

Contents

1 Geschicht	1
1.1 Industrieroboter	1
1.2 Serviceroboter	1
1.2.1 Definition	1
1.2.2 Klassen	1
2 Software-Architekturen für mobile Robotersysteme	2
2.1 Probleme und Anforderungen	2
2.1.1 Definition mobile Roboter	2
2.1.2 Umgebung mobiler Roboter	2
2.1.3 Roboterkontroll-Architekturen	2
2.1.4 Anforderungen an das Kontrollsyste eines autonomen Roboters	3
2.2 Mögliche Modelle	3
2.2.1 Klassisches Modell - der funktionale Ansatz	3
2.2.2 Verhaltensbasiertes Modell	4
2.2.3 Hybrider Ansatz	5
2.2.4 Probabilistische Robotik	5
2.2.5 Subsumption-Architektur in Bezug auf die Anforderungen des Robotersteuerungssystems	6
2.3 ROS - Robot Operating System	7
2.3.1 Entwicklung	7
2.3.2 Design Prinzipien	7
2.3.3 Publish - Subscribe	8
2.3.4 Parameter Server und Konfigurationsdateien	8
3 Lokalisation autonomer mobiler Robotersysteme	10
3.1 Abgrenzung: Lokalisation - Mapping - SLAM - Navigation	10
3.2 Varianten der Selbstlokalisierung	10
3.3 Relative Lokalisierung versus Absolute Lokalisierung	11
3.4 Transformation von Koordinatensystemen lokale <-> globale	11
3.5 Karten für statistische und dynamische Umgebungen	12
3.5.1 Mapping Methoden	13
3.5.2 Arten von Modellen	13
3.5.3 Kontinuierliche Metrische Karten	14
3.5.4 Grid Maps - Rasterkarten	14
3.5.5 Adaptive Unterteilung	15
3.5.6 Weitere Beispiele für Umgebungskarten	16
3.5.7 Topologische Karten	16
3.5.8 Hybrid Maps	17
3.6 Passive und Aktive Selbstlokalisierung	17
3.6.1 Passive Verfahren	17
3.6.2 Aktive