bei der SIMPLEFITNESSFUNCTION berechnet. Dies stellt eine Verfeinerung der SIMPLEFITNESSFUNCTION dar, weil Individuen mit Kontrollprogrammen, die das Robotermodell mehr robbend als laufend vorwärts bewegen, radikal aussortiert werden.

Kapitel 5

Ergebnisse

In diesem Kapitel werden die Ergebnisse zusammengefasst, die mit unserem System erzielt worden sind. Die durchgeführten Experimente sind mit unterschiedlichen Untersuchungsschwerpunkten durchgeführt worden, welche sich allgemein in die Grundlagenexperimente mit Fokus GP (Kapitel 5.1 - 5.4) und die Fortgeschrittende Experimente - Fokus Robotermodell / Fitnessfunktion (Kapitel 5.5 - 5.10) unterteilen.

Bei den Grundlagenexperimenten war der Fokus mehr auf die Fähigkeiten des implementierten GP Systems gerichtet. Die zentrale Frage dieser Experimente war, ob unser GP System überhaupt fähig ist, ein Steuerungsprogramm zu evolvieren. Weitere grundlegende Experimente sollten untersuchen, welchen Einfluss die Variation der genetischen Parameter auf die Ergebnisse unseres Systems hat und wie weit diese Ergebnisse mit bekannten Theorien der Genetischen Programmierung vergleichbar sind. Um alle Ergebnisse der grundlegenden Experimente vergleichen zu können, wurde allen grundlegenden Experimenten ein sehr einfaches Robotermodell und eine sehr einfache Fitnessfunktion zugrunde gelegt.

Diesen Experimenten folgen die fortgeschrittenen Untersuchungen, deren Fokus auf das Robotermodell und auf komplexere Fitnessfunktionen gerichtet ist. Die Komplexität dieser Modelle ist im Vergleich zu den einführenden Experimenten sehr hoch. Es wurden verschiedene Modelle gewählt, deren Bewegung unterschiedliche Bewegungsabläufe erfordern. Weiterhin wurde bei diesen Experimenten die Fitnessfunktion variiert, um zu untersuchen, welchen Einfluss eine komplexere Fitnessfunktion auf die Qualität des Evolutionsergebnisses hat.

5.1 Funktionalität mit hoher Crossoverrate

Im Rahmen dieses Experiments soll für ein sehr einfaches Robotermodell eine Fortbewegung entwickelt werden. Das Robotermodell, welches nur aus zwei (mit einem Drehgelenk verbundenen) Gliedern besteht, ist bewusst sehr einfach gehalten worden. Durch dieses Experiment sollen eher die grundlegenden Fähigkeiten des GP Systems untersucht werden. Es soll die Fragestellung untersucht werden,

Länge	3,0 m	Breite	1,0 m
Höhe	1,0 m		
Drehgelenke	1	davon motorisiert	1
Schubgelenke	0	davon motorisiert	0
Sensoren	0	Anzahl Beine	0

Tabelle 5.1: Ein sehr einfaches Robotermodell

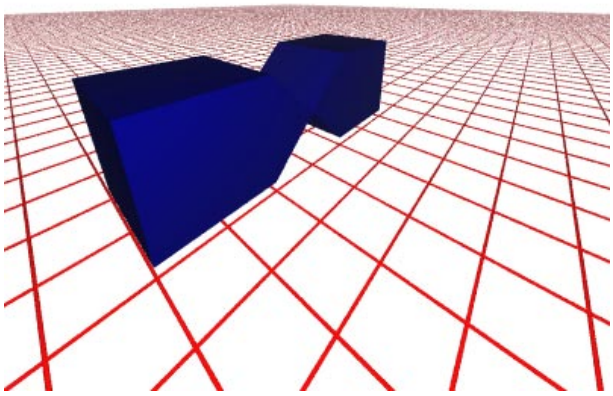


Abbildung 5.1: Ein einfaches Robotermodell für grundlegende Versuche

ob das implementierte GP System in der Lage ist, *überhaupt* eine Fortbewegung und zwar bereits für ein sehr einfaches Robotermodell zu entwickeln. Wenn diese Frage positiv beantwortet werden kann, soll untersucht werden, *welche* Bewegung aus der Evolution hervorgegangen ist, und *wie* die Evolution dieser Bewegung zu bewerten ist. Ebenfalls kann bei einem positiven Ergebnis im Rahmen späterer Versuche auf komplexere Robotermodelle eingegangen werden.

5.1.1 Roboter

Abb. 5.1 zeigt das verwendete sehr einfache Robotermodell. Dabei handelt es sich lediglich um ein Modell mit zwei Gliedern, welche mit einem Gelenk verbunden sind. Die folgenden Maßangaben sind in Tab. 5.1 ersichtlich.

Library	DynaMechs	Step size	0.01
Time to Sim.	3 Min.	Random Seed	0
Integrator	RK4	Joint fric. const.	0.35
Joint lim. spr. const.	25000	Joint lim. damp. const.	2500

Tabelle 5.2: Simulationsparameter für grundlegende Versuche

Gravitation	$(0;-9.81;0)$
Start Position	$(0;1;0)$
Plan. spr. const.	5500
Norm. spr.const.	7500
Plan. damp. const.	50
Norm. damp. const.	50
Stat. fric. coeff.	1.5
Kin. fric. coeff.	1

Tabelle 5.3: Die Umgebung für grundlegende Versuche

5.1.2 Umgebung

Die Simulationsseinstellungen (Tab. 5.2) entsprechen weitgehend den Standardeinstellungen. Lediglich die Simulationszeit wurde von 10 Sekunden auf 3 Minuten erhöht und verschiedene Konstanten (Joint, Spring and Damp Constant) entsprechend dem Robotermodell und den wirkenden Kräften angepasst. Bei der Umgebung (Tab. 5.3) wurden die Standardeinstellungen vollständig übernommen.

5.1.3 GP

Die GP-Einstellungen (Tabelle 5.5) entsprechen weitgehend den Standardeinstellungen. Im Rahmen dieses Experiments ist der Crossoveroperator der dominante Operator. Der Mutation wird die Rolle eines Hintergrundoperators zuteil. Als Befehle waren alle Befehle außer NOP und JMP zugelassen (s. Tabelle 5.4).

Com.	Dur.	Prob.	Com.	Dur.	Prob.	Com.	Dur.	Prob.
ADD	0.001	500	JMP	n/a	0	MOVE	0.01	500
CMP	0.001	500	LOAD	0.001	500	MUL	0.001	500
COPY	0.001	500	MAX	0.001	500	NOP	n/a	0
DELAY	0.001	500	MIN	0.001	500	SENSE	0.001	500
DIV	0.001	500	MOD	0.001	500	SUB	0.001	500

Tabelle 5.4: Die Sprachparameter der Grundlagenexperimente

Crossover	67,0%	Min. Prg. length	10
Mutation	5,0%	Max. Prg. length	1024
Reproduction	34,0%	Maximal age	50
Fitnessfunction	simpleFitnessFunction	Random seed	0
Tournmts. per Gen.	500	No. of Individuals	100

Tabelle 5.5: Die GP-Parameter des Grundlagenexperiments mit hoher Crossoverrate

5.1.4 Evolution

Insgesamt kamen acht Rechner (unter Solaris) im Rahmen der PVM zum Einsatz, auf welchen maximal je zwei Prozesse laufen durften. Die Evolutionszeit betrug insgesamt 10 Stunden. Dieser Prozess war weitgehend durch cron - Jobs automatisiert und wurde automatisch abends um 21.00 Uhr gestartet und morgens um 7.00 Uhr beendet.

5.1.5 Ergebnisse

Die evolvierte Bewegung gleicht einer sprunghaften sinusförmigen Bewegung (s. auch Abb. 5.3) mit einer Geschwindigkeit von 0,84 m/s. Beim Betrachten der Simulation wird deutlich, dass sich diese Bewegung jedoch erst entwickeln muss. Werden zu Beginn beide Glieder gleichzeitg auf- und abbewegt, so scheint sich die Kraft auf beide Glieder im Laufe der Simulation ungleich zu entwickeln. Wenige (Simulations-) Sekunden nach der kontinuierlichen Auf- und Abbewegung ist die Kraft, welche auf eines der Glieder wirkt, ausreichend, das Robotermodell eine sprunghafte Bewegung (nach rechts) vollziehen zu lassen. Bei dieser sprunghaften Bewegung wird die Auf- und Abbewegung jedoch fortgesetzt. Im Gegensatz zur vorherigen Situation, bei der beide Glieder den Boden stets berührten, ergeben sich für die Simulation jedoch neue Kräfteentwicklungen. Da nun ein Glied nicht mehr den Boden berührt, existieren auch keine direkten Gegenkräfte mehr und es entwickelt sich eine Art Schwungkraft, die der Bewegung Nachdruck verleiht. Durch die Schwungkraft werden zusätzliche Kräfte erzeugt, welche ausreichen, die entwickelte springende Bewegung aufrechtzuerhalten und stetig fortzusetzen. Interessant ist jedoch der Richtungswechsel dieser Bewegung in die entgegengesetzte Richtung. Nach einer Simulationszeit von ca. 4 Minuten kommt es zu einer Umkehrung der Bewegungsrichtung durch sich akkumulierende Kräfte auf die Glieder. Es hat den Anschein, als verbleibe das Robotermodell in diesem Bewegungszyklus (mit sich stets wechselnder Bewegungsrichtung).

Die Evolution dieser Bewegung ist durch die Fitnesskurve charakterisiert (Abb. 5.2). Die Qualität des besten Individuums wird stetig verbessert, wie es im Rahmen von unserer Turnierselektion auch zu erwarten ist, da gut Individuen niemals gelöscht werden. Zu unserer Überraschung ist die Entwicklung vielleicht

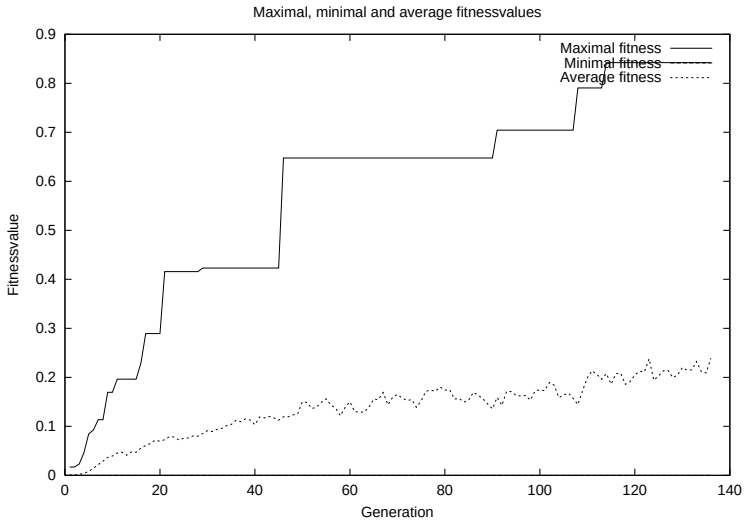


Abbildung 5.2: GP mit hoher Crossoverrate und einfachem Robotermodell

etwas langsamer als erwartet. Die durschnittliche Fitness bleibt auf einem recht niedrigen Niveau. Diese Entwicklung ist auf die Eigenschaften des hier dominanten Crossoveroperators (s. auch Kapitel 2.1.3) zurückzuführen, welcher eine relativ hohe Diversität der Individuen erzeugen kann.

5.1.6 Fazit

Das Ergebnis, einen sich bewegenden Roboter erhalten zu haben, ist zwar nicht unbedingt überraschend, aber doch sehr erfreulich, da unsere Erwartungen bzgl. der *Bewegungsart bzw. Systematik* weit übertroffen worden sind. Andere Experimente (früherer Versionen) haben zwar auch Bewegungen evolviert, jedoch handelte es sich bei diesen Bewegungen um weitaus unkoordiniertere bzw. weitaus weniger *elegante* Bewegungen. Als das wohl wichtigste Ergebnis kann hervorgehoben werden, dass unser GP System in der Lage ist, Bewegungsabläufe für ein Robotermodell zu entwickeln. Da die Funktionsfähigkeit unseres GP Systems nun experimentell gezeigt worden ist, können nun weitere Experimente folgen, welche die Einflüsse verschiedener Parameter (bei gleichbleibend einfachem Robotermodell) analysieren und herausstellen.

5.2 Evolution mit hoher Mutationsrate

Da die Evolution mit einer hohen Crossoverrate und einer untergeordneten Mutation ein sehr gutes Ergebnis hervorgebracht hat, ist ebenfalls die Frage interessant, ob eine Evolution mit dominierender Mutation und untergeordnetem Crossover

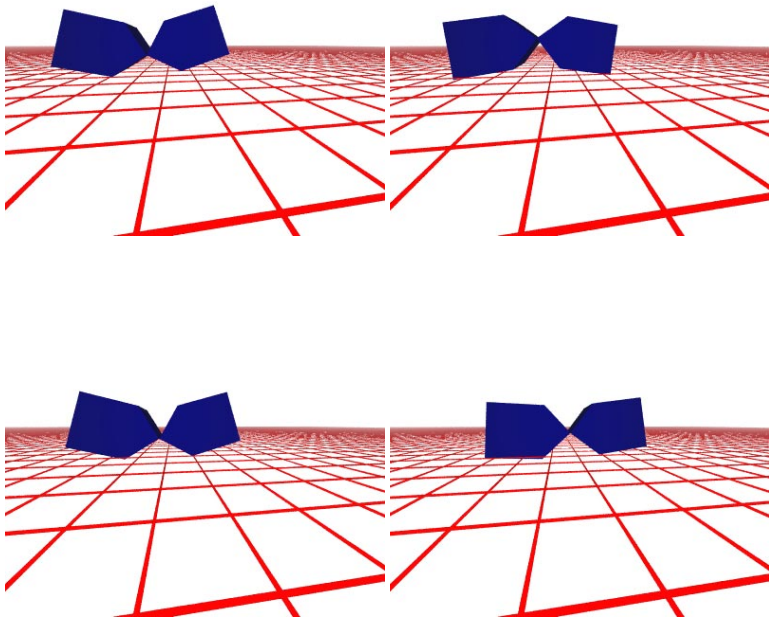


Abbildung 5.3: Der Bewegungsablauf in Ausschnitten

ebenfalls erfolgreich ist. Diese Frage soll das in diesem Kapitel beschriebene Experiment beantworten. Im Rahmen dieses Experiments soll für ein sehr einfaches Robotermodell eine Fortbewegung entwickelt werden. Beim verwendeten Robotermodell handelt es sich um das bereits unter Kapitel 5.1 verwendete Robotermodell, um die im Rahmen dieses Experiments evolvierten Ergebnisse mit denen des aktuellen vergleichen zu können.

5.2.1 Roboter

Abb. 5.1 zeigt das verwendete sehr einfache bereits erläuterte Robotermodell. Da dieses nun wieder verwendetes Modell bereits in Kapitel 5.1 näher erläutert wurde, soll an dieser Stelle auf eine Wiederholung der Robotereigenschaften verzichtet werden. Für Details sei auf Tab. 5.1 und Abb. 5.1 verwiesen.

5.2.2 Umgebung

Um die Ergebnisse vergleichen zu können, wurden ebenfalls die gleichen Umgebungs- und Simulationseinstellungen verwendet, wie sie in Tab. 5.2 und Tab.

Crossover	5,0%	Min. Prg. length	10
Mutation	80,0%	Max. Prg. length	1024
Reproduction	15,0%	Maximal age	50
Fitnessfunction	simpleFitnessFunction	Random seed	0
Tournmts. per Gen.	500	No. of Individuals	100

Tabelle 5.6: Die GP-Parameter des Grundlagenexperiments mit hoher Mutation

5.3 beschrieben sind. Für Details sei auch an dieser Stelle auf das vorangegangene Kapitel hingewiesen.

5.2.3 GP

Als Befehle waren (wie unter 5.1) alle Befehle ausser NOP und JMP zugelassen (s. Tabelle 5.4). Die GP-Einstellungen (Tabelle 5.6) entsprechen weitgehend den Standardeinstellungen. Im Rahmen dieses Experiments ist der Mutationsoperator der dominante Operator. Der Crossoveroperator arbeitet nur im Hintergrund.

5.2.4 Evolution

Insgesamt kamen wieder acht Rechner (unter Solaris) im Rahmen der PVM zum Einsatz, auf welchen maximal zwei Prozesse laufen durften. Die Evolutionszeit betrug insgesamt 10 Stunden. Dieser Prozess war ebenfalls weitgehend durch cron-Jobs automatisiert, und wurde automatisch abends um 21.00 Uhr gestartet und morgens um 7.00 Uhr beendet.

5.2.5 Ergebnisse

Das Erstaunliche dieses Ergebnisses ist, dass die Systematik der evolvierten Bewegung prinzipiell der aus Kapitel 5.2 gleicht. Der Unterschied leigt jedoch in der Geschwindigkeit des Roboters, welche in diesem Experiment (mit dominierender Mutation) wesentlich höher, und zwar bei 1,03 m/s, liegt. Auch scheint die Bewegung insgesamt etwas robuster und stabiler. Ebenfalls kommt es nicht zu einer Umkehrung der Bewegungsrichtung.
Die Evolution dieser Bewegung ist wieder durch die Fitnesskurve charakterisiert (Abb. 5.4). Die Qualität des besten Individuums wird stetig verbessert, wie es im vorangegangenen Experiment auch bereits der Fall war. Der entscheidende Unterschied zu Abb. 5.2 liegt jedoch in der Geschwindigkeit der Individuenentwicklung. Eine Qualitätszunahme erfolgt sehr schnell. Dieses Ergebnis ist vielleicht etwas überraschend, da Änderungen durch den dominierenden Mutationsoperator meist nur punktuell vorgenommen werden (s. auch Kapitel 4.5.7), und daher Änderungen am Genom des Individuums in nicht so großem Ausmaß wie beim Crossover stattfinden. Den Einfluss der Mutation auf das beste Individuum im Gegensatz zum

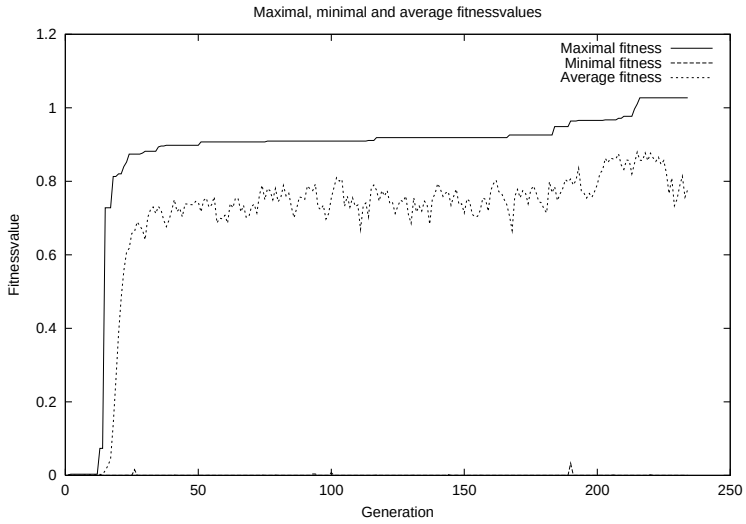


Abbildung 5.4: GP mit hoher Mutationsrate und einfachem Robotermodell

Crossover ist ebenfalls an der *Glätte* der Kurve zu erkennen. Durch die punktuellen (und damit verbundenen geringfügigen) Änderungen sind die Fitnesssteigerungen des besten Individuums (von der Anfangsphase abgesehen) nicht allzu sprunghaft, wie es durchaus während der gesamten Evolution mit dominierendem Crossover der Fall war. Die durchschnittliche Fitness ist wie erwartet relativ hoch (bei gleicher Generation wie im vorhergehenden Experiment ca. 0,9 m/s), da die relativ geringfügigen Änderungen durch die Mutation in den meisten Fällen einen nicht sehr grossen negativen Einfluss auf die Gesamtfitness eines Individuums haben. Nachteilig ist jedoch die dadurch ebenfalls nachlassende Diversität der Population.

5.2.6 Fazit

Das Ergebnis ist ebenfalls sehr erfreulich, da auch *nur* mit der Mutation und einem stark reduziertem Crossover ein sehr gutes Ergebnis erzielt worden ist. Viele allgemeine GP theoretische Aspekte (wie der Einfluss eines dominierenden Crossovers oder einer dominierenden Mutation) konnten in den Ergebnissen unseres GP Systems wiedererkannt werden, was auf ein funktionierendes GP System schließen lässt. Erfreulich ist auch, dass mit diesem Ergebnis das Ergebnis der ersten Evolution mit dominierendem Crossover (rein subjektiv wegen der längeren Evolution) verbessert werden konnte.

5.3 Evolution mit importierten Individuen

Das erste Experiment mit dominierendem Crossover hat zwar eine gute Bewegungssystematik hervorgebracht, jedoch war die Evolutionszeit für eine so ein-

fache Bewegung relativ hoch. Wie bereits in Kapitel 4.5.8 erwähnt, stellt das GP System Funktionen zur Verfügung, die einen Import von Individuen (mit Programmen) erlauben. Im Rahmen dieses Experiments sollen alle Systemeinstellungen identisch zu denen des ersten Experiments mit dominierendem Crossover (aus Kapitel 5.1) sein, inklusive der Ausgangspopulation, was wegen des gleichen Random Seeds den *gleichen* Evolutionsverlauf zur Folge hätte. Es sollen nun eine Zahl von Individuen (mit hoher Fitness) aus älteren Experimenten importiert werden. Im Anschluss wird untersucht, welchen Einfluss die importierten Individuen auf die Evolution hatten, und ob sie die Evolution wirklich beschleunigen konnten.

5.3.1 Roboter, Umgebung und Evolution

Diese Einstellungen wurden aus Kapitel 5.1 vollständig übernommen.

5.3.2 GP

Alle Einstellungen, inklusive der Ausgangspopulation entsprechen denen des Kapitel 5.1. Zusätzlich wurden Individuen mit dem Ziel importiert, die Evolutionszeit positiv zu beeinflussen. Insgesamt wurden zwei Individuen in je zehnfacher Ausführung importiert. Diese Individuen stammen aus älteren (hier nicht dokumentierten) Experimenten. Eines dieser Individuen beschreibt eine Bewegung, welche der unter 5.1 beschriebenen *sinusförmigen* Bewegung ähnlich ist. Das andere Individuum stammt jedoch aus einem Experiment, dessen bestes Individuum sehr sprunghafte Bewegungen machte. Diese Experimente wurden jedoch mit einer wesentlich älteren SIGEL Version und in einer anderen Umgebung durchgeführt, so dass die Ergebnisse dieser Experimente mit dem aktuellen nicht direkt vergleichbar sind. Die Bewegungsabläufe, welche in den Programmen der Individuen kodiert sind, können jedoch sehr wohl (vielleicht unter Einfluss geringer evolutionärer Änderungen) von der aktuellen Evolution übernommen werden. Natürlich ist die Frage offensichtlich, warum nicht gleich das beste Individuum, welches bereits aus 5.1 hervorgegangen ist, importiert wird: Das wäre natürlich ohne Weiteres möglich, nur würde dies eine einfache *Fortsetzung* eines Experiments im *gleichen* evolutionären Kontext bedeuten. Im Rahmen dieses Experiments soll eher untersucht werden, ob *ähnliche* evolutionäre Ergebnisse (codiert in Form eines Individuums) aus einem völlig *unterschiedlichen* Kontext in einen anderen Kontext übertragen werden können. Mit dem importierten Individuum wird nicht nur das Programm übertragen, sondern auch eine evolutionäre Strategie, welche durch das Programm kodiert wird. Für die GP ergeben sich somit die Einstellungen aus Tabelle 5.7

5.3.3 Ergebnisse

Um beide Ergebnisse direkt miteinander vergleichen zu können, sei das Ergebnis von Kapitel 5.1 noch einmal in Abbildung 5.5 dargestellt. Auch die aktuelle Evo-

Crossover	67,0%	Min. Prg. length	10
Mutation	5,0%	Max. Prg. length	1024
Reproduction	34,0%	Maximal age	50
Fitnessfunction	simpleFitnessFunction	Random seed	0
Tournmts. per Gen.	500	No. of Individuals	120

Tabelle 5.7: Die GP-Parameter für das Experiment mit importierten Individuen

lution mit importierten Individuen brachte eine Bewegung hervor, wie sie bereits im ersten Experiment beschrieben worden ist. Dieses Ergebnis ist nicht überraschend, der Fokus dieses Experiments lag eher auf der Evolutionszeit, obwohl das jetzige Ergbnis mit 1,15 m/s wesentlich besser ausfällt als ohne importierte Individuen! Wie bei einem direkten Vergleich der Abbildungen unter 5.5 gesehen werden kann, ist die Evolution durch den Import der Individuen tatsächlich zeitlich positiv beeinflusst worden, indem die Qualitätszunahme schneller erfolgt als in dem Experiment, in welches keine Individuen importiert worden sind.

5.3.4 Fazit

Dieses Ergbnis lässt die Vermutung zu, das erfolgreiche evolutionäre Strategien eines fremden Kontexts an einen anderen Kontext adaptiert werden können. Damit könnte sich die Importfunktion unseres GP Systems durchaus als eine sehr hilfreiche Möglichkeit herausstellen. Weiterhin ist damit ebenfalls die Leistungs- und Anpassungsfähigkeit von GP Systemen eindrucksvoll gezeigt. Die Tatsache, dass das Endergebnis sogar noch besser ausgefallen ist, als im Vergleichsexperiment, unterstützt die aufgestellten Thesen. Durch den Import von evolutionären Ergebnissen ist der aktuellen Evolution scheinbar der richtige Weg gewiesen bzw. angedeutet worden.

5.4 Evolutionen mit reduzierten Roboterbefehlssätzen

Im Rahmen dieser Experimente soll für ein sehr einfaches Robotermodell eine Fortbewegung entwickelt werden. Auch bei diesen Experimenten handelt es sich wieder um den bereits sehr häufig verwendeten sehr einfachen zweigliedrigen Roboter. Unter Verwendung der klassischen GP Evolutionsparameter (hohe Crossoverrate, geringe Mutationsrate) soll in diesen Experimenten der verwendete Instruktionssatz reduziert werden. Dabei sollen nach Möglichkeit die Fragen beantwortet werden, wie weit der Instruktionssatz des Roboters verringert werden kann, um noch sinnvolle Bewegungsabläufe evolvieren zu können.

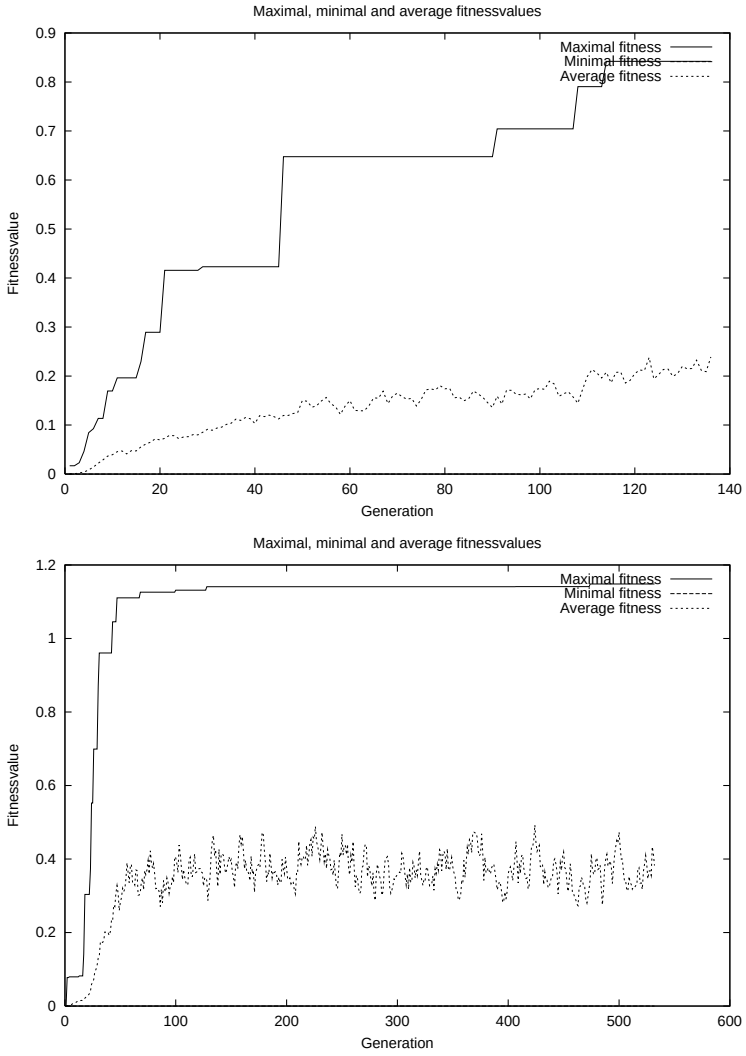


Abbildung 5.5: Oben: Langsamere Evolution mit dominierendem Crossover (wie Abb. 5.1), unten: Evolution wie unter 5.1, jedoch mit 20 importierten Individuen

Com.	Dur.	Prob.	Com.	Dur.	Prob.	Com.	Dur.	Prob.
ADD	0.001	500	JMP	n/a	0	MOVE	0.01	500
CMP	n/a	0	LOAD	0.001	500	MUL	0.001	500
COPY	0.001	500	MAX	n/a	0	NOP	n/a	0
DELAY	n/a	0	MIN	n/a	0	SENSE	0.001	500
DIV	0.001	500	MOD	n/a	0	SUB	0.001	500

Tabelle 5.8: Reduzierter Roboterbefehlssatz der Grundlagenexperimente

Com.	Dur.	Prob.	Com.	Dur.	Prob.	Com.	Dur.	Prob.
ADD	0.001	500	JMP	n/a	500	MOVE	0.01	500
CMP	n/a	500	LOAD	0.001	500	MUL	n/a	500
COPY	0.001	500	MAX	n/a	500	NOP	n/a	500
DELAY	n/a	500	MIN	n/a	500	SENSE	0.001	500
DIV	n/a	500	MOD	n/a	500	SUB	n/a	500

Tabelle 5.9: Stark reduzierter Roboterbefehlssatz der Grundlagenexperimente

5.4.1 Roboter, Umgebung und GP

Das zweigliedrige Robotermodell und die Umgebung entsprechen denen aus Kapitel 5.1. Die GP Einstellungen gleichen denen aus Tabelle 5.5. Für die Instruktionsmenge des Roboter wurden im ersten Experiment die Einstellungen aus der Tabelle 5.8 vorgenommen.

Weiterhin wurde der Befehlssatz in einem weiteren Experiment (unter den gleichen Anfangsbedingungen und Einstellungen wie oben) stärker reduziert. Die Einstellungen können der Tabelle 5.9 entnommen werden.

5.4.2 Evolution

Insgesamt kamen acht Rechner (unter Solaris) im Rahmen der PVM zum Einsatz, auf welchen maximal zwei Prozesse laufen durften. Die Evolutionszeit betrug insgesamt 10 Stunden pro Experiment. Die Experimente wurden jeweils (an unterschiedlichen Tagen) abends um 21.00 Uhr gestartet und morgens um 7.00 Uhr beendet.

5.4.3 Ergebnisse

Das Experiment mit dem reduzierten Befehlssatz (s. Tabelle 5.8) brachte ebenfalls eine Bewegung hervor, die in ihre Systematik mit der aus dem ersten Experiment (s. 5.1) vergleichbar ist. Die Evolution kommt sogar sehr schnell zu guten Ergebnissen (s. Abb. 5.6). Das nach 10 Stunden beste Ergebnis mit einer Laufgeschwindigkeit von 0,94 m/s kann durchaus mit den bislang besten Ergebnissen mithalten

und ist sogar noch besser als das ursprüngliche Experiment mit dominierendem Crossoveroperator und nahezu vollem Befehlssatz. Dies lässt die Vermutung zu, dass einige Befehle für die Entwicklung einer Bewegung nicht essentiell sind. Die Erweiterung dieser Vermutung könnte sein, dass einige Befehle die Evolution sogar eher verzögern als beschleunigen. Diese Vermutung liegt darin begründet, dass, wenn genau die *nicht essenziellen* Befehle nicht verwendet werden, die Evolution schneller zu Ergebnissen kommt, da die Wahrscheinlichkeit für vorteilhafte Kombinationen von Befehlen einfach höher ist. Daher wurde der oben unter Abb. 5.6 verwendete Befehlssatz noch weiter reduziert (s. Tabelle 5.9) , und das Ergebnis ist in unten inAbb. 5.6 zu sehen. Mit dem Verzicht auf grundlegende arithmetische Funktionen (Ausnahme: Addition) wird die Evolution sehr stark eingeschränkt. Das Ergebnis ist zwar ebenfalls ein sich bewegender Roboter, jedoch ist die Bewegung relativ langsam und erfolgt stark verzögert. Starke ruckartige Bewegungen sind kaum zu erkennen. Die Bewegung gleicht einem leichten Hüpfen. Mit dem besten Individuum wurde nur eine Geschwindgikeit von 0,57 m/s erreicht. Die Geschwindigkeit der Evolution ist sehr langsam. Ebenfalls vollzieht sich die Entwicklung in eher kleinen Schritten, was insgesamt zu dem Ergebnis führt, dass eine Evolution durchaus mit einem vollständigen Satz arithmetischer Funktionen durchgeführt werden soll. Mit grosser Wahrscheinlichkeit handelt es sich bei den arithmetischen Funktion um *essenzielle* Funktionen.

5.4.4 Fazit

Zu unserer Überraschung scheinen einige Roboterbefehle die Evolution nicht stark zu beeinflussen, da das Ergebnis selbst mit einem eingeschränkten Befehlssatz, welcher nur auf arithmetische Funktionen und grundlegende Roboterbefehle beschränkt war, noch sehr gut war. Eine weitere Einschränkung der arithmetischen Funktionen war nicht mehr sinnvoll. Grundsätzlich sei angemerkt, dass die GP in der Lage ist, nicht essenzielle Operationen herauszufiltern. Jedoch verschafft eine Kenntnis des essentiellen Instruktionssatzes einen enormen zeitlichen Vorteil. Wenn der Evolution dieses Wissen vorgegeben werden kann, kann der dadurch gewonnene Zeitvorteil bereits für komplexere evolutionäre Aufgabenstellungen verwendet werden. Der Grund für diese Geschwindigkeitszunahme ist der verkleinerte Suchraum.

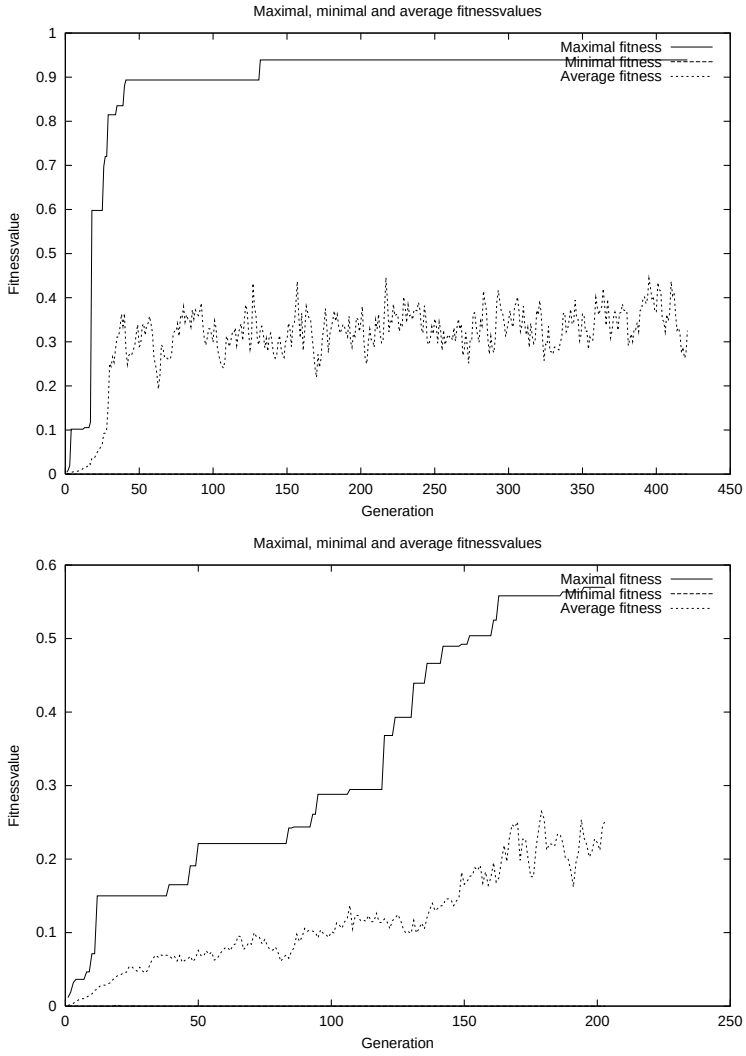


Abbildung 5.6: Ergebnisse, oben: Mit reduziertem Befehlssatz, vorwiegend arithmetische Funktionen, unten: Ergebnis mit stark reduziertem Befehlssatz, vorwiegend Addition

5.5 Evolution eines raupenähnlichen Roboters

In diesem Experiment wurden Laufsteuerungsprogramme für einen Roboter evolviert, der aus einer einfachen kinematischen Kette besteht. Die Hoffnung war, dass der Roboter sich durch eine raupenähnliche Bewegung fortbewegt.

5.5.1 Roboter

Er besteht aus einem würfelförmigen Torsoglied, drei Zwischengliedern und einem abgerundeten Endglied. Jedes Zwischenglied besteht aus zwei um 90 Grad gegeneinander versetzten Prismen. Alle Gelenke sind Rotationsgelenke. Weitere Details können der Tabelle 5.10 und Abbildung 5.7 entnommen werden.

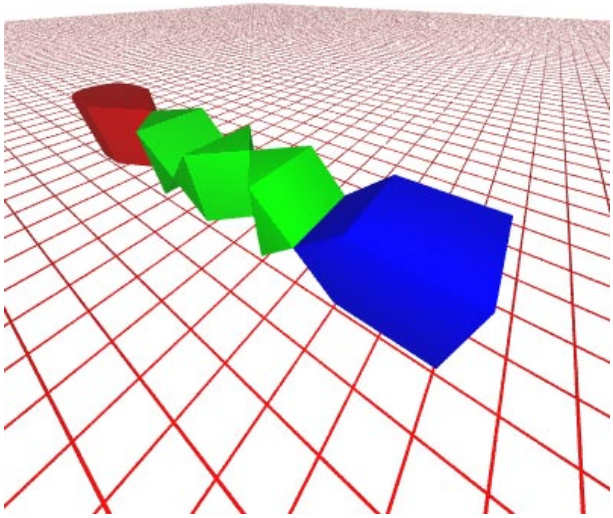


Abbildung 5.7: Ein raupenähnlicher (längerer) Roboter

Länge	13,0 m	Breite	2,0 m
Höhe	2,0 m		
Drehgelenke	4	davon motorisiert	4
Schubgelenke	0	davon motorisiert	0
Sensoren	0	Anzahl Beine	0

Tabelle 5.10: Maße der Raupe

5.5.2 Umgebung

Während die Umgebung weitgehend Standardparameter enthält (s. Tabelle 5.12), wurden bei den Parametern der Simulation eine längere Simulationszeit (3 Minuten) gewählt und etwas abweichende Werte für verschiedene Konstanten gewählt. Details bezüglich dieser Parameter enthält Tabelle 5.11.

Library	DynaMechs	Step size	0.01
Time to Sim.	3 Min.	Random Seed	0
Integrator	RK4	Joint fric. const.	0.35
Joint lim. spr. const.	100	Joint lim. damp. const.	5

Tabelle 5.11: Simulationsparameter der Raupe

Gravitation	$(0;-9.81;0)$
Start Position	$(0;1.2;0)$
Plan. spr. const.	5500
Norm. spr.const.	7500
Plan. damp. const.	50
Norm. damp. const.	50
Stat. fric. coeff.	1.5
Kin. fric. coeff.	1

Tabelle 5.12: Die Umgebung der Raupe

5.5.3 GP und Evolution

Die Einstellungen für die Genetische Programmierung lassen im Gegensatz zu denen der Grundlagenexperimenten (s. Kapitel 5.1 bis Kapitel 5.4) nur eine Minimallänge der Programme von 100 Programmzeilen zu. Die Maximallänge ist dagegen auf 300 Programmzeilen beschränkt. Die Mutation ist der dominante Operator, während in diesem Experiment keine Reproduktion erfolgt. Weitere GP-Parameter können der Tabelle 5.14 entnommen werden. Bei den Sprachparametern ist hervorzuheben, dass der SENSE Befehl mit sehr geringer Wahrscheinlichkeit und der MOVE Befehl mit sehr hoher Wahrscheinlichkeit gewählt wird. Andere Befehle wie NOP und JMP werden im Rahmen dieses Experiments nicht unterstützt. Details können der Tabelle 5.13 entnommen werden. [h] Es wurde über 160 Generationen evolviert.

5.5.4 Ergebnisse

Der Roboter bewegt sich tatsächlich durch eine kriechende, regelmäßige Bewegung mit durchschnittlich 0.46 Metern pro Sekunde fort. Dabei werden nur Gelen-

Com.	Dur.	Prob.	Com.	Dur.	Prob.	Com.	Dur.	Prob.
ADD	0.001	850	JMP	n/a	0	MOVE	0.1	1000
CMP	0.001	850	LOAD	0.001	850	MUL	0.001	850
COPY	0.001	850	MAX	0.001	850	NOP	n/a	0
DELAY	0.001	850	MIN	0.001	850	SENSE	0.001	50
DIV	0.001	850	MOD	0.001	850	SUB	0.001	850

Tabelle 5.13: Die Sprachparameter der Raupe

Crossover	30%	Min. Prg. length	100
Mutation	70%	Max. Prg. length	300
Reproduction	0%	Maximal age	50
Fitnessfunction	NiceWalkingFitnessFunction	Random seed	1
Tournmts. per Gen.	500	No. of Individuals	100

Tabelle 5.14: Die GP-Parameter der Raupe

ke benutzt, die den Roboter sich zusammenziehen lassen. Das anschließende Entfalten der Raupe bewirkt das Vorankommen. Das Gelenk zur Seitwärtsbewegung wird wie erwartet nicht benutzt, da der höchste Fitnesswert durch reine Geradeausbewegung zu erreichen ist. Allerdings wird vermutet, dass dieser Bewegungablauf schneller durchgeführt werden kann, dies aber ein stärkeres Zusammenziehen der Raupe erfordern würde, wobei der Torso die erlaubte Maximalhöhe der NiceWalkingFitnessFunction verletzen würde. Die Entwicklung der Fitness kann in Abb. 5.8 betrachtet werden, während Abb. 5.9 einen Ausschnitt der Bewegungsabläufe darstellt.

5.5.5 Fazit

Der von SIGEL erzeugte Bewegungsablauf erscheint intuitiv der effizienteste zu sein, der mit solch einer Roboterarchitektur möglich ist. Außerdem nutzt der Roboter seine Freiheitsgrade sinnvoll, die wohl fitnessverschlechternde Seitwärtsbewegung findet nicht statt. Wahrscheinlich wäre es aber fruchtbarer gewesen, eine andere Fitnessfunktion (nämlich die SimpleFitnessFunction) zu benutzen, um in diesem Fall eine starke Torsobewegung zu erlauben, doch diese langsamere Bewegung wirkt viel eleganter.

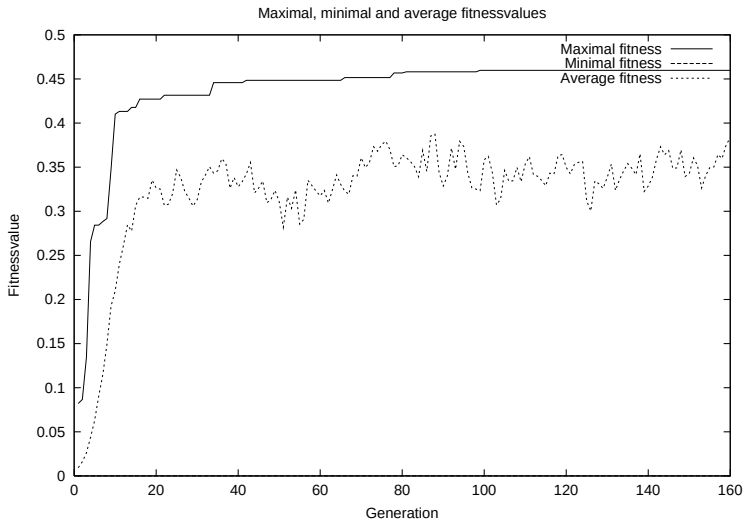


Abbildung 5.8: Die Fitnesskurve der Raupe mit *NiceWalkingFitness*

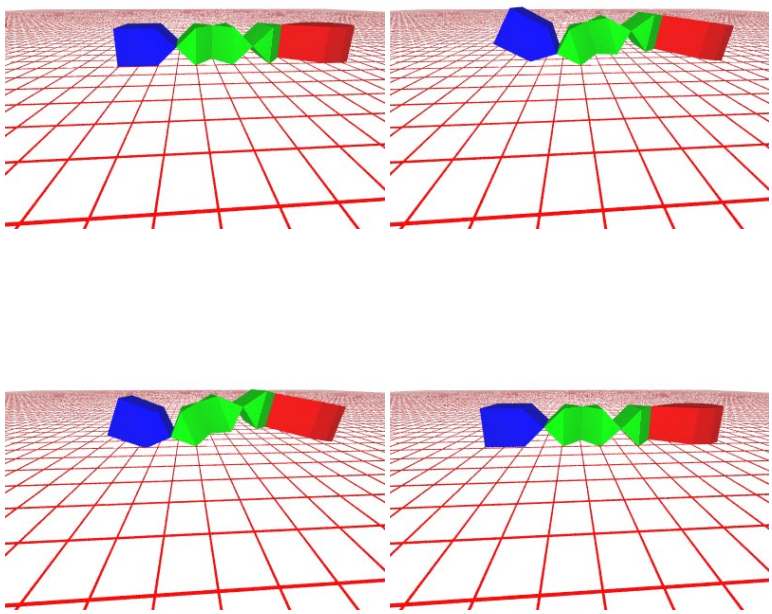


Abbildung 5.9: Der Bewegungsablauf der Raupe in Ausschnitten

5.6 Verkürzter raupenähnlicher Roboter

Dieses Experiment greift das oben beschrieben Experiment des Raupenroboters auf, bei dem nun ein Zwischenglied entfernt wurde.

5.6.1 Roboter

Dieser raupenähnliche Roboter entspricht weitgehend dem aus Kapitel 5.5. Lediglich die Länge ist aufgrund eines fehlenden Gliedes kürzer. Die Tabelle 5.15 und Abbildung 5.10 geben einen genaueren Überblick.

Länge	11,0 m	Breite	2,0 m
Höhe	2,0 m		
Drehgelenke	3	davon motorisiert	3
Schubgelenke	0	davon motorisiert	0
Sensoren	0	Anzahl Beine	0

Tabelle 5.15: Die Maße der verkürzten Raupe

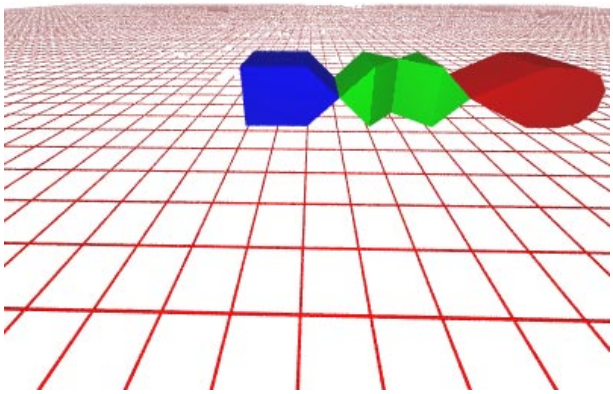


Abbildung 5.10: Ein raupenähnlicher (kürzerer) Roboter

Library	DynaMechs	Step size	0.01
Time to Sim.	3 Min.	Random Seed	0
Integrator	RK4	Joint fric. const.	0.35
Joint lim. spr. const.	100	Joint lim. damp. const.	5

Tabelle 5.16: Simulationsparameter der verkürzten Raupe

Gravitation	$(0;-9.81;0)$
Start Position	$(0;1.2;0)$
Plan. spr. const.	5500
Norm. spr.const.	7500
Plan. damp. const.	50
Norm. damp. const.	50
Stat. fric. coeff.	1.5
Kin. fric. coeff.	1

Tabelle 5.17: Die Umgebung der verkürzten Raupe

5.6.2 Umgebung

Während die Umgebung weitgehend Standardparameter enthält (s. Tabelle 5.17), wurden bei den Parametern der Simulation eine längere Simulationszeit (3 Minuten) gewählt und etwas abweichende Werte für verschiedene Konstanten gewählt. Details bezüglich dieser Parameter enthält Tabelle 5.16.

5.6.3 GP und Evolution

Diese Einstellungen entsprechen weitgehend denen aus dem vorherigen Kapitel 5.5. Details können den Tabellen 5.18 und 5.19 entnommen werden. Es wurde über 450 Generationen evolviert.

Com.	Dur.	Prob.	Com.	Dur.	Prob.	Com.	Dur.	Prob.
ADD	0.001	850	JMP	n/a	0	MOVE	0.1	1000
CMP	0.001	850	LOAD	0.001	850	MUL	0.001	850
COPY	0.001	850	MAX	0.001	850	NOP	n/a	0
DELAY	0.001	850	MIN	0.001	850	SENSE	0.001	50
DIV	0.001	850	MOD	0.001	850	SUB	0.001	850

Tabelle 5.18: Die Sprachparameter der verkürzten Raupe

Crossover	30%	Min. Prg. length	100
Mutation	70%	Max. Prg. length	300
Reproduction	0%	Maximal age	50
Fitnessfunction	NiceWalkingFitnessFunction	Random seed	1
Tournmts. per Gen.	500	No. of Individuals	100

Tabelle 5.19: Die GP-Parameter der verkürzten Raupe

5.6.4 Ergebnisse

Der Roboter bewegt sich auf die gleiche Art wie die ländere Version der Raupe mit 0.49 Metern pro Sekunde fort. Auch hier scheint die gewählte NiceWalkingFitnessFunction eine bessere Fitness zu verhindern, da der Torso eine Maximalhöhre nicht überschreiten darf. Der Einfluss auf die Fitnessentwicklung ist in Abb. 5.11 ersichtlich. Ein Bewegungsablauf in Ausschnitten ist in der Abb. 5.12 zu sehen.

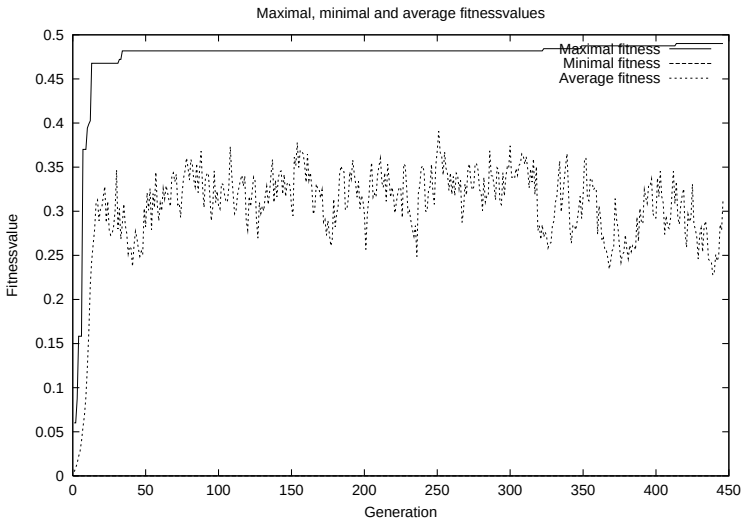


Abbildung 5.11: Die Fitnesskurve der verkürzten Raupe

5.6.5 Fazit

Hier ist das gleiche Fazit wie beim Experiment des längeren Raupenroboters zu ziehen: Es wurde erfolgreich eine Art der Fortbewegung gelernt, jedoch sind bessere Ergebnisse mit einer anderen Fitnessfunktion zu erwarten.

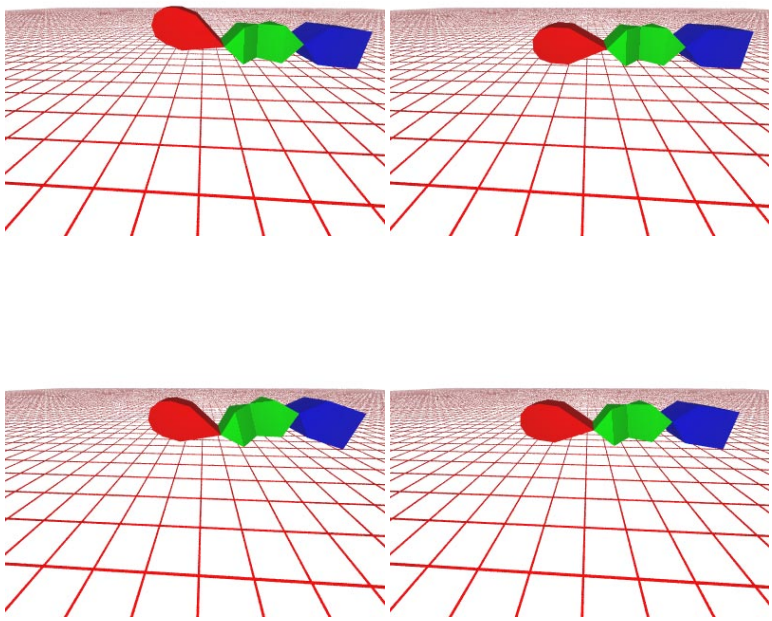


Abbildung 5.12: Der Bewegungsablauf in Ausschnitten der verkürzten Raupe mit NiceWalkingFitness

5.7 Dreibeiniger Roboter mit SimpleFitness

In diesem Experiment wurden Laufsteuerungsprogramme für einen dreibeinigen Roboter evolviert. Dazu wurde die SimpleFitnessFunction benutzt.

5.7.1 Roboter

Er besteht aus drei Beinen mit je drei Gliedern, die an einem Torso in der Mitte befestigt sind. Die Details können der Tabelle 5.20 und Abbildung 5.7.1 entnommen werden.

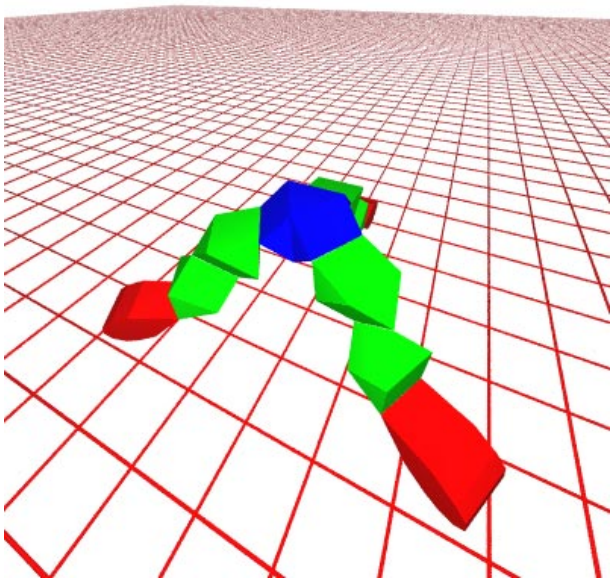


Abbildung 5.13: Ein dreibeiniger Roboter

Länge	6,5 m	Breite	6 m
Höhe	3 m		
Drehgelenke	9	davon motorisiert	9
Schubgelenke	0	davon motorisiert	0
Sensoren	9	Anzahl Beine	3

Tabelle 5.20: Maße des Dreibeiners

5.7.2 Umgebung

Während die Umgebung weitgehend Standardparameter enthält (s. Tabelle 5.22), wurden bei den Parametern der Simulation eine längere Simulationszeit (3 Minu-

Library	DynaMechs	Step size	0.01
Time to Sim.	3 Min.	Random Seed	0
Integrator	RK4	Joint fric. const.	0.35
Joint lim. spr. const.	100	Joint lim. damp. const.	5

Tabelle 5.21: Simulationsparameter des Dreibeiners

Gravitation	$(0;-9.81;0)$
Start Position	$(0;2;0)$
Plan. spr. const.	5500
Norm. spr.const.	7500
Plan. damp. const.	50
Norm. damp. const.	50
Stat. fric. coeff.	1.5
Kin. fric. coeff.	1

Tabelle 5.22: Die Umgebung des Dreibeiners

ten) gewählt und etwas abweichende Werte für verschiedene Konstanten gewählt. Details bezüglich dieser Parameter enthält Tabelle 5.21.

5.7.3 GP und Evolution

Die Einstellungen für die Genetische Programmierung lassen im Gegensatz zu denen der Grundlagenexperimenten (s. Kapitel 5.1 bis Kapitel 5.4) nur eine Minimallänge der Programme von 100 Programmzeilen zu. Die Maximallänge ist dagegen mit 300 Programmzeilen beschränkt. Die Mutation ist der dominante Operator, während in diesem Experiment keine Reproduktion erfolgt. Weitere GP-Parameter können der Tabelle 5.24 entnommen werden. Bei den Sprachparametern ist hervorzuheben, dass der SENSE-Befehl mit sehr geringer Wahrscheinlichkeit und der MOVE-Befehl mit sehr hoher Wahrscheinlichkeit gewählt wird. Details können der Tabelle 5.23 entnommen werden. Es wurde über 75 Generationen evolviert.

Com.	Dur.	Prob.	Com.	Dur.	Prob.	Com.	Dur.	Prob.
ADD	0.001	850	JMP	n/a	0	MOVE	0.1	1000
CMP	0.001	850	LOAD	0.001	850	MUL	0.001	850
COPY	0.001	850	MAX	0.001	850	NOP	n/a	0
DELAY	0.001	850	MIN	0.001	850	SENSE	0.001	50
DIV	0.001	850	MOD	0.001	850	SUB	0.001	850

Tabelle 5.23: Die Sprachparameter des Dreibeiners

Crossover	30%	Min. Prg. length	100
Mutation	70%	Max. Prg. length	300
Reproduction	0%	Maximal age	50
Fitnessfunction	SimpleFitnessFunction	Random seed	1
Tournmts. per Gen.	500	No. of Individuals	100

Tabelle 5.24: Die GP-Parameter des Dreibeiners mit SimpleFitness

5.7.4 Ergebnisse

Der Roboter bewegt sich mit durchschnittlich 0.83 Metern pro Sekunde fort. Allerdings handelt es sich um eine Kriechbewegung, bei der sich der Roboter mit einem Bein abstößt. Die Entwicklung der Fitness kann in Abb. 5.14 betrachtet werden, während Abb. 5.15 einen Ausschnitt der Bewegungsabläufe darstellt.

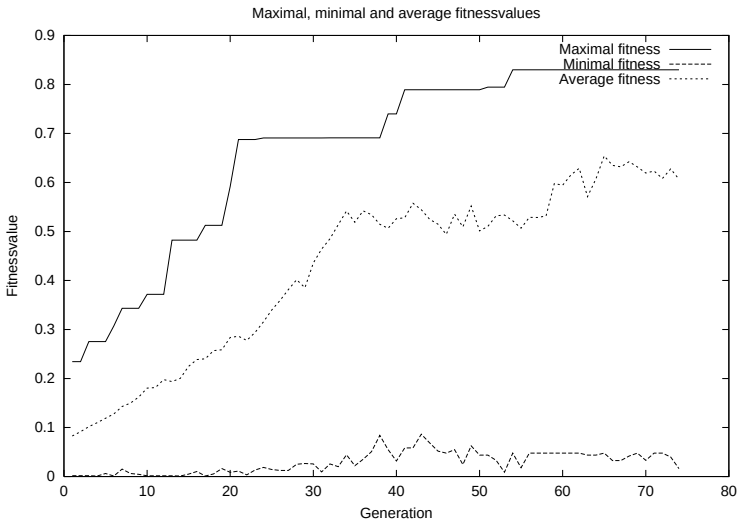


Abbildung 5.14: Die Fitnesskurve des Dreibeiners mit SimpleFitness

5.7.5 Fazit

Der Roboter hat erfolgreich gelernt, sich fortzubewegen. Dies geschieht allerdings in einer Art und Weise, die man eher als Kriechen statt Laufen bezeichnen würde. Dazu müsste der Torso auf einer gewissen Mindesthöhe bleiben und nicht über den Boden schleifen. Daher wurde das Experiment mit der NiceWalkingFitnessFunction wiederholt (siehe Abschnitt 5.8).

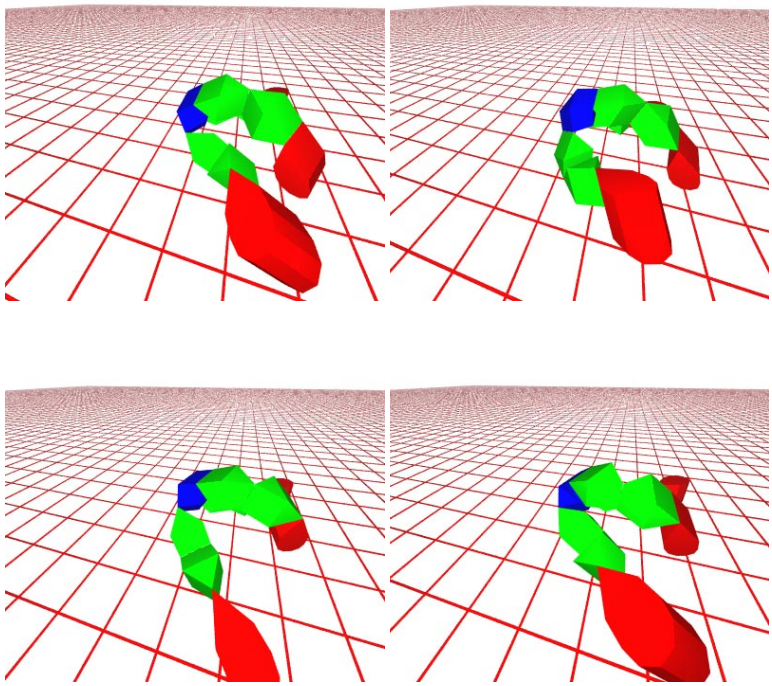


Abbildung 5.15: Der Bewegungsablauf des Dreibeiners mit SimpleFitness

5.8 Dreibeiner mit NiceWalkingFitness

In diesem Experiment wurde das in Abschnitt 5.7 beschriebene Experiment mit der NiceWalkingFitnessFunction wiederholt, um aus der kriechenden Bewegung eine gehende zu erzeugen.

5.8.1 Roboter, Umgebung, GP und Evolution

Es wird der in Abschnitt 5.7 beschriebene Roboter benutzt. Auch die Umgebung entspricht der aus dem o.g. Kapitel. Die GP Einstellungen (s. Tab. 5.25) ändern sich nur in der gewählten Fitnessfunktion. Es wurde über 260 Generationen evolviert.

5.8.2 Ergebnisse

Der Roboter bewegt sich mit durchschnittlich 0.52 Metern pro Sekunde fort. Auf Kosten eines geringeren Fitnesswertes wurde jedoch erreicht, dass der Bewegungsablauf nun als Laufen zu bezeichnen ist, da der Torso auf einer gewissen Minimalhöhe gehalten wird und der Roboter nicht „hinfällt“ (s. auch Abb. 5.16 und 5.17).

Crossover	30%	Min. Prg. length	100
Mutation	70%	Max. Prg. length	300
Reproduction	0%	Maximal age	50
Fitnessfunction	NiceWalkingFitnessFunction	Random seed	1
Tournmts. per Gen.	500	No. of Individuals	100

Tabelle 5.25: Die GP-Parameter des Dreibeiners mit NiceWalkingFitness

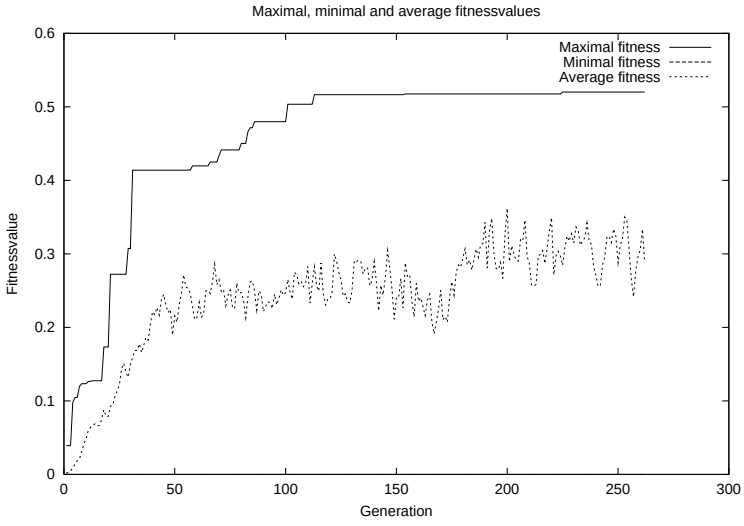


Abbildung 5.16: Die Fitnesskurve des Dreibeiners mit NiceWalkingFitness

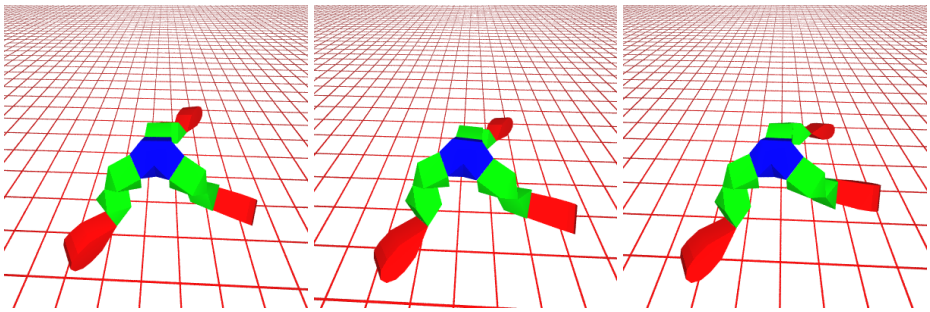


Abbildung 5.17: Der Bewegungsablauf des Dreibeiners mit NiceWalkingFitness

5.8.3 Fazit

Der Roboter hat erfolgreich gelernt, sich fortzubewegen. Durch die strengere NiceWalkingFitnessFunction wurde der Suchraum der Evolution eingeschränkt. Vermutet man, dass mit dieser Einschränkung der Minimalhöhe des Torsos kein besseres Laufen dieses Roboters möglich ist, hat dieses Experiment gezeigt, dass intuitive Vorstellung des Menschen vom „Laufen“ für diesen Roboter nicht das optimale Ergebnis geliefert hat, da der kriechende Bewegungsablauf des Experiments in Abschnitt 5.7 einen besseren Fitnesswert erzielte, trotzdem wäre im realen Fall die Bewegung aus diesem Experiment wesentlich materialschonender.

5.9 Evolution eines Sechsbeiners

In diesem Experiment wurden Laufsteuerungsprogramme für einen sechsbeinigen Roboter unter Benutzung der NiceWalkingFitnessFunction evolviert. Sinn dieses Experiments war der Versuch, die Erfolge mit Dreibeinern auf Sechsbeiner auszuweiten.

5.9.1 Roboter

Der Roboter besteht aus sechs Beinen, die aus je zwei Gliedern bestehen und kreisförmig um einen Torso angeordnet sind, gleicht also einem Insekt (vgl. Tab. 5.26 und Abb. 5.18).

Länge	3.8 m	Breite	3.8 m
Höhe	2 m		
Drehgelenke	12	davon motorisiert	12
Schubgelenke	0	davon motorisiert	0
Sensoren	12	Anzahl Beine	6

Tabelle 5.26: Maße des Insekts

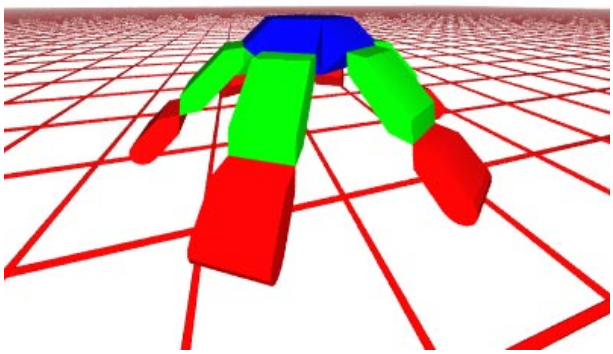


Abbildung 5.18: Ein insektenähnlicher Sechsbeiner

Library	DynaMechs	Step size	0.002
Time to Sim.	3 Min.	Random Seed	0
Integrator	RK4	Joint fric. const.	0.35
Joint lim. spr. const.	500	Joint lim. damp. const.	1

Tabelle 5.27: Simulationsparameter des Insekts

Gravitation	$(0;-9.81;0)$
Start Position	$(0;1;0)$
Plan. spr. const.	5500
Norm. spr.const.	7500
Plan. damp. const.	50
Norm. damp. const.	50
Stat. fric. coeff.	2
Kin. fric. coeff.	1.8

Tabelle 5.28: Die Umgebung des Insekts

5.9.2 Umgebung

Die Einstellungen von Simulation und Umgebung entsprechen den Standardeinstellungen, die Konstanten wurde wie üblich an das Modell angepasst (vgl. Tab. 5.27 und Tab. 5.28). Lediglich die Simulationszeit wurde wieder auf 3 Minuten erhöht. Bisher wurde nur über 17 Generationen evolviert.

5.9.3 GP und Evolution

Als Sprache wurden die üblichen Parameter genutzt (vgl. Tab. 5.29) und das Insekt unter Nutzung einer Crossoverrate von 30% und einer Mutationsrate von 70% evolviert (vgl. Tab. 5.30). Während JMP und NOP verboten wurden, ist außerdem die Wahrscheinlichkeit von MOVE erhöht, dafür die von SENSE verringert worden.

Com.	Dur.	Prob.	Com.	Dur.	Prob.	Com.	Dur.	Prob.
ADD	0.001	850	JMP	n/a	0	MOVE	0.2	1000
CMP	0.001	850	LOAD	0.001	850	MUL	0.001	850
COPY	0.001	850	MAX	0.001	850	NOP	n/a	0
DELAY	0.001	850	MIN	0.001	850	SENSE	0.001	50
DIV	0.001	850	MOD	0.001	850	SUB	0.001	850

Tabelle 5.29: Die Sprachparameter des Insekts

Crossover	30%	Min. Prg. length	100
Mutation	70%	Max. Prg. length	300
Reproduction	0%	Maximal age	50
Fitnessfunction	NiceWalkingFitnessFunction	Random seed	1
Tournmts. per Gen.	500	No. of Individuals	100

Tabelle 5.30: Die GP-Parameter des Insekts

5.9.4 Ergebnisse

Der Roboter bewegt sich mit durchschnittlich 0.11 Metern pro Sekunde fort. Allerdings ist hier nach weiteren Generationen Verbesserung zu erwarten (vgl. Abb. 5.19). Außerdem neigt der Roboter momentan dazu, über den Boden zu “rutschen”. Eine Anpassung der Umgebungsvariablen könnte hier zu besseren Ergebnissen führen (vgl. auch Abb. 5.20).

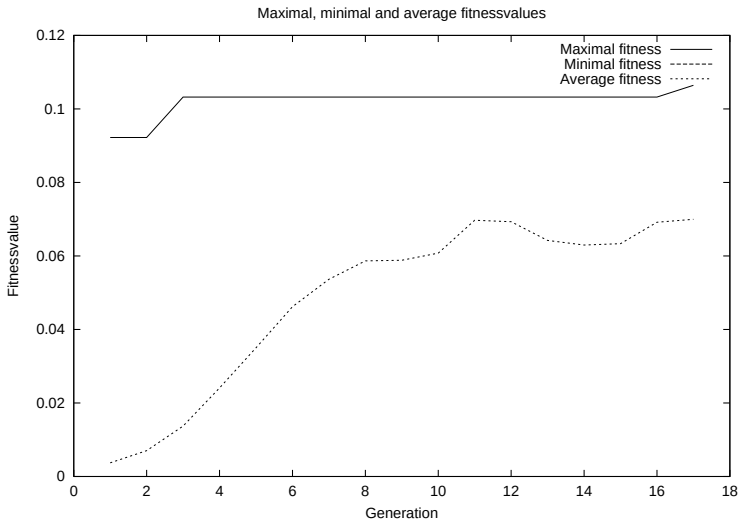


Abbildung 5.19: Die Fitnesskurve des Insekts

[h]

5.9.5 Fazit

Der Roboter hat erfolgreich gelernt, sich fortzubewegen. Allerdings müssen weitere Experimente mit größerer Bodenreibung durchgeführt werden. Außerdem scheint das Einfügen eines Schultergelenks sinnvoll, um die Beine auch horizontal bewegen zu können. Dieser Ansatz wurde bei der Konstruktion des in Abschnitt 5.10 beschriebenen Roboters verfolgt.

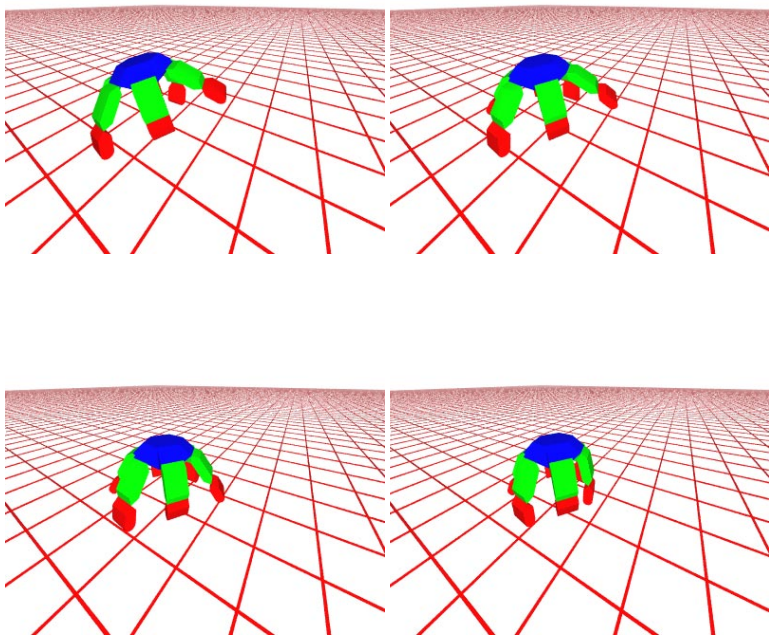


Abbildung 5.20: Der Bewegungsablauf des Insekts in Ausschnitten

5.10 Achsensymmetrischer Sechsbeiner mit NiceWalking

In diesem Experiment wurden Laufsteuerungsprogramme für einen sechsbeinigen Roboter evolviert. Dazu wurde die NiceWalkingFitnessFunction benutzt. Der Roboter unterscheidet sich zu dem im vorangegangenen Abschnitt vor allem durch die achsenparallele Konstruktion sowie die Einführung eines Schultergelenkes.

5.10.1 Roboter

Im Gegensatz zum unter 5.9 beschriebenen Roboter besitzt dieser Schultergelenke, die eine horizontale Rotation der Beine erlauben. Außerdem sind die Beine nicht kreisförmig um den Torso angeordnet, sondern achsensymmetrisch (vgl. Tab. 5.31 und Abb. 5.21).

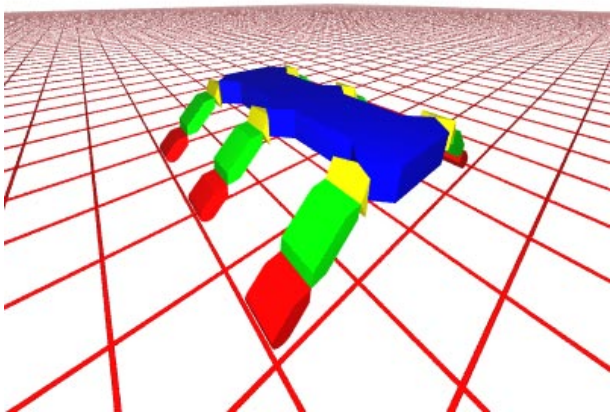


Abbildung 5.21: Ein weiterer sechsbeiniger Roboter, der Walker

Länge Höhe	5 m 2 m	Breite	4 m
Drehgelenke Schubgelenke Sensoren	18 0 18	davon motorisiert davon motorisiert Anzahl Beine	18 0 6

Tabelle 5.31: Maße des Walkers

Library	DynaMechs	Step size	0.002
Time to Sim.	1 Min.	Random Seed	0
Integrator	RK4	Joint fric. const.	0.35
Joint lim. spr. const.	500	Joint lim. damp. const.	1

Tabelle 5.32: Die Simulationsparameter des Walkers

Gravitation	$(0;-9.81;0)$
Start Position	$(0;1;0)$
Plan. spr. const.	5500
Norm. spr.const.	7500
Plan. damp. const.	1
Norm. damp. const.	1
Stat. fric. coeff.	3
Kin. fric. coeff.	2.9

Tabelle 5.33: Die Umgebung des Walkers

5.10.2 Umgebung

Wie üblich entspricht die Umgebung den Standardeinstellungen mit angepassten Konstanten und einer Simulationszeit von diesmal einer Minute (vgl. Tab. 5.32 und 5.33).

5.10.3 GP und Evolution

Hier wurden noch einmal die GP-Parameter des vorhergehenden Experiments benutzt (vgl. Tab. 5.29 und 5.30). Es wurde über 180 Generationen evolviert.

5.10.4 Ergebnisse

Der Roboter bewegt sich mit durchschnittlich 0.26 Metern pro Sekunde fort (vgl. Abb. 5.22). Die Bewegung ist noch sehr ungleichmäßig und langsam (siehe auch Abb. 5.23), kann aber wahrscheinlich durch weiter Evolutionen noch erheblich verbessert werden.

5.10.5 Fazit

Der Roboter hat erfolgreich gelernt, sich fortzubewegen. Auch hier ist eine Verbesserung durch eine längere Evolution zu erwarten. Es ist aber dennoch gezeigt, dass das System SIGEL für die Evolution von Steuerprogrammen für komplexe Mehrbeiner durchaus nutzbar ist.

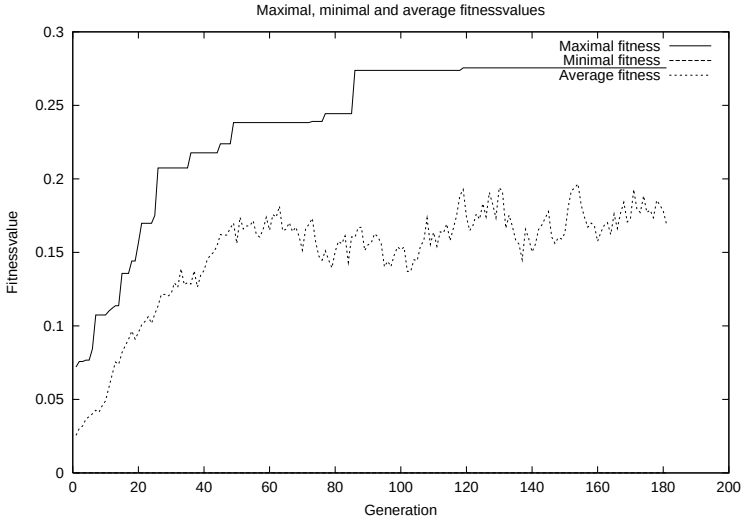


Abbildung 5.22: Die Fitnesskurve des Walkers

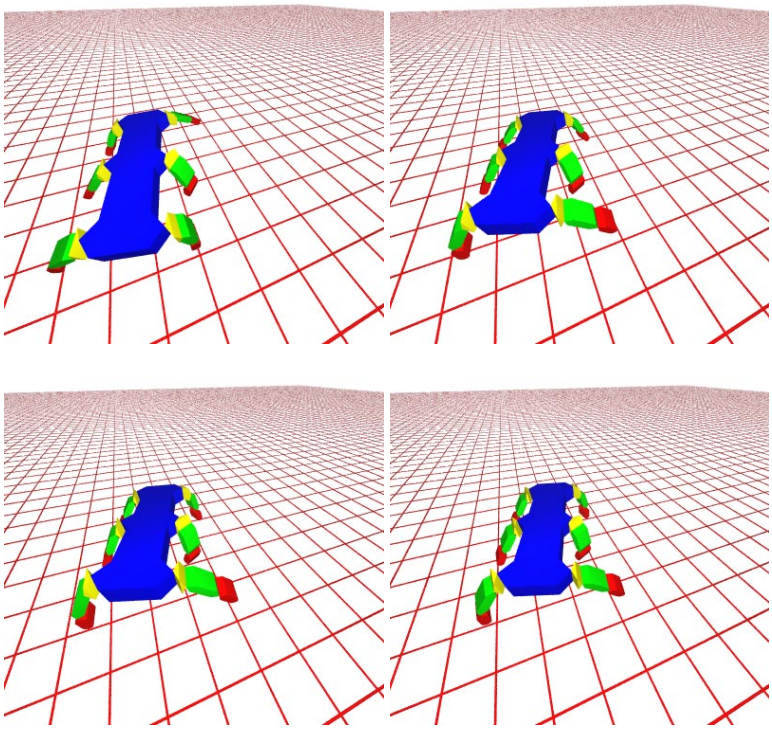


Abbildung 5.23: Der Bewegungsablauf des Walkers

5.11 Zusammenfassung

Die grundlegenden Experimente in den Kapiteln 5.1 bis 5.4 haben gezeigt, dass das implementierte GP-System ein vergleichbares Verhalten zu bekannten GP-Systemen aufweist. Das System war für ein sehr einfaches Modell in der Lage, eine Bewegung zu evolvieren. Verschiedene Variationen von genetischen Parametern und der Import von kontextfremden Individuen haben einen Teil der Funktionsfähigkeit dieses Systems demonstriert.

Die Ergebnisse der fortgeschrittenen Untersuchungen unterstreichen nochmals den Leistungsumfang dieses Systems. Als das wohl wichtigste Ergebnis lässt sich festhalten, dass unser System auch für komplexe Robotermodelle in der Lage ist, Steuerprogramme für Bewegungsabläufe zu evolvieren.

Die evolvierten Bewegungen entsprechen in den meisten Fällen natürlich (leider) noch nicht den natürlichen Vorbildern, was in der meist fehlenden *Eleganz* der Bewegung zum Ausdruck kommt, jedoch haben unsere Ergebnisse vermutlich gezeigt, dass dieses Aufgabe nicht völlig unlösbar ist. Leider sind unsere zeitlichen Ressourcen beschränkt, so dass wir keine weiteren komplexeren Untersuchungen bis zu diesem Zeitpunkt vornehmen konnten. Eine unserer Erfahrungen ist nämlich, dass dieses Thema ein riesiges Potential an Möglichkeiten bietet, was alleine durch die scheinbar unbegrenzte Vielfalt von denkbaren *intelligenten* Fitnessfunktionen reflektiert wird. Es ist schon faszinierend zu sehen, welch einen grossen Einfluss auf die Evolution selbst kleinste Änderungen (von Parametern) im System haben können.

Kapitel 6

Der Umgang mit dem System SIGEL

Dieses Kapitel ist als Benutzungshandbuch gedacht. Wir stellen hier die wichtigen Elemente des Programmes dar und zeigen in einfachen Schritten wie man mit SIGEL eine Evolution starten kann. Dazu wird in Abschnitt 6.1 die Installation beschrieben und im Abschnitt 6.2 allgemein in das System eingeführt. Anhand von Unterkapitel 6.3 wird die Modellierung mit unserer Roboterbeschreibungssprache erklärt. Um ein solches Modell zu evolvieren, braucht man die richtigen Simulationsparameter (Abschnitt 6.4) und GP-Parameter (Abschnitt 6.5). Die einzelnen Individuen können im Sinne einer interaktiven Evolution auch im- und exportiert sowie bearbeitet werden, wie im Abschnitt 6.6 erläutert.

6.1 Inbetriebnahme

In diesem Abschnitt wird beschrieben, welche Schritte durchgeführt werden müssen, um die kompilierte, statisch gelinkte Version des Programms SIGEL in Betrieb zu nehmen. Die Vorgehensweise sollte für SOLARIS und LINUX fast identisch sein.

Die SIGEL-Master Applikation sollte dann durch Ausführen der Datei `sigel` möglich sein. Möchte man ein Experiment ohne grafische Oberfläche evolvieren, geschieht dies über den Parameter `-evolve <experimentname>`.

6.1.1 Systemvorraussetzungen

Es folgt eine Auflistung von dynamischen Bibliotheken, die zur Laufzeit in den Systemverzeichnisse `/usr/lib` oder `/usr/openwin/lib` (bzw. `/usr/X11R6/lib` im LINUX-Fall) zu finden sein sollten (dem SOLARIS-Fall entnommen, für LINUX kann dies leicht variieren):
`libGLU.so.1`, `libGL.so.1`, `libXmu.so.4`, `libXext.so.0`, `libX11.so.4`, `libresolv.so.2`, `libsocket.so.1`, `libnsl.so.1`, `libSM.so.6`, `libICE.so.6`,

libdl.so.1, libc.so.1, libm.so.1, libintl.so.1, libdga.so.1,
libXt.so.4, libmp.so.2
und /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1.

6.1.2 Konfiguration der Umgebung

Es folgt eine Tabelle mit Namen von Umgebungsvariablen und ihren entsprechenden Werten. Diese müssen gesetzt werden, um SIGELbenutzen zu können.

SIGEL_ROOT	muss den Pfad der SIGEL-Installation enthalten.
PVM_ROOT	$SIGEL_ROOT/supportingLibs/pvm3
PATH	$PVM_ROOT/bin/SUN4SOL2:$PVM_ROOT/lib:$PATH
PVM_DPATH	$PVM_ROOT/lib/pvmd

Im Falle der Variable PATH bedeutet der Anhang ...:$PATH, dass die neuen, SIGEL-spezifischen Pfade an den Anfang der bisherigen Pfadvariable angefügt werden.

6.1.3 Das Skript sigelLauncher

Lässt man eine Evolution automatisch starten (beispielweise über den CRONTAB Mechanismus), so wird vorher evtl. nicht das shell-spezifische Startskript ausgeführt (bei der CSH ist dies die Datei .cshrc). Das Skript sigelLauncher setzt deswegen vorher alle entscheidenen Umgebungsvariablen. Dazu gehört die bereits oben aufgefuhrte Variable SIGEL_ROOT. Nach einer entsprechenden Anpassung des Skript auf das lokale Installationsverzeichnis kann es benutzt werden, um eine Evolution ohne bereits konfigurierte Umgebung zu starten. Ein Beispielaufruf könnte ./sigelLauncher -evolve insect.exp sein.

6.1.4 Benötigte SIGEL-Ressourcen

Während die Binärdistribution von SIGEL nur die wirklich benötigten Komponenten enthalten sollte, liegen der Quellcodedistribution Komponenten bei, die der Benutzer nach erfolgreichem Kompilieren löschen kann. Hier werden die Dateien bzw. Unterverzeichnisse aufgeführt, die zum Betreiben von SIGEL notwendig sind.

- Natürlich die Dateien sigel und sigel_slave.
- Die Datei flatTerrain.ter.
- Das Skript sigelLauncher.
- Das Unterverzeichnis pixmaps.
- Das Unterverzeichnis supportingLibs/pvm3.

All diese Ressourcen müssen sich im in der Umgebungsvariable SIGEL_ROOT genannten Verzeichnis befinden.

6.2 Ein erster Überblick über SIGEL

Nachdem in Kapitel 6.1 beschrieben wurde, wie SIGEL installiert und eingerichtet wird, gibt dieses Kapitel nun einen kurzen Überblick über die beiden Programme die zum System SIGEL gehören, den Master und den Slave. Hier wird der grundsätzliche Aufbau der GUI erklärkt, Details zu den Parametern sind in den folgenden Unterkapiteln.

6.2.1 Die Arbeitsumgebung des SIGEL-Masters

Der SIGEL-Master ist das Programm, in welchem die Einstellungen, die für eine Evolution benötigt werden, vorgenommen werden können. Abbildung 6.1 zeigt die grafische Benutzeroberfläche des Masters nach dem Start des Programms. Wie

Abbildung 6.1: Der Startbildschirm des SIGEL-Masters.

aus der Abbildung ersichtlich ist, besitzt das Programm alle Eigenschaften einer modernen Applikation. Ein Menü, über welches dem Programm Befehle gegeben werden können, eine Toolbar, über die die wichtisten Befehle schnell erreichbar sind, sowie eine Status-Leiste, welche Zusatzinformationen bietet. Den Großteil des Fensters nimmt jedoch die Arbeitsfläche ein, auf der die verschiedenen Experimente verwaltet und manipuliert werden können.

6.2.1.1 Das Menü

Das Hauptmenü (Abbildung 6.2) besitzt fünf Untermenüs, auf jedes wird nun etwas näher eingegangen.

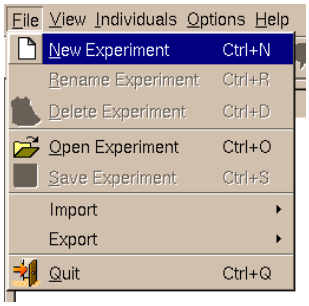


Abbildung 6.2: Das Hauptmenü des SIGEL-Masters. Das Untermenu „File“ ist geöffnet.

File: In diesem Menü können Experimente neu erzeugt, umbenannt und gelöscht werden. Außerdem besteht hier die Möglichkeit, Experimente zu laden und zu speichern. Wenn sich mindestens ein Experiment in der Liste der Experimente befindet, können Parameter über dieses Menü importiert bzw. exportiert werden. Ein Menüpunkt zum Verlassen des Programms findet sich ebenso hier.

View: Über dieses Menü kann selektiert werden, welche Parameter des aktuell angewählten Experiments betrachtet werden sollen.

Individuals: Hier können Individuen hinzugefügt, gelöscht, zurückgesetzt und im Simulator betrachtet werden.

Options: Unter diesem Punkt kann die Toolbar konfiguriert, sowie die Schriftart bzw. Schriftgröße der Applikation geändert werden. Für die Toolbar besteht die Option, große Icons zu verwenden sowie Textlabel anzuzeigen.

Help: Hier besteht die Möglichkeit, sich die Info-Box von SIGEL anzeigen zu lassen.

6.2.1.2 Die Toolbar

In der Toolbar befinden sich einige der wichtigsten und am häufigsten benötigten Befehle des Masters. Jeder Befehl der Toolbar findet sich auch in einem der Menüs wieder. Abbildung 6.3 zeigt die Toolbar. Ganz links finden sich die wichtig-

Abbildung 6.3: Die Toolbar des SIGEL-Masters.

sten Einträge des File-Untermenüs, also die Funktionen Experiment neu erzeugen,

löschen, laden und speichern, sowie das Verlassen des Programms. Direkt daneben befindet sich das View-Menü in seiner Toolbar-Form und ganz rechts finden sich drei Einträge aus dem Individuals-Menü (Individuen hinzufügen, löschen und visualisieren).

6.2.1.3 Die Arbeitsfläche

Direkt unter dem Hauptmenü und der Toolbar befindet sich die Arbeitsfläche. Sie ist in zwei Teile geteilt. Auf der linken Seite werden alle geöffneten Experimente angezeigt, während auf der rechten Seite jeweils die entsprechenden zu einem Experiment gehörigen Daten angezeigt werden. Nach dem Start ist die Liste auf der linken Seite noch leer. Es muss erst ein Experiment geladen oder neu erzeugt werden. Beides geht entweder über das Menü, die Toolbar oder aber ein Kontextmenü, welches erscheint, wenn man mit der rechten Maustaste in die Liste der Experimente auf der linken Seite der Arbeitsfläche klickt.

Nach der Erzeugung eines neuen Experiments wird dieses nun in der Liste der Experimente angezeigt. Der Bildschirm sieht dann in etwa wie in Abbildung 6.4 aus. Neue Experimente bekommen grundsätzlich den Namen `Experiment-` gefolgt von einer Zahl. Im Beispiel hat das neu erzeugte Experiment den Namen

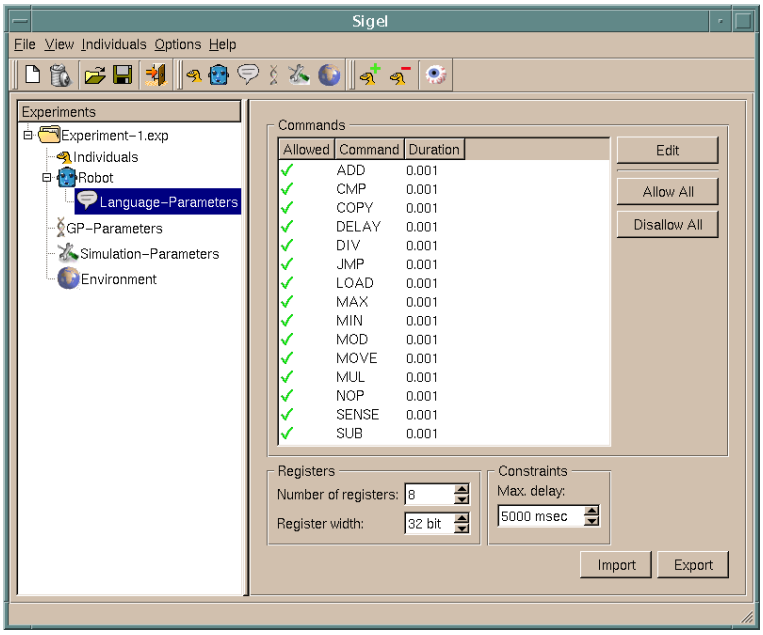


Abbildung 6.4: Der Master mit einem neu erzeugten Experiment.

`Experiment-1` bekommen und besitzt wie jedes Experiment noch sechs Unterpunkte, die alle mit der Maus anwählbar sind. Wird ein Unterpunkt angeklickt, so werden die entsprechenden Daten auf der rechten Seite der Arbeitsfläche angezeigt.

In Abbildung 6.4 werden zum Beispiel gerade die Language-Parameter angezeigt. Auf jeden dieser Punkte wird nun im Folgenden kurz eingegangen.

Individuals: Unter diesem Punkt können alle sich im Pool befindlichen Individuen betrachtet werden. Ebenso wird hier deren Anzahl angezeigt. Wird ein Individuum selektiert, so werden seine Informationen im unteren Fenster angezeigt. Ein Doppelklick liefert dieselben Informationen in einem Extrafenster, was recht praktisch ist, wenn man zwei oder mehrere Individuen miteinander vergleichen möchte.

Robot: Hier können Roboter geladen und gespeichert werden. *Import* lädt die Roboter im `.rrb`-Format, ein Format bei dem die Roboter noch nicht durch das Programm bearbeitet wurden. Zum Speichern benutzt SIGEL ein anderes Format, das `.crb`-Format, welches über *Load* wieder geladen werden kann. Die Modellierung eines Roboters erfolgt normalerweise im `.rrb`-Format und ist im Kapitel 6.3 näher beschrieben.

Language-Parameters: Die sprachlichen Restriktionen können an dieser Stelle geändert werden. Ein Befehl kann erlaubt oder verboten werden und die Dauer eines Befehl kann verändert werden. Ebenso wird hier die Registerbreite und die Anzahl der Register eingestellt.

GP-Parameters: Hier können alle für die Genetische Programmierung erforderlichen Parameter eingestellt werden. Details hierzu gibt es in Kapitel 6.5.

Simulation-Parameters: Unter diesem Punkt können die die Simulation betreffenden Einstellungen vorgenommen werden. Kapitel 6.4 gibt hier weiteren Aufschluss.

Environment: Der Gravitationsvektor und der Startpunkt des Roboters können hier eingestellt werden. Ebenso einige Dynamiksimulations-spezifische Einstellungen.

6.2.2 Die Nutzung des Masters

Nachdem in Kapitel 6.2.1 die Arbeitsumgebung des SIGEL-Masters beschrieben wurde, wird hier nun auf die praktische Nutzung des Masters eingegangen. Ziel dieses Kapitels ist es, aufzuzeigen, welche Maßnahmen getroffen werden müssen, um für ein modelliertes Robotermodell Robotersteuerungen zu evolvieren.

6.2.2.1 Evolution von Robotersteuerungsprogrammen

Es wird davon ausgegangen, dass wie in Kapitel 6.3 beschrieben, ein Robotermodell erzeugt wurde und nun vorliegt. Da die Steuerung komplett neu evolviert werden soll, muss zuerst ein neues Experiment erzeugt werden, zum Beispiel durch drücken der Tastenkombination CTRL+N. Das Experiment erscheint in der Liste

der Experimente und ist auch das momentan selektierte. Die wichtigsten Punkte, die nun auf jeden Fall (am Besten in dieser Reihenfolge) abgearbeitet werden müssen, sind die Folgenden:

1. Importieren bzw. Laden des Robotermodells.
2. Einstellung der Language-Parameter.
3. Einstellung der GP-Parameter.
4. Zufällige Erzeugung von mindestens vier Individuen.
5. Einstellung der Simulation-Parameter.
6. Einstellung der Umwelt-Parameter.

Egal ob evolviert oder visualisiert werden soll, in jedem Fall wird ein Robotermodell benötigt. Daher sollte das Laden des Robotermodells ganz an erster Stelle stehen. Robotermodelle können unter dem Punkt „Robot“ importiert bzw. geladen werden.

Anschließend sollten die Sprachparameter des Robotermodells festgelegt werden, da diese auch bei der Erzeugung von zufälligen Programmen schon berücksichtigt werden. Dies kann in den „Language-Parametern“ bewerkstelligt werden, welche ein Unterpunkt von „Robot“ sind.

Ein weiterer Punkt, der bei der Erzeugung zufälliger Programme berücksichtigt wird, sind die „GP-Parameter“. Hier kann die Minimal- bzw. Maximallänge von Programmen, sowie einige andere Dinge festgelegt werden. Unter anderem kann unter dem Reiter „PVM“ die virtuelle Maschine verwaltet werden, d.h. es kann eingestellt werden, welche Rechner an der parallelen Evolution teilnehmen sollen. Mehr dazu jedoch in Kapitel 6.5.

Nachdem diese drei Punkte abgearbeitet wurden, können nun endlich zufällige Programme erzeugt werden. Für eine Evolution werden mindestens vier Programme benötigt. Programme können jedoch nicht nur zufällig erzeugt werden, sie können auch importiert werden. Um ein Programm zu importieren, muss jedoch ein Individuum vorhanden sein, in welches es „hineingeladen“ werden kann. Damit nicht immer erst ein Individuum erzeugt werden muss, können auch ganze Individuen importiert bzw. exportiert werden.

Anschließend sollten noch einige Einstellungen in den „Simulation-Parametern“ (z.B. die verwendete Simulationsbibliothek) und dem „Environment“ (z.B. der Startpunkt des Roboters) gemacht werden. Ist dies gemacht, dürfte einer Evolution nichts mehr im Wege stehen.

Die Evolution kann auf unterschiedliche Art und Weise gestartet werden. Eine Möglichkeit ist ein Kontextmenü, welches erscheint, wenn mit der rechten Maustaste in der Liste der Experimente auf den Namen des Experiments geklickt wird. Eine andere Möglichkeit besteht darin, mit der linken Taste auf den Namen des Experimentes zu klicken. In diesem Fall kann die Evolution über die zwei Buttons

mit der Aufschrift „Start“ bzw. „Stop“ gestartet bzw. gestoppt werden. Diese finden sich dann im rechten Bereich der Arbeitsfläche.

6.2.2.2 Visualisierung der Ergebnisse

Die Evolution von Robotersteuerungsprogrammen würde nicht viel bringen, wenn die Früchte der Arbeit nicht auch betrachtet werden könnten. Zur Visualisierung der Ergebnisse benutzt SIGEL den SIGEL-Slave, der auch benutzt wird, um Fitnesswerte zu berechnen. In Kapitel 6.2.3 wird auf die Arbeitsumgebung des SIGEL-Slave eingegangen.

Um ein Individuum zu visualisieren, muss zuerst im Individuen-Bildschirm ein Individuum (oder mehrere durch Drücken der CTRL- bzw. SHIFT-Taste) angewählt werden. Anschließend gibt es wieder mehrere Wege diese(s) zu visualisieren. Entweder im Hauptmenü unter dem Punkt „Individuals“ mit „Visualize“ oder aber über die Toolbar durch Drücken auf das Augen-Icon. Dabei sollte allerdings größte Vorsicht geboten sein, dass nicht aus Versehen zu viele Individuen angewählt werden, da für jedes Individuum über PVM ein eigener Slave-Prozess gestartet wird.

6.2.3 Die Arbeitsumgebung des SIGEL-Slave

Der SIGEL-Slave wird auf zwei verschiedene Arten genutzt. Zum einen während der Evolution, um die Individuen zu bewerten, zum anderen um die Individuen zu visualisieren. Im ersteren Fall wird der Slave ohne eine grafische Benutzeroberfläche gestartet und liefert einen Fitnesswert zurück. Im zweiten Fall stellt der Slave eine grafische Benutzeroberfläche zur Verfügung, über die auf die Visualisierung Einfluss genommen werden kann. Abbildung 6.5 zeigt den SIGEL-Slave bei der Visualisierung eines Roboters. Die Arbeitsumgebung des Slaves besteht aus einer Toolbar und einem großen Hauptfenster, in welchem auch die Visualisierung läuft.

6.2.3.1 Die Toolbar des Slave

Die Toolbar des SIGEL-Slave ist in Abbildung 6.6 dargestellt. Wie der Abbildung zu entnehmen ist gibt es sechs Aktionen, welche über die Toolbar ausgeführt werden können. Diese werden nun von links nach rechts beschrieben:

Stop: Über diesen Button wird die Simulation gestoppt. Gleichzeitig wird sie in den Anfangszustand zurückgesetzt.

Start: Dieser Button startet die Evolution. Wird er gedrückt, ändert sich das Start-Symbol in ein Pause-Symbol, über welches die Simulation pausiert werden kann. Die Pause wird durch erneutes Drücken des Button aufgehoben.

Step: Wird dieser Button gedrückt, so wird genau ein Simulationsschritt ausgeführt. Die simulierte Zeit entspricht dabei der in den „Simulation-Parameter“ eingestellten Schrittweite (Stepsize) des Integrators.

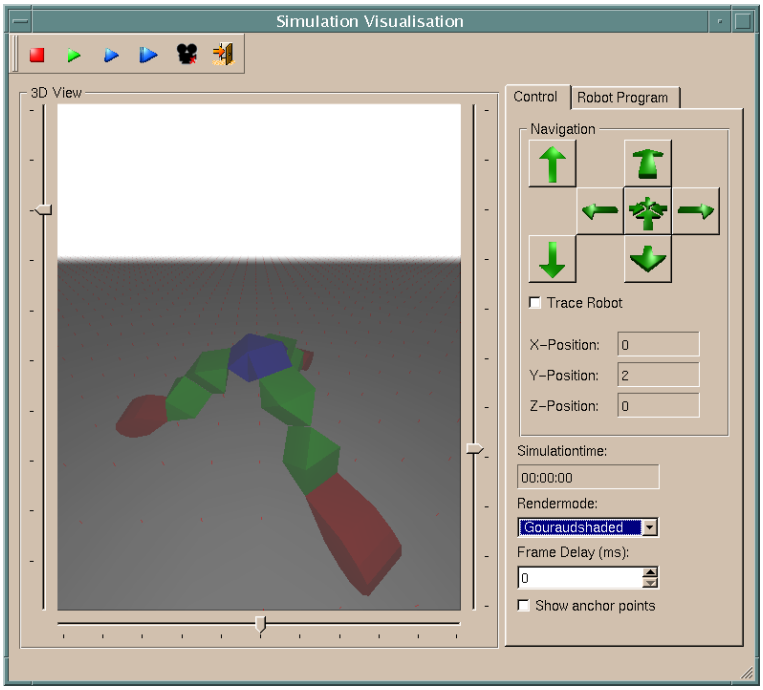


Abbildung 6.5: Der SIGEL-Slave bei der Visualisierung.

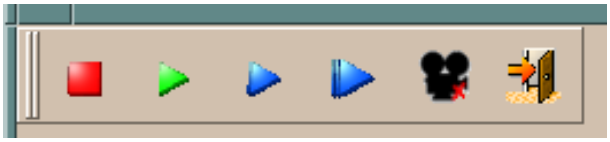


Abbildung 6.6: Die Toolbar des SIGEL-Slave.

Fast forward: Dieser Button veranlasst die Simulation fünf Sekunden zu simulieren, ohne dass es in der Visualisierung angezeigt wird.

Movie settings: Es ist auch möglich den Slave zu benutzen, um Filme (z.B. im MPEG-Format) zu erstellen. Die dafür nötigen Einstellungen können durch Drücken dieses Buttons vorgenommen werden. Dazu jedoch später mehr.

Quit: Beendet das Programm.

6.2.3.2 Das Hauptfenster des Slaves

Das Hauptfenster des Slaves ist genau wie beim Master in zwei Teile geteilt. Links wird die Simulation visualisiert, während rechts einige Einstellungen vorgenommen werden können bzw. der Code des aktuell ausgeführten Programms betrachtet

werden kann.

Über die drei Slider neben der Visualisierung kann die Darstellung gezoomt und rotiert werden. Dies geht jedoch nicht nur über die Slider, sondern auch bequem mit der Maus, wenn diese mit gedrückten Buttons in der Visualisierung bewegt wird. Wird der rechte Button gedrückt, so kann durch hoch und runter Bewegen der Maus gezoomt werden, während ein Bewegen der Maus mit gedrückter linker Taste in Rotationen um den Roboter resultiert.

Auf der rechten Seite findet sich ein Tab mit zwei Reitern, welche mit „Control“ und „Robot Program“ beschriftet sind. Unter „Robot Program“ kann das aktuell simulierte Programm betrachtet werden und „Control“ bietet die Möglichkeit, einige Einstellungen vorzunehmen. Dazu gehört zum Beispiel die Einstellung, ob der Roboter verfolgt werden soll während er sich bewegt oder aber die Art des Shading. Wird der Roboter nicht verfolgt, so kann über die Buttons mit den Pfeilen darauf die Kameraposition eingestellt werden. Das Rotieren der Ansicht funktioniert in diesem Modus auch noch. Weiterhin wird hier der aktuelle Sichtpunkt der Kamera (beim Verfolgen des Roboters die Position des Torsos) in Koordinaten sowie die simulierte Zeit angezeigt.

6.2.4 Die Nutzung des Slaves

Der Slave wird auf zwei verschiedene Arten genutzt. Einmal zur Fitnessberechnung und einmal zur Visualisierung. Bei der Visualisierung gibt es wiederum zwei verschiedene Möglichkeiten. Entweder man schaut sich die Bewegungen des Roboters direkt an oder aber man lässt Bilder der Simulation auf die Festplatte schreiben, um daraus z.B. einen MPEG-Film zu erstellen. Für letzteres müssen einige Einstellungen gemacht werden, was durch Drücken des Buttons mit der Kamera erreicht werden kann. Wird der Button gedrückt, erscheint der Dialog, welcher in Abbildung 6.7 zu sehen ist. Auf die verschiedenen Einstellmöglichkeiten wird nun kurz eingegangen.

Geometry: Hier kann die Größe und die Breite der gespeicherten Bilder festgelegt werden.

Frequency: Mit diesen Optionen wird festgelegt, jedes wievielte Bild gespeichert werden soll und wieviele Bilder höchstens gespeichert werden. Es ist zu beachten, dass die Frequenz mit der die Bilder gespeichert werden müssen, von der Schrittweite des Integrators abhängt.

File conventions: Unter diesem Punkt kann u.a. das Verzeichnis, in dem die Bilder gespeichert werden, angegeben werden. Ist das Verzeichnis nicht vorhanden, so wird es automatisch angelegt. Desweiteren kann der Datei-Präfix angegeben werden und ob führende Nullen benutzt werden sollen. Ein Beispiel: Das Präfix sei „sigel_pic“, die maximale Anzahl zu speichernder Bilder ist 1000 und führende Nullen werden benutzt. Dann werden die entstehenden Zieldateien mit `sigel_pic0000, sigel\_pic0001,...,sigel_pic0999`

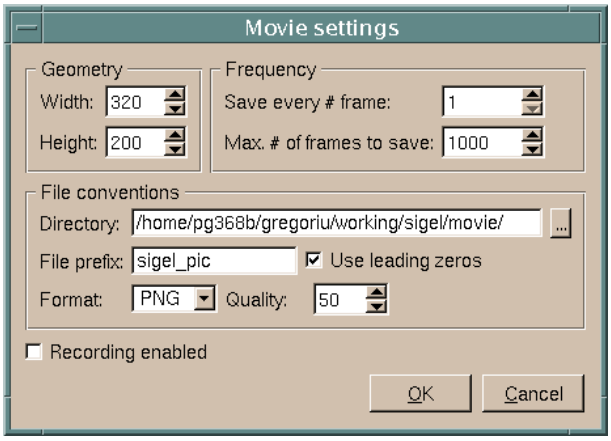


Abbildung 6.7: Der „Movie Settings“-Dialog des SIGEL-Slave.

bezeichnet. Zu guter Letzt kann hier noch das verwendete Dateiformat und die Kompressionsrate, sofern das Format komprimiert, angegeben werden.

Recording enabled: Über diese Checkbox kann die Aufzeichnung von Bildern ein- bzw. ausgestellt werden. Die momentane Einstellung wird auch durch den Kamera-Button repräsentiert, welcher entweder ein rotes „X“ (keine Aufzeichnung) oder einen grünen Haken (Aufzeichnung aktiviert) zeigt.

6.3 Die Konstruktion von Robotermodellen

Aufgabe dieses Abschnittes ist, Informationen zu vermitteln, die es dem Leser ermöglichen, Robotermodelle zur Verwendung mit SIGEL zu erstellen. Die Verwendung von CAD-Programmen kann Bestandteil des Erstellungsprozesses sein. Kenntnisse im Umgang mit einem solchen Werkzeug zu vermitteln, ginge über den Rahmen des Endberichts und insbesondere dieses Abschnitts hinaus. Die notwendigen Kenntnisse werden vorausgesetzt.

Wenn die zu modellierenden Roboterglieder einfacher Gestalt sind, kann auf die Verwendung eines CAD-Programms verzichtet werden. In dem Falle müssen Sie die Oberfläche selbst in einzelne Polygone zerlegen und die Eckpunkte in die Konfigurationsdatei eintragen. Diese Methode war anfangs nicht vorgesehen, ist sehr rudimentär und nur für sehr einfache geometrische Gebilde zu Testzwecken brauchbar. Nichtsdestotrotz bietet es sich an, für Übungs- und Verständniszwecke darauf zurückzugreifen.

6.3.1 Phasen der Modellerstellung

Grundsätzlich lässt sich die Erstellung eines Robotermodells in folgende Phasen aufteilen:

1. Der Roboter wird betrachtet und seine kinematische Struktur wird analysiert. Das umfasst folgende Fragen:
   - Aus welchen Gliedern besteht der Roboter?
   - Welche Glieder hängen auf welche Weise aneinander?
2. Den bisher rein qualitativen Daten werden nun präzise Daten hinzugefügt. Nur wenn lediglich eine grobe Annäherung erstellt werden soll, kann darauf verzichtet werden, den Roboter zu zerlegen.
   - Wie lang/breit/hoch sind die Glieder? Wo befinden sich Unregelmäßigkeiten in der Form?
   - An welchen Stellen sind die Glieder mit Gelenken verbunden? Welchen Typs sind die Gelenke?
3. Es werden Beschreibungen der Glieder erstellt und die Oberflächenbeschreibungen eingegeben.
4. Es werden Beschreibungen der Gelenke erstellt. Damit geht die Verbindung der Glieder einher.
5. Letztlich werden Sensoren und Motoren des Roboter identifiziert und dem Modell hinzugefügt.

Im Folgenden wird der gesamte, von der Projektgruppe berücksichtigte Umfang der Roboterbeschreibung wiedergegeben. Bitte beachten Sie die in Abschnitt 6.3.7 angegebenen Einschränkungen und Besonderheiten der Dynamiksimulationen.

6.3.2 Kinematische Analyse

In dieser Phase wird analysiert, aus wie vielen und welchen Gliedern der Roboter besteht und wie diese zusammenhängen.

6.3.2.1 Glieder

Im Sinne unserer Definition und Semantik unserer Beschreibungssprache ist ein Glied ein physisches Teil eines Roboters, von dem idealisierend angenommen wird, dass es komplett aus einem total steifen Material gefertigt ist. Dies bedeutet, dass sich ein aus Gummi gegossener Stoßfänger nicht modellieren lässt. Es ist in diesem Fall bestimmt sinnvoller, einen steifen Stoßfänger zu modellieren, der mit einem extrem hohen Elastizitätswert ausgestattet ist, als ihn einfach wegzulassen.

Glieder, die aus verschiedenen Materialien gefertigt sind, lassen sich modellieren, in dem man annimmt, dass verschiedenen Glieder vorliegen und sie durch Gelenke ohne jeden Freiheitsgrad miteinander verbunden sind, was sie praktisch aneinander klebt.

Eines der Glieder müssen Sie als Rumpfteil auszeichnen. Anhand dieses Gliedes wird die Position und Orientierung des Roboters gemessen - vor allem diese Daten gehen in die Berechnung der Fitness ein.

6.3.2.2 Gelenke

Neben dem oben angeführten Ungelenk, der festen Verbindung, existieren in unserer Modellierungssprache drei weitere Gelenktypen:

1. Das *Drehgelenk* hat einen Freiheitsgrad und ermöglicht das Rotieren der Glieder um eine gemeinsame Achse.

2. Das *Schubgelenk* hat ebenfalls einen Freiheitsgrad und ermöglicht das Verschieben der Glieder entlang einer gemeinsamen Achse.

3. Das *Zylindergelenk* hat zwei Freiheitsgrade und ermöglicht beide Bewegungsarten der vorgenannten Gelenke, wobei Dreh- und Schubachse identisch sind.

In Abbildung 6.8 sind die verschiedenen Gelenktypen, die mit SIGEL modelliert werden können, aufgeführt.

Stellen Sie zu jeder Gliedverbindung fest, welcher Gelenktyp am Besten passt. Wenn Sie ein Gelenk vorfinden, dass nicht mit den aufgeführten Gelenktypen modellierbar ist, können Sie versuchen, es durch zwei Gelenke und ein sehr kleines Glied nachzubilden. Ein Kugelgelenk ließe sich auf diese Weise durch zwei Drehgelenke simulieren.


Praxistipp

Zeichnen Sie die kinematische Struktur auf Papier und geben Sie allen Gliedern und Gelenken aussagekräftige Namen, wie z.B. *linkes Knie* oder *rechter Oberarm*. Das verhindert, dass sie hinterher `gelenk_1`, `glied_2` usw. heißen und die Konfigurationsdatei unleserlich und schwer zu warten ist. Gleiches gilt für Namen von Punkten (s.u.).

6.3.3 Geometrische und physikalische Analyse

Hierbei geht es darum, die genauen Maße der Glieder herauszufinden. Wenn Sie ein akkurates Modell haben möchten, werden Sie an dieser Stelle wohl nicht darum herum kommen, zum Schraubenzieher zu greifen und den Roboter in seine Einzelteile zu zerlegen, um sie genau anschauen und das Lineal überall anlegen zu können. Je robuster Ihr Roboter gegenüber extremen Bewegungen und Kollisionen mit sich selbst ist, und je mehr Sie das resultierende Programm auf dem Roboter durch weitergehende Evolution verfeinern wollen, desto eher können Sie auf die Demontage verzichten und mit einfachen geometrischen Figuren modellieren.

Die meisten physikalischen Attribute werden vom Programm selbst errechnet, wenn es sich um Daten handelt, die mit der Geometrie zusammenhängen, wie z.B.

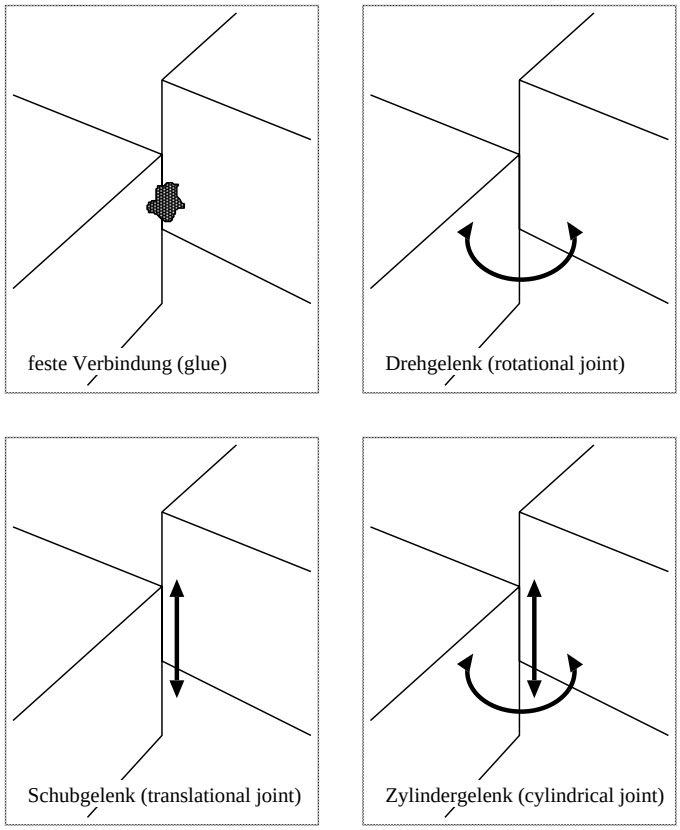


Abbildung 6.8: Die verschiedenen SIGEL-Gelenktypen im Überblick.

Schwerpunkt, Hauptachsen, Trägheitstensor. Angeben müssen Sie insbesondere die Materialeigenschaften, von denen die Dichte die wahrscheinlich wichtigste ist, beeinflusst sie doch alle oben genannten Werte, die für die Dynamiksimulation wichtig sind. Ferner werden Elastizität und Reibungskonstanten für die Kollisionen und eine Farbe für die grafische Darstellung verlangt.

6.3.4 Eingabe in die Konfigurationsdatei

Für die komplette Beschreibung eines Robotermodells benötigen Sie eine Konfigurationsdatei, die in der Regel die kinematische Beschreibung enthält, und eine Reihe von VRML-Dateien, aus denen die Oberflächenformen der Glieder geladen werden. Vorneweg sei noch gesagt, dass alle Zeilen, die in der ersten Spalte ein Hashzeichen (#) tragen, als Kommentare betrachtet und somit ignoriert werden.

6.3.4.1 Materialien

Alle Materialdaten werden in einer Materialbeschreibung wie der Folgenden zusammengefasst:

```
material steel {
  density 7800;
  elasticity 0.95;
  friction 0.7 on steel;
  colour red 0.5 green 0.5 blue 0.5;
}
```

Damit beschreiben wir das Material *Stahl*. Für die Dichte werden SI-Einheiten erwartet, von daher handelt es hier um $7800 \frac{\mathrm{kg}}{\mathrm{m}^3}$. Die Elastizität ist eine Hilfsgröße, die angibt, wieviel Energie bei einem Stoß erhalten bleiben soll. Mittels `friction` wird die Reibungskonstante festgelegt. Ein Körper aus Stahl reibt mit 0,7 mal der Normalkraft auf einem anderen Körper aus Stahl („`on steel`“). Die letzte Zeile des Körpers der Definition beschreibt die Farbe in der grafischen Darstellung.

6.3.4.2 Glieder und ihre relativen initialen Positionen

Haben Sie alle Materialien, die Sie am Roboter identifiziert haben, eingegeben, können Sie mit der Eingabe der Glieder beginnen. An dieser Stelle ist es bereits wichtig zu wissen, welche Gelenke ein Glied mit den nachfolgenden Gliedern verbinden. Jedes Gelenk wird durch drei Punkte an jedem beteiligten Glied beschrieben, nämlich einem Überdeckungspunkt, einer Achsenrichtung und einer Fixierrichtung oder Auslenkungsrichtung. Diesen Punkten müssen Sie in jedem Glied unterscheidbare Namen geben. Um zu verstehen, wie diese Punkte verwendet werden, genügt es, sich anzusehen, wie sie im Falle einer festen Verbindung zwischen zwei Gliedern wirken. Bitte betrachten Sie begleitend dazu Abbildung 6.9, in der der Vorgang illustriert ist.

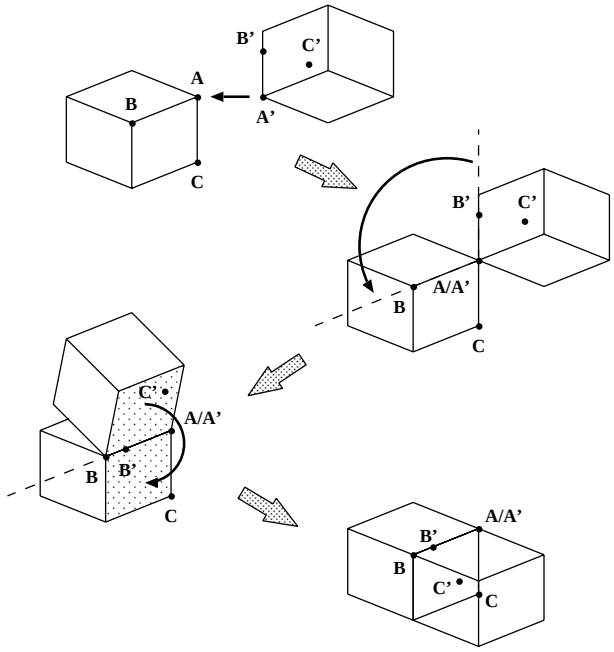


Abbildung 6.9: Beispiel für die Befestigung zweier Würfel aneinander.

Bitte behalten Sie ständig im Hinterkopf, dass Sie die Punkte relativ zum Koordinatensystem des Gliedes machen, wenn aber im Folgenden von Überdeckung oder Übereinanderlegen die Rede ist, sind immer Operationen und Zustände der Punkte in einem gemeinsamen Weltkoordinatensystem gemeint. Eine Überdeckung der Punkte führt also zu einer Positions- oder Orientierungsveränderung der Glieder, so dass die ins Weltkoordinatensystem zurückgerechneten Punkte korrespondieren.

Nehmen wir an, an einem Glied gäbe es die drei Punkte A,B,C und auf der Gegenseite die Punkte A',B',C'. Wenn die Glieder verbunden werden, werden zunächst die Punkte A und A' übereinandergelegt. Liegen diese übereinander, werden die Geraden AB und $A'B'$, die in Abbildung 6.9 als gestrichelte Linien dargestellt sind, zur Deckung gebracht. Der letzte Schritt besteht darin, die Ebenen, die durch die Gerade und den Punkt C, bzw C' aufgespannt werden, durch Rotation um die Gerade zu überdecken. Die Würfelseiten, die auf diesen Ebenen liegen, sind in der Abbildung schattiert dargestellt. Bitte beachten Sie, dass die Dreiecke ABC und $A'B'C'$ nicht kongruent sein müssen. Solange es sich um Dreiecke handelt, d.h. die Punkte liegen nicht auf jeweils einer Geraden, lässt sich die Verbindung der Glieder durchführen.

Während Sie bei festen Verbindungen beliebige Punkte zur Fixierung der Glieder benutzen können, ist dies bei Gelenken mit Freiheitsgraden nicht möglich, denn in den Punktangaben befindet sich dann auch die Information über die Dreh- bzw.

Gelenktyp	*Schlüsselwort*
feste Verbindung	`glue`
Drehgelenk	`joint rotational`
Schubgelenk	`joint translational`
Zylindergelenk	`joint cylindrical`

Tabelle 6.1: Die in SIGEL modellierbaren Gelenktypen und ihre Schlüsselwörter.

Schubachse und die Punkte, an denen die Gelenkauslenkung gemessen wird. Lesen Sie bitte dazu die Informationen im nächsten Abschnitt aufmerksam durch.

Weitere Daten, die zur Beschreibung eines Gliedes benötigt werden, sind das Material, aus dem es besteht, die Angabe der Datei, die die geometrische Form beschreibt, und die Angabe, ob es sich um das Rumpfglied handelt. Eine Beschreibung eines Rumpfteils könnte also folgendermaßen aussehen:

```
link koerper {
  torso;
  geometry "rumpf.wrl";
  material steel;
  point flap_vorne_offset = (0, 0, 0);
  point flap_vorne_achse = (1, 0, 0);
  point flap_vorne_zeiger = (0, 1, 0);
  [...]
}
```

Handelte es sich nicht um das Rumpfglied, würde die Zeile `torso;` entfallen.

6.3.4.3 Gelenke

Gelenke in einem Robotermodell sind Parametersätze, die beschreiben, wie sich zwei Glieder zueinander bewegen können. In Abbildung 6.8 sind die grundsätzlich verschiedenen Möglichkeiten der Bewegung zueinander aufgeführt. Tabelle 6.1 zeigt, mit welchen Schlüsselwörtern die Gelenkmodelle erzeugt werden. Wenn Sie zwischen dem Kopf und dem Rest eines Körpers einen nach vorne klappbaren Hals modellieren möchten, könnten Sie das auf die folgende Art und Weise tun:

```
joint rotational genick {
  between koerper (schulterrechts, schulterlinks, oben)
      and kopf (aufschulterrechts, aufschulterlinks, nachoben);
}
```

Hierbei muss es Punkte mit den Namen `schulterrechts`, `schulterlinks` und `oben` geben, die im Glied `koerper` definiert wurden, wie es weiter oben erläutert ist. Entsprechendes gilt für das Glied `kopf`. In einem Gelenk mit Freiheitsgraden sind die Punkte nicht nur zur Bestimmung der initialen, relativen Positionen da,

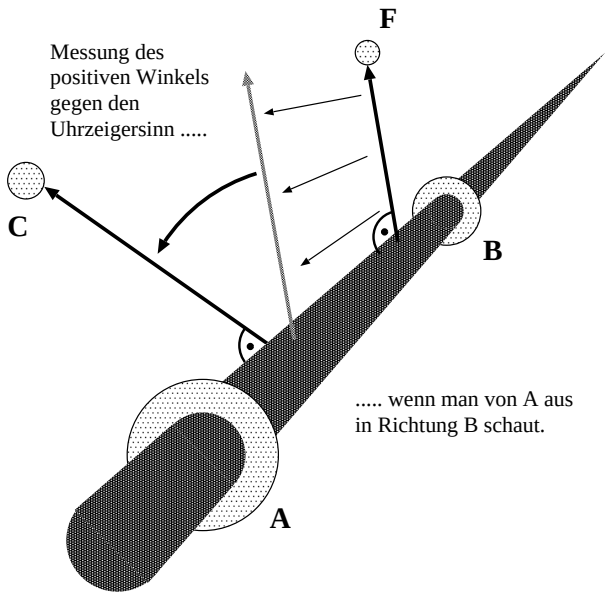


Abbildung 6.10: Stellung eines Gelenks mit Rotationsfreiheitsgrad.

sondern auch, um die Dreh-/Schubachse festzulegen. Sei im Folgenden das entsprechende folgendermaßen definiert:

```
between X (A,B,C) and Y (D,E,F)
```

Die Punkte A bis F sind lokal zu den entsprechenden Gliedern X und Y. Während der Simulation ergeben sich für die Punkte durch die Positionierung der Glieder Weltkoordinaten.

Drehgelenk Die Geraden AB und DE sind überdeckt und legen die Drehachse fest. Bei Drehgelenken gilt die Gleichheit der Ebenen ABC und DEF nicht. Der Winkel zwischen den Lotgeraden auf der Achse durch die Punkte C bzw. F bestimmt die gemessene Auslenkung des Gelenks.

Betrachten Sie bitte Abbildung 6.10. Der Punkt F gehört zu Y, dem zweiten angegebenen Glied und C zu X. Wenn man die Szenerie entlang der Drehachse von A nach B betrachtet, dann ist die Gelenkstellung derjenige Winkel, um den man Y gegen den Uhrzeigersinn drehen muss, damit die Lotgeraden auf der Drehachse durch die Punkte C und F parallel sind.

Schubgelenk Die Geraden AB und DE sind überdeckt und legen die Schubachse fest. In einem Schubgelenk gilt die Bedingung $A = D$ nicht. Die Auslenkung des Gelenks ist der Abstand der Punkte A und D, und sie ist positiv, wenn D in Richtung B verschoben ist.

Festlegung	*Drehgelenk*	*Schubgelenk*	*Zylindergelenk*
minimale Drehung	`minimal`	—	`minimal_rot`
maximale Drehung	`maximal`	—	`maximal_rot`
Ausgangdrehung	`init`	—	`init_rot`
minimale Verschiebung	—	`minimal`	`minimal_trans`
maximale Verschiebung	—	`maximal`	`maximal_trans`
Ausgangsverschiebung	—	`init`	`init_trans`

Tabelle 6.2: Die Schlüsselwörter für Minimal-, Maximal- und Initialstellungen der Gelenke.

Zylindergelenk Die Geraden *AB* und *DE* sind überdeckt und legen sowohl Dreh- als auch Schubachse fest. Ferner sind weder *ABC* und *DEF* identisch, noch gilt die Überdeckung *A* und *D*. Die Auslenkungen der beiden Freiheitsgrade werden gemessen wie bei Dreh- und Schubgelenk angegeben.

Für das oben angegebene Beispiel bedeutet das: Wenn die Punkte `schulterrechts` ($\widehat{=}A$) und `schulterlinks` ($\widehat{=}B$) in diesem Drehgelenk sinnvoll gewählte Namen haben, wird dem Roboter ermöglicht, mit dem Kopf zu nicken. Es bleiben noch die Begrenzungen für die Bewegungen an den Gelenken. Für jedes Gelenk müssen minimale, maximale und initiale Auslenkungen angegeben werden. Nehmen wir folgende Festlegungen für das obige Beispiel an:

```
minimal -10;
maximal 30;
init 0;
```

Wir müssen die Drehachse vom ersten Punkt, `schulterrechts`, zum zweiten Punkt, `schulterlinks` betrachten. Das nachfolgende Glied ($\widehat{=}Y$) ist `kopf`. Wenn der Kopf nach vorne geneigt ist, muss man ihn, bezüglich der Blickrichtung „von rechts nach links“, gegen den Uhrzeigersinn rotieren, um ihn wieder aufzurichten. Wenn wir die Werte interpretieren, kann der Kopf 30° nach vorne und 10° nach hinten geworfen werden. Die Ausgangsstellung ist das erhobene Haupt. In Tabelle 6.2 finden Sie die Schlüsselworte für die Gelenkbegrenzungen und die Initialwerte.

6.3.5 Beschreibung der geometrischen Form

Es stehen Ihnen zwei Möglichkeiten für die Eingabe der Oberflächenbeschreibung zur Verfügung:

1. über VRML-Dateien.
2. über die Konfigurationsdatei.

Wann immer es möglich ist, sollten Sie nach Variante 1 vorgehen.

6.3.5.1 Eingabe über VRML-Dateien

Wenn Sie die geometrische Form der Glieder mit einem CAD-Programm eingeben wollen, ist das der Weg, um die Daten in den Simulator zu bekommen. Allerdings bestehen einige Einschränkungen bezüglich des VRML-Formats, deren Einhaltung Sie gewährleisten müssen.

- Die VRML-Datei muss der Version VRML'97 entsprechen.
- Die VRML-Datei darf keine DEF- oder USE-Statements enthalten.
- Nur die Polygondaten werden berücksichtigt. Ferner müssen die Polygonpunkte gegen den Uhrzeigersinn angeordnet sein, wenn man von außen auf das Polygon schaut.


Praxistipp

Ersetzen Sie sofort die Punktdaten in den Gliedbeschreibungen, sobald Sie sinnvolle Werte haben und sich sicher sind, dass die eingegebene Form stimmt.
Fehleingaben der Punkte sind auch mit der grafischen Visualisierung von SIGEL nur schwer aufzuspüren, außer bei sehr einfachen Robotern.

Für die Daten können Sie ein beliebiges Koordinatensystem wählen, d.h. Sie können jedes Glied ohne Berücksichtigung der anderen Glieder modellieren. Die in der Konfigurationsdatei definierten Punkte müssen Sie dann auch in diesem Koordinatensystem angeben.

6.3.5.2 Eingabe über die Konfigurationsdatei

Wenn Sie die geometrischen Daten über die Konfigurationsdatei eingeben wollen, bleibt Ihnen nichts anderes übrig, als den Körper selbst in Polygone der Oberfläche zu zerlegen und die Polygonpunkte gegen den Uhrzeigersinn, wenn man das Polygon von außen betrachtet, einzugeben. Die Eingabe der Gliedbeschreibung ändert sich nicht. Sie müssen auch einen Dateinamen angeben, aber diese Datei wird niemals geöffnet, wenn Sie *am Schluss* der Konfigurationsdatei deren (auch nicht vorhandene) Inhalte erklären.

Betrachten Sie bitte Abbildung 6.11. Sie sehen dort einen Würfel mit einer schraffierten Frontfläche, deren Eckpunkte hervorgehoben sind. Wenn wir diese Eckpunkte gegen den Uhrzeigersinn durchlaufen, ergibt das folgende Beschreibung:

```
link foo {
  geometry "bar.wrl";
  [... weitere Glieddaten]
}

surface for "bar.wrl"
```

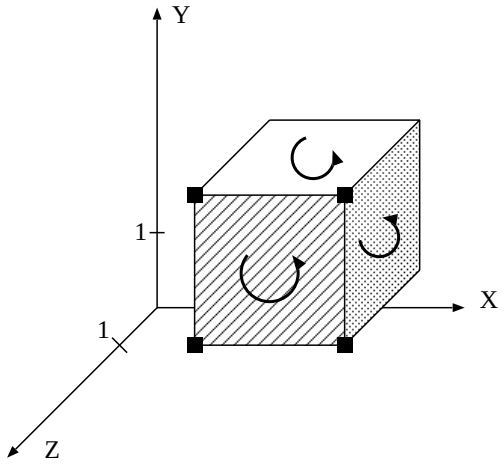


Abbildung 6.11: Polygone werden, von außen betrachtet, gegen den Uhrzeigersinn angegeben.

```
( 1, 2, 1 / 1, 0, 1 / 3, 0, 1 / 3, 2, 1 )
  [... weitere Polygone]
( 3, 2, 1 / 3, 0, 1 / 3, 0, -1 / 3, 2, -1 );
```

Das Polygon in der letzten Zeile dieses Beispiels entspricht der Würfelfläche die ganz rechts auf Abbildung 6.11 zu sehen ist, also jene, die parallel zur y/z-Ebene ist. Vergessen Sie nicht, auch die Punktdaten in den Gliedern zu aktualisieren. Wie bei der Eingabe über VRML-Dateien können Sie auch hier das Koordinatensystem je Oberflächenbeschreibung frei wählen.

6.3.6 Sensoren und Motoren

Als letzte Elemente des Roboters fehlen Sensoren und Motoren. Sensoren dienen dazu, dem Steuerungsprogramm Werte über den Roboter oder aus der Umwelt mitzuteilen, Motoren dazu, den Roboter aktiv zu bewegen. Implementiert sind bei den Sensoren derzeit nur diejenigen, die die Gelenkstellungen des Roboters messen. Ein Sensor wird wie folgt definiert:

```
sensor kniesensor {
  joint knie;
}
```

Damit wird für das Steuerungsprogramm eine Eingabemöglichkeit geschaffen, durch die es die Kniestellung erfahren kann. Das Gegenstück dazu, der Motor, der am Knie angebracht ist, lässt sich folgendermaßen definieren:

```
drive kniemotor {
```

```
  absolute knie;
  minimalforce 0.01;
  maximalforce 25.0;
}
```

Das Schlüsselwort `absolute` sagt aus, dass es sich bei dem Kniemotor um einen Motor handelt, der mit absoluten Positionen im zulässigen Bereich der Gelenkstellung angesteuert wird (wie z.B. ein Servomotor). Weitere Motortypen sind `relative`, für Motoren, denen man „5° nach rechts“ sagt, wie z.B. Schrittmotoren, und `force`, die direkt mit der auszuübenden Kraft angesteuert werden.

`minimalforce` und `maximalforce` geben die kleinste und größte Kraft in N (bzw. Drehmoment in Nm) an, mit denen der Motor agieren kann. Ergibt sich durch die Ansteuerung eine Kraft, die kleiner ist als die kleinstmögliche, so bleibt der Motor inaktiv.

6.3.7 Einschränkungen spezieller Dynamiksimulationen

Bisher beschrieben ist der Umfang, der für die Modellierung vorgesehen war. Die internen Datenstrukturen erlauben auch eine Speicherung all dieser Eigenschaften. Hinter der Modellierung steht jedoch die Dynamiksimulation, und die muss nicht zwangsläufig mit allen diesen Parametern arbeiten können. Experimentiert hat die Projektgruppe mit den Simulationsbibliotheken *DynaMo* und *DynaMechs*. Der Verwendung anderer Bibliotheken mit ihren Möglichkeiten und Einschränkungen steht prinzipiell nur der Arbeitsaufwand der Integration entgegen.

6.3.7.1 DynaMo

Mit der Bibliothek *DynaMo* lassen sich bis auf die Coulomb'sche Reibung alle Eigenschaften simulieren. Leider ist es der Projektgruppe nicht gelungen, diese fehlerfrei zu nutzen. Es wird davon abgeraten, *DynaMo* als Simulationsbibliothek anzuwählen.

6.3.7.2 DynaMechs

Die Bibliothek *DynaMechs* funktioniert stabil, ihre Benutzung zieht aber einige Einschränkungen nach sich.

1. Kinematische Ketten dürfen nicht geschlossen sein, d.h. die Glieder und Gelenke müssen eine Baumstruktur bilden.

2. Es dürfen nur Dreh- und Schubgelenke verwendet werden. Auch die feste Verbindung zweier Glieder ist *nicht* erlaubt.

3. Es gibt nur kraftangesteuerte Motoren.

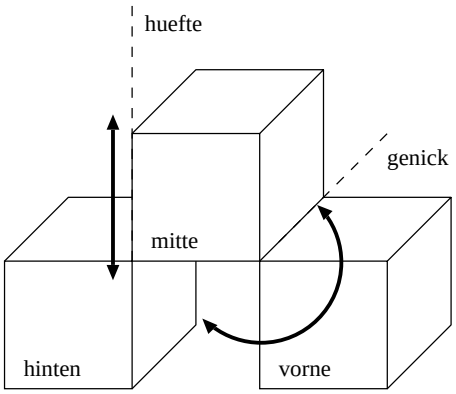


Abbildung 6.12: Der Beispielroboter.

4. Kollisionen zwischen Gliedern werden nicht erkannt. Das bedeutet, dass man die Bewegungsfreiheit der Gelenke stärker einschränken muss, wenn sich die Glieder nicht berühren sollen.

6.3.8 Ein durchgehendes Beispiel

In diesem letzten Unterkapitel zur Konstruktion von Robotermodellen werden alle Aspekte an einem Beispiel, das in Abbildung 6.12 zu sehen ist, durchgeführt.

Der Roboter besteht aus den drei Gliedern *hinten*, *mitte* und *vorne*, die durch zwei Gelenke *huefte* und *genick* verbunden sind. Bei allen Gliedern handelt es sich um die gleichen Einheitswürfel, die aus dem auf Seite 201 definierten Stahl gefertigt sind. In Abbildung 6.13 ist der Quelltext der Glieder zu sehen.

Das Gelenk von `mitte` nach `hinten` soll ein Schubgelenk sein. Der Roboter soll es soweit hochziehen können, dass das Glied `hinten` mit `mitte` auf einer Linie liegt. Das Gelenk von `mitte` nach `vorne` soll ein Drehgelenk sein, das soweit ausgelenkt werden kann, bis die beteiligten Glieder aneinanderstoßen. Wie Sie aus den Punktangaben in Abbildung 6.13 entnehmen können, befinden sich alle Gelenke in der Stellung Null, wenn der Roboter aussieht wie in Abbildung 6.12. In Abbildung 6.14 sind die Gelenkdefinitionen angegeben. Die Definitionen der Motoren finden Sie in Abbildung 6.15. Wundern Sie sich nicht über die Größenordnung der Kräfte. Alle Angaben sollen in SI-Einheiten sein, daher ist ein Würfel $8m^3$ groß und wiegt 62,6 Tonnen. Zum Schluss ist in Abbildung 6.16 die Oberflächenbeschreibung eines Würfels dargestellt. Bitte beachten Sie: Wenn Sie sich entschließen, geometrische Formen in die Konfigurationsdatei aufzunehmen, müssen diese sich am Ende der Datei befinden.

```
link hinten {
  geometry "wuerfel.wrl";
  material steel;
  point midoffset = (1, 1, 1);
  point midaxis = (1, 3, 1);
  point midfixed = (1, 1, -1);
}
link mitte {
  torso;
  geometry "wuerfel.wrl";
  material steel;
  point backoffset = (-1, -1, 1);
  point backaxis = (-1, 1, 1);
  point backfixed (-1, -1, -1);
  point frontoffset = (1, -1, 1);
  point frontaxis = (1, -1, -1);
  point fronthand = (1, 1, 1);
}
link vorne {
  geometry "wuerfel.wrl";
  material steel;
  point midoffset = (-1, 1, 1);
  point midaxis = (-1, 1, -1);
  point midhand = (-1, 3, 1);
}
```

Abbildung 6.13: Die Gliedbeschreibungen des Beispielroboters.

```
joint translational huefte {
  between hinten (midoffset, midaxis, midfixed)
      and mitte (backoffset, backaxis, backfixed);
  minimal 0;
  maximal 2;
  init 0;
}
joint rotational genick {
  between mitte (frontoffset, frontaxis, fronthand)
      and vorne (midoffset, midaxis, midhand);
  minimal -90;
  maximal 90;
  init 0;
}
```

Abbildung 6.14: Die Gelenkbeschreibungen des Beispielroboters.

```
drive halsmotor {
  force genick;
  minimalforce 8.0;
  maximalforce 50000.00;
}
drive beinmotor {
  force huefte;
  minimalforce 10.0;
  maximalforce 40000.00;
}
```

Abbildung 6.15: Die Motoren des Beispielroboters.

```
surface for "wuerfel.wrl"
( -1,-1,-1 / -1,1,-1 / 1,1,-1 / 1,-1,-1 )
( -1,-1,1 / 1,-1,1 / 1,1,1 / -1,1,1 )
( -1,-1,-1 / 1,-1,-1 / 1,-1,1 / -1,-1,1 )
( 1,1,1 / 1,1,-1 / -1,1,-1 / -1,1,1 )
( 1,-1,-1 / 1,1,-1 / 1,1,1 / 1,-1,1 )
( -1,-1,-1 / -1,-1,1 / -1,1,1 / -1,1,-1 ) ;
```

Abbildung 6.16: Die Oberflächenbeschreibung eines Würfels als Eintrag in der Konfigurationsdatei.

6.4 Einstellungen und Grenzen des Simulators

Zur Visualisierung und Bewertung von Individuen benutzt das System SIGEL eine physikalische Simulation, deren Interna in Kapitel 4.4 beschrieben sind. Hier wird auf Benutzung und Grenzen der Simulation eingegangen.

6.4.1 Einstellungen

In der graphischen Benutzeroberfläche des SIGEL-Masters lassen sich die Simulationsparameter getrennt nach allgemeinen Einstellungen (General settings) und solchen, die speziell für eine der beiden Simulationsbibliotheken benötigt werden, tätigen (DynaMechs, DynaMo). Dies gilt ebenfalls für die Einstellung der Umgebung, in der der Roboter simuliert wird (Environment). Zu den allgemeinen Einstellungen gehört insbesondere die Wahl der Simulationsbibliothek. Der Benutzer hat die Wahl zwischen DYNAMECHS und DYNAMO. Da wir letztendlich aber nur die Einbindung von DYNAMECHS verfolgt haben, sollte die Standardeinstellung DYNAMECHS belassen werden.

Im Folgenden wird auf die einzelnen Einstellungsmöglichkeiten allgemeiner Natur und speziell in Bezug auf die Bibliothek DYNAMECHS eingegangen. Die Parameter, die für die Benutzung von DYNAMO entscheidend sind, werden hier aus oben genanntem Grund nicht beschrieben.

6.4.1.1 Schrittweite (Step Size)

Die Simulation überführt den Zustand der simulierten Objekte (in unserem Fall nur der Roboter) von einem Zeitpunkt zum nächsten. Die Zeitspanne, um die dabei fortgeschritten wird, nennt sich Schrittweite. Da die Verfahren zur physikalischen Simulation das realistische Verhalten des Roboters nur annähern, kann sich auch das Ergebnis der Simulationsberechnungen mit veränderter Schrittweite ändern. Je kleiner die Schrittweite, desto genauer die Simulation. Natürlich müssen zur Simulation einer bestimmten Zeitspanne bei kleiner Schrittweite mehr Zwischenschritte berechnet werden, der Vorgang nimmt mehr Rechenzeit in Anspruch.


Praxistipp

Bei komplexen Robotern kann eine zu große Schrittweite zu extrem hohen Kräften führen, die den Roboter in unendliche Weiten schleudern. Ein Verringern der Schrittweite kann hier die Lösung sein. Ein guter Wert, mit dem alle von uns modellierten Roboter simulierbar waren, ist 0.002.
Zunächst sollten jedoch ruhig größere Werte beginnend bei 0.02 ausprobiert werden, da so nicht zuletzt die Evolution enorm beschleunigt werden kann.

6.4.1.2 Simulationsbedingungen (Simulation conditions)

Unter dieser Rubrik findet sich die Einstellung der Simulationszeit (Time To Simulate) und des Random Seeds der Simulation. Letzterer ist allerdings nur als Zu-

kunftsfunktion vorgesehen und momentan noch inaktiv.

Die Simulationszeit entscheidet, für welche Zeitspanne die Fitnessfunktionen den Roboter mit dem Programm des entsprechenden Individuums simulieren. Für die Simulation im Rahmen der Visualisierung ist die Simulationzeit unentscheidend.

6.4.1.3 Integrator

Die physikalische Simulation verwendet einen numerischen Integrator zur Berechnung von Position und Geschwindigkeit der einzelnen Roboterglieder. Hier hat der Benutzer die Wahl zwischen den Typen...

- ...EULER. Dies ist der schnellste und einfachste Integrator.
- ...RUNGE KUTTA 4. Dies ist eine Implementierung des Runge-Kutta Integrators vierter Ordnung.
- ...RUNGE KUTTA 45. Dies ist eine Implementierung des Runge-Kutta Integrators vierter/fünfter Ordnung mit adaptiver Schrittweite.

Der Leser muss weder die Details der Eigenarten der verschiedenen Integratoren noch den Grund ihrer Bezeichnungen verstehen, sondern sollte den voreingestellten RUNGE KUTTA 4 verwenden.

6.4.1.4 Gelenkkonstanten (Joint constants)

Unter dieser Rubrik sind einstellbare Konstanten zusammengefasst, die sich auf das Simulationsverhalten von Gelenken beziehen.

Joint Limits spring constant: Diese Konstante nimmt Einfluss auf die Stärke der Gegenkraft, die die Simulation bei Überschreitung der erlaubten Minimal- und Maximalauslenkungen von Gelenken ausübt. Der voreingestellte Wert von 50 sollte für viele Fälle gut funktionieren.


Praxistipp

Macht man die Erfahrung, dass die Auslenkungsgrenzen nicht streng genug eingehalten werden, kann die Wahl eines höheren Wertes zum Erfolg führen.

Joint Limits damper constant: Diese Konstante steuert einen von der aktuellen Geschwindigkeit des Gelenks abhängigen Term, der von der Gegenkraft subtrahiert wird, um Schwingungen zu vermeiden. Auch hier sollte der voreingestellte Wert von 5 gute Ergebnisse liefern.


Praxistipp

Durch eine zu hohe Dämpfungskonstante kann es bei komplexeren Robotern zu unnatürlich großen Kräften kommen. Hier empfiehlt sich, die Konstante auf den Wert 1 zu setzen. Dies sollte ausreichen, um dann trotzdem Schwingungen zu vermeiden.

Joint friction constant: Diese Konstante steuert die Stärke der Gelenkreibung. Hat man durch den zu simulierenden Roboter hier keine speziellen Vorgaben, sollte der voreingestellte Wert von 0.35 übernommen werden.

6.4.1.5 Gravitation und Startposition

Gravitation und Startposition sind die Einstellungen der zu simulierenden Umgebung, die beide Simulationsbibliotheken gemein haben. Der Benutzer kann einen Gravitationsvektor angeben sowie einen Positionsvektor, der den Ort beschreibt, an dem der Roboter anfangs plaziert wird. Bei beiden Einstellungen ist zu berücksichtigen, dass sich die Y-Achse othogonal zur Bodenebene befindet.

6.4.1.6 Bodenkonstanten

Für die Simulation mit DYNAMECHS können 6 Konstanten manipuliert werden, die die Bodeneigenschaften beeinflussen.

Planar spring constant und Normal spring constant: Ähnlich wie bei den Konstanten zur Einhaltung von Minimal- und Maximalauslenkung steuern diese Konstanten die Gegenkräfte, die erzeugt werden, um ein Eintreten von Objekten in den Boden zu verhindern. Auch hier gilt: Größere Werte erzeugen größere Kräfte - der Boden wird elastischer.

Planar damper constant und Normal damper constant: Genau wie bei Minimal- und Maximalauslenkung steuern diese Konstanten die Dämpfung der Gegenkräfte in Abhängigkeit von der aktuellen Geschwindigkeit. Die Voreinstellung sollte hier gute Ergebnisse liefern.

Static friction coefficient und Kinetic friction coefficient: Diese Reibungskonstanten steuern die Reibungskräfte, die dem “Rutschen” von Objekten über den Boden entgegenwirken. Ein kleiner Wert entspäche also Eis, ein hoher etwa Gummi.

Der kinetic friction coefficient muss dabei kleiner als der static friction coefficient sein. Ersterer Koeffizient wird angewandt, um die Reibung eines bereits gleitenden Objekts zu berechnen, zweitere um die Reibungskraft zu berechnen, die überwunden werden muss, um in den gleitenden Zustand überzugehen.

6.4.2 Grenzen des Simulators

Hier werden zwei wesentliche Einschränkungen des Simulators angesprochen, die wegen der verwendeten Simulationsbibliothek nicht geändert werden konnten.

6.4.2.1 Kollisionserkennung und -behandlung

Sinnvollerweise sind Überschneidungen zwischen Robotergliedern unerwünscht. Leider Unterstützt die Simulationsbibliothek DYNAMECHS keine Kollisionserkennung zwischen Gliedern. Auch die Einbindung einer externen Bibliothek wie z.B. SOLID, die Kollisionen erkennt und sie dann von DYNAMECHS behandeln lässt, ist in letzterer Bibliothek nicht vorgesehen. Daher muss versucht werden, die gegenseitige Durchdringung der Glieder durch die Wahl von Minimal- und Maximalauslenkungen der einzelnen Gelenke zu vermeiden.

Natürlich wird eine Kollisionserkennung und -behandlung mit dem Boden durchgeführt. Dazu werden in der momentanen Implementierung einfach alle Knotenpunkte der Geometrie eines Gliedes als Kontaktpunkte an DYNAMECHS übereicht. Auch hier könnte aus Effizienzgründen eine eingeschränkte Auswahl von Kontaktpunkten sinnvoll sein. Dies ist jedoch momentan nicht geplant.

6.4.2.2 Zusammenhang zwischen Schrittweite und Dauer eines MOVE-Befehls

Kräfte bzw. Drehmomente können bei DYNAMECHS nur pro Zeitschritt angewandt werden. Da diese aber über die Dauer eines MOVE-Befehls wirken sollen, ist es sinnvoll, dass diese Dauer ein Vielfaches der Schrittweite ist. Dies sollte beim Planen eines Experimentes berücksichtigt werden.

Praxistipp

Es ist sowohl realistischer als erfahrungsgemäß auch erfolgsversprechender, eine kleinere/mittlere Kraft bzw. ein Drehmoment über eine längere Zeitspanne wirken zu lassen, als eine große über eine kurze Zeit hinweg. Was konkret „klein“, „groß“ und “mittel” hier bedeutet, hängt vom Roboter ab - besitzt dieser eine große Masse, sind auch größere Kräfte erforderlich.
Beispiel: Ein Drehmoment von $50Nm$ als Maximaleinstellung eines Motors in Kombination mit einer Dauer von 0.1 Sekunden ist sinnvoller, als $1000Nm$ über einen Zeitraum von 0.005 Sekunden wirken zu lassen.

6.5 Die Evolution und ihre Parameter

In diesem Unterpunkt wird beschrieben, wie eine genetische Evolution zu starten ist. Wie man in dem Programm SIGEL unter dem Punkt GP-PARAMETER sehen kann, gibt es zahlreiche Parameter, die man richtig einstellen sollte, um eine vernünftige genetische Evolution zu bekommen.

6.5.1 Genetic Programming

Unter diesem Reiter kann man die Eigenschaften des Roboterprogramms, die Wahrscheinlichkeit der genetischen Operatoren und die Art der Fitnessfunktion,

die bei der Evolution benutzt wird, einstellen. (siehe Abbildung 6.17).

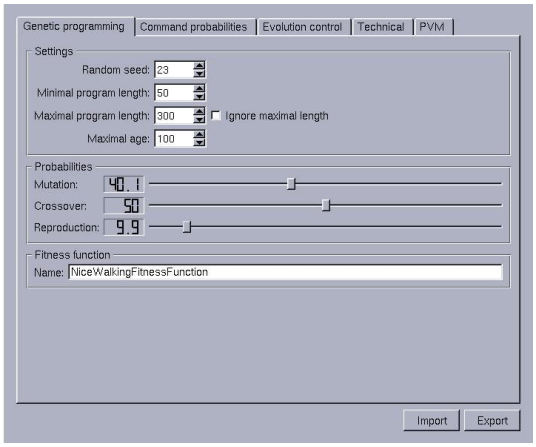


Abbildung 6.17: Genetic Programming

Settings

Unter dem Punkt *Settings* kann man den *Random Seed* für die GP einstellen, dieser dient dazu, Ergebnisse reproduzieren zu können. Man kann hier auch die Eigenschaften des Roboterprogramms einstellen, also die minimale und maximal Länge und ob diese ignoriert werden sollte. Desweiteren kann man Individuen eines gewissen Alters aussortieren.

Probabilities

Unter diesem Punkt kann man die Wahrscheinlichkeiten der genetischen Operatoren *Reproduktion*, *Crossover* und *Mutation* einstellen.

Fitness function

Unter dem Punkt *Fitness function* kann man die Art der Fitnessfunktion eingeben, nach der die Bewertung und Gewichtung der Individuen bestimmt wird. Dabei werden momentan folgende Fitnessfunktionen unterstützt: SimpleFitnessFunction, NiceWalkingFitnessFunction und RealSpeedFitnessFunction.

6.5.2 Command probabilities

Hier kann man die Wahrscheinlichkeit der verschiedenen Instruktionen eines Roboterprogramms einstellen. Je nach Wahrscheinlichkeit einer Instruktion wird diese Instruktion in einem Roboterprogramm vorhanden sein (siehe Abbildung 6.18).

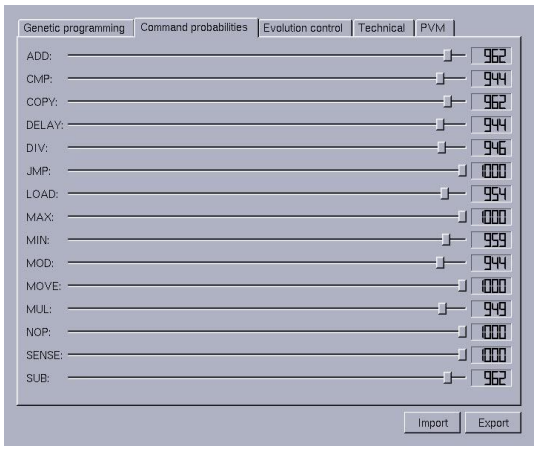


Abbildung 6.18: Wahrscheinlichkeiten der Instruktionen

6.5.3 Evolution control

Dieser Reiter bietet die Möglichkeit die verschiedenen Steuerungen der Evolution gezielt einzustellen (siehe Abbildung 6.19).

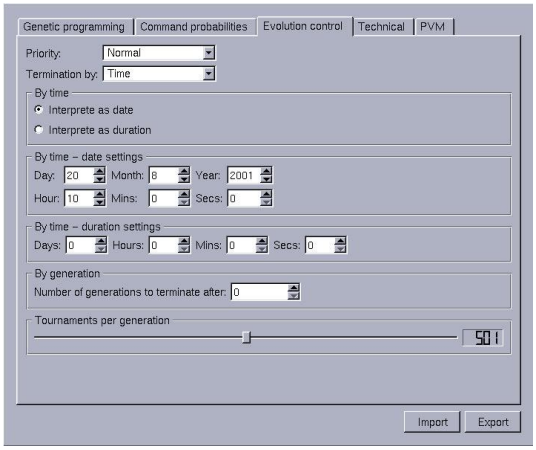


Abbildung 6.19: Steuerung der Evolution

Priority

In dem Punkt *Priority* kann man die Priorität der Evolution auswählen. Das heißt je höher diese ist, desto weniger Ressourcen bleiben für die GUI während der Evolution übrig.

Termination by

Hier kann man die Terminierungsbedingung auswählen, bei der die Genetische Evolution stoppt. In *Termination by* kann man zwischen verschiedenen Terminierungsbedingungen auswählen. Wenn man als User die Terminierungsbedingung auswählt, heißt das, dass die Evolution nur vom User gestoppt werden kann. Bei der Wahl von *Time* wird die Evolution so lange laufen, bis die Terminierungszeit erreicht wurde und dann automatisch gestoppt.

By time

Hier kann man zwischen zwei Optionen auswählen, nämlich *Interprete as date* für Interpretation der Zeit als Datum oder *Interprete as duration* für Dauer der Evolution.

By time - date settings

Wenn *Interprete as date* aktiviert wird, dann kann man unter dem Punkt *By time - date settings* die Terminierungszeit eingeben. Außer den Zeitraum in Stunden, Minuten und Sekunden, kann man auch den Tag, Monat und das Jahr der Evolutionsterminierung eingeben.

By time - duration settings

Wenn *Interprete as duration* aktiviert wird, dann kann man unter dem Punkt *By time - duration settings* die Dauer eingeben. Der Zeitraum der Evolutionsdauer kann in Tagen, Stunden, Minuten und Sekunden eingegeben werden.

By generation

Als Terminierungsbedingung kann man auch *Generation* auswählen, dann wird die Evolution nur bis zum Erreichen einer bestimmten Generationszahl ausgeführt. Die maximale Anzahl Generationen stellt man im Punkt *By generation* ein. Man kann auch *Time or Generation* als Terminierungsbedingung für die Evolution auswählen. Das heißt hier läuft die Evolution bis die erste der beiden Terminierungsbedingungen eintrifft.

Tournaments per generation

Die Anzahl der Turniere pro Generation wird unter dem Punkt *Tournaments per generation* eingestellt.

6.5.4 Technical

Unter dem Punkt *Technical* kann man die *Graveyard* und *Pool image options* einstellen (siehe Abbildung 6.20).

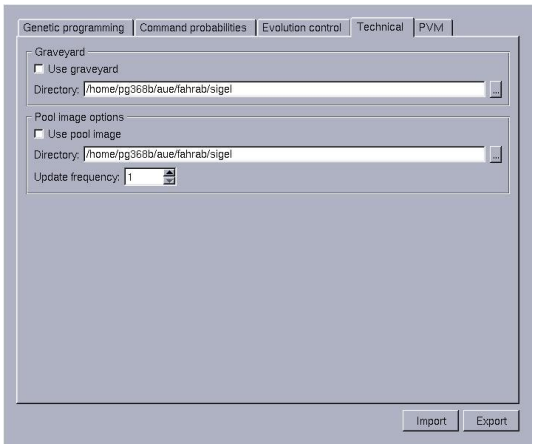


Abbildung 6.20: Technische Daten

Graveyard

Bei der Aktivierung des *Graveyards* werden alle verstorbenen Individuen, die nicht mehr an der Evolution teilnehmen dürfen gespeichert. Die verstorbenen Individuen werden unter dem Format **.ind* in dem vom Benutzer eingegebenen Verzeichnis gespeichert.

Praxistipp

Achtung: Erfahrungen haben gezeigt, dass mit dieser Option sehr schnell viel Festplattenspeicher verbraucht wird. Um die Evolution nachvollziehen zu können empfiehlt es sich eher, das u.g. Poolimage zu nutzen.

Pool image options

Hier kann man bei der Aktivierung der *Use pool image* eine Kopie der Population unter dem Format **.pol* in einem angegebenen Verzeichnis speichern. Je nach Einstellung der *Update frequency* kann man die Kopien frequentiert erstellen. Das heißt bei einer *Update frequency* von 1 wird eine Kopie jeder Generation gespeichert und bei 2 wird eine Kopie jeder zweiten Generation gespeichert usw.

6.5.5 PVM

Unter diesem Reiter können die Einstellungen von *PVM* geändert werden. So kann man unter *Hosts* die verschiedenen Hosts, die an dem Parallelisierungsprozess teil-

nehmen eingeben. *PVM* ist dazu da, um die große Menge der Fitnessberechnungen der beteiligten Individuen auf verschiedene Rechner aufzuteilen (siehe Abbildung 6.21).

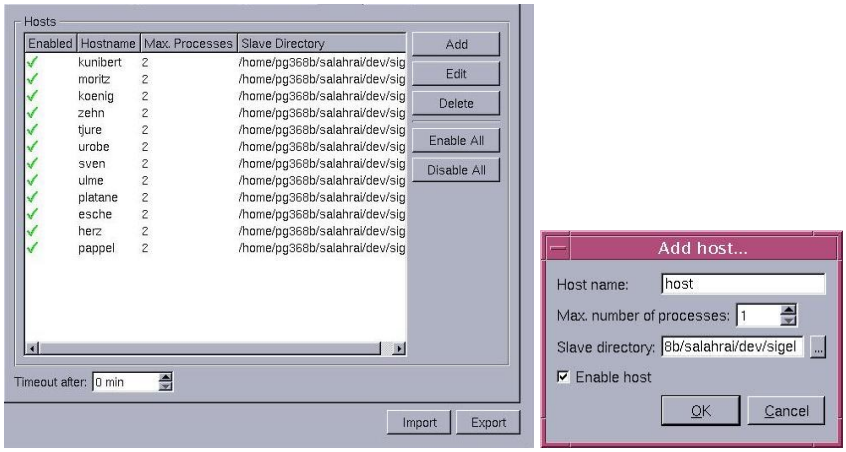


Abbildung 6.21: Einstellung der PVM und Einfügen von einem Host

Add

Hier gibt man den Host, die maximale Anzahl der Prozesse und das Verzeichnis, in dem der SIGEL-Slave steht ein. Jener wird dann im Parallelisierungsprozess teilnehmen. Je nach Wunsch kann man hier auch den Host deaktivieren oder aktivieren (siehe Abbildung 6.21).

Edit

Hier kann man die maximale Anzahl der Prozesse und das Verzeichnis, wo der durchzuführende Prozess steht neu einstellen. Man kann hier auch den Host neu aktivieren oder deaktivieren, der Dialog entspricht dem oben genannten zum Einfügen eines Hosts (vgl. Abbildung 6.21).

Delete, Enable All und Disable All

Durch *Delete* kann man einen Host löschen. Man kann alle Hosts aktivieren (mit *Enable All*) oder deaktivieren (mit *Disable All*).

6.6 Import und Export von GP Daten

Wie bereits unter 4.5.8 erwähnt, unterstützt das GP System den Import von Individuen und Programmen, damit Individuen und Programme zwischen Experimenten

ausgetauscht werden können. Dieses Kapitel erläutert, wie Programme und Individuen manuell erstellt, importiert und exportiert werden können. Für die Erstellung kann ein beliebiger Texteditor verwendet werden.

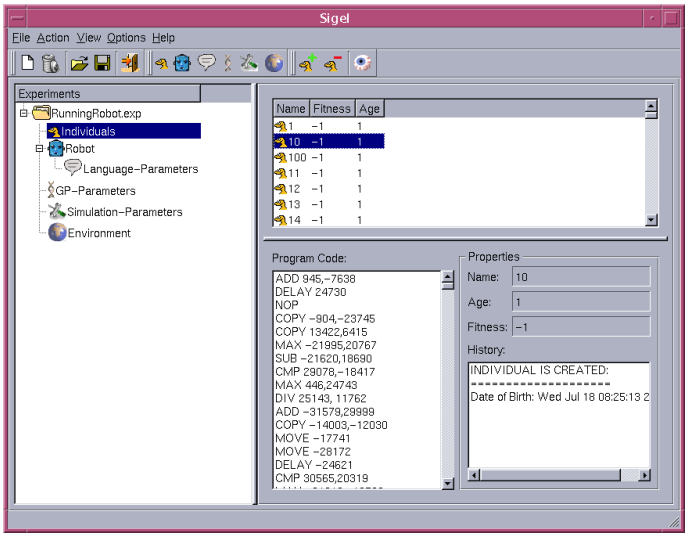


Abbildung 6.22: Darstellung der Individuen

6.6.1 Import eines Programms

Bevor ein Programm importiert werden kann, muss ein Individuum in der Individuendarstellung (s. 6.22) gewählt werden, welches das zu importierende Programm beinhalten soll. Dabei ist zu beachten, dass das vorherige Programm bei einem Import gelöscht wird. Ist ein Individuum gewählt, so kann man ein Programm entweder über *File - Import - Programm*, über *Ctrl + Shift + K* oder über das kontextsensitive Menü der Baumdarstellung importieren.

6.6.2 Export eines Programms

Bevor ein Programm exportiert werden kann. muss ein Individuum in der Individuen-Darstellung (s. 6.22) gewählt werden, welches das zu exportierende Programm beinhalten soll. Ähnlich dem oben genannten Import kann nun über Hauptmenü, *Ctrl + Alt + K* oder kontextsensitives Menü das Programm exportiert werden.

6.6.3 Manuelles Erstellen von Programmen

Die Syntax der Programme entspricht der Beschreibung von Abschnitt 4.3.2. Daher sei für die Beschreibung des Befehlsumfangs auf dieses Kapitel verwiesen. Alle

Befehle aus Abschnitt 4.3.2 können ausnahmslos im Rahmen der SIGEL Programme verwendet werden.Die Programmlänge ist beliebig, und es kann ein beliebiger Editor verwendet werden.

6.6.4 Im- und Export eines Individuums

Individuen können im Sinne einer interaktiven Evolution exportiert und in fremde Experimente importiert werden. Dabei werden nicht erlaubte Instruktionen im Laufe der Evolution durch NOPs ersetzt. Diese Funktion funktioniert in der GUI analog zum Im- und Export von Programmen.

6.6.5 Manuelle Nachbearbeitung eines Individuums

Auch Individuen können manuell mit einem Texteditor bearbeitet werden. Die Daten eines exportiertes Individuums werden grundsätzlich mit dem Schlüsselwort

```
INDIVIDUAL BEGIN{
```

eingeleitet und mit dem Schlüsselwort

```
INDIVIDUAL END{
```

wieder abgeschlossen. Ist ein Individuum exportiert worden, so ist es ebenfalls möglich das enthaltene Programm wie unter 6.6.3 beschrieben zu bearbeiten. Zu beachten ist hierbei lediglich, dass das Programm von den Schlüsselwörtern

```
PROGRAM BEGIN{
```

und

```
}PROGRAM END
```

eingeschlossen ist. Die restlichen Daten des Individuums können ebenfalls editiert werden. Jedoch werden die meisten Daten bei einem späteren Import automatisch angepasst, so dass viele manuelle Einträge keine Wirkung haben.

Nachwort

Ich erwache. Meine Sinne sind noch verwirrt, aber langsam erfasse ich meine Umwelt. Ich stehe aufrecht, um mich herum ist eine weite Ebene, über mir ein strahlend weißer Himmel. Ich schaue mich um, Blicke hin und her, suche etwas oder jemand. Irgend etwas anderes als mich, aber außer mir ist nichts und niemand hier. Ich schaue auf den Boden. Auf ihm ist ein perfektes, rechtwinkliges Gitter aufgezeichnet. Ich schaue die Linien entlang, aber keine Unregelmäßigkeiten sind zu entdecken. Die Linien verlaufen parallel, ohne Abweichung und verschwinden in der Unendlichkeit des Horizonts. Ein Gefühl der Verlorenheit überkommt mich, schlägt wie eine Welle über mir zusammen. Wenn jemand oder etwas außer mir existiert, dann ist es nicht hier. Da alle Richtungen, in die ich mich wenden kann, gleich verheißungsvoll erscheinen, ist es egal wohin ich gehe, nur weg hier. Ich mache vorsichtig einen Schritt nach vorne, noch unsicher, denn ich habe das Gefühl, als ob ich noch nie in meinem Leben auch nur eine Schritt weit gelaufen bin. Ich mache einen weiteren Schritt, schon weniger unsicher, dann noch einen und noch einen. Ich kann laufen, schießt es mir durch den Kopf. Ich laufe, laufe weiter, laufe schneller, laufe bis ich das Gefühl habe, über den Boden zu fliegen. Ich versuche etwas in der Richtung zu erkennen, in die ich mich bewege. Aber ich erkenne nichts, nur das ewig gleiche Bodengitter. Vielleicht bin ich noch nicht weit genug gelaufen. Ich versuche noch schneller einen Fuß vor den anderen zu setzen, versuche genau an einer Linie entlang zu laufen. Keine unnötigen Abweichungen nach rechts oder links, die halten mich nur auf. Es muss noch etwas anderes geben. Ich kann doch nicht ganz alleine hier sein. Wo ist der, der mich geschaffen hat? Warum ist er nicht hier und warum sollte er nur mich geschaffen haben? Ich muss weiter, weiter zu den anderen. Und je schneller ich mich bewege, um so schneller bin ich da.

Literaturverzeichnis

[1] Thomas Bäck. Evolutionäre Algorithmen und die Rolle der Mutation in genetischen Algorithmen. Technical report, Informatik Centrum Dortmund e.V., Göttingen 1995.

[2] Hans-Paul Schwefel und Thomas Bäck. Künstliche Evolution - eine intelligente Problemlösungsstrategie? Technical report, Universität Dortmund, Fachbereich Informatik, 1992.

[3] Thomas Bäck. Evolutionary Algorithms. Technical report, University of Dortmund, Department of Computer Science, 1992.

[4] Schöneburg, E. Heinzmann, and S. F. Feddersen. *Genetische Algorithmen und Evolutionsstrategien*. Addison-Wesley, 1994.

[5] Koza. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*. Stanford University, 1990.

[6] P. J. Angeline. *Evolutionary Algorithms and Emergent Intelligence. PhD thesis*. Ohio State University, 1993.

[7] W. A. Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programms. PhD thesis*. University of Southern California, Department of Electrical Engineering Systems, 1994.

[8] L. Altenberg. *The evolution of evolvability in genetic programming. In Kinnear, Jr., K. E., editor, Advances in Genetic Programming*. MIT Press, Cambrige, MA, 1994.

[9] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming - An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. dpunkt.verlag, 1998.

[10] V. Nissen. *Einfuhrung in Evolutionäre Algorithmen*. Vieweg, 1997.

[11] Brian Mirtich. *Rigid Body Contact: Collision Detection to Force Computation*. http://www.merl.com, 1998.

[12] Jeff Trinkleand Jong-Shi Pang and Sandra Sudarsky. *On Dynamic Multi-Rigid-Body Contact Problems with Coulomb Friction*, 1995.

[13] S. Gottschalk and M.C. Lin and D. Manocha. *OBBTree, A Hierarchical Structure for Rapid Interference Detection*.
via Internet (22.08.2001):
http://www.cs.unc.edu/~geom/OBB/OBBT.html.

[14] J.J. Craig. *Introduction to robotics*. Addison-Wesley, 1991.

[15] P. MacKerrow. *Introduction to robotics*. Addison-Wesley, 1991.

[16] M. A. Salant. *Introduction to robotics*. 1991.

[17] M. Vukobratovic. *Introduction to robotics*. Springer, 1989.

[18] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley-Longman, 3. auflage edition, 1998.

[19] Michael Hanus. The Integration of Function into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[20] Michael Hanus. *A Truly Integrated Functional Logic Language*.
via Internet (22.08.2001):
http://www.informatik.uni-kiel.de/~curry/.

[21] *DynaMo Homepage*.
via Internet (22.08.2001) :http://www.win.tue.nl/~bartb/dynamo/.

[22] *SOLID Homepage*.
via Internet (22.08.2001) :http://www.win.tue.nl/~gino/solid/.

[23] *DynaMechs Homepage*.
via Internet (04.09.2001) :
http://dynamechs.sourceforge.net/.

[24] *QT On-Line Reference Documentation*.
via Internet (22.08.2001) :
http://doc.trolltech.com/.

[25] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.1)*, 1997.

[26] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, 1993.

[27] *Eine kurze Einführung in CVS*.
via Internet (22.08.2001) :
wwwipr.ira.uka.de/~paro/CVS.html.

[28] *Eine ganz kurze Einführung in CVS*.
via Internet (22.08.2001) :
www.kj.uue.org/papers/cvs-handout/.

[29] *CVS Kurzanleitung*.
via Internet (22.08.2001) :
www.numerik.uni-kiel.de/~khe/cvs/cvs.html.

[30] *CVS*.
via Internet (22.08.2001) :
linux.uni-regensburg.de/kurse/progdev/make/cvs.html.

[31] *GNU make, A Program for Directing Recompilation*.
via Internet (22.08.2001) :
www.gnu.org/manual/make/html_node/make_toc.html.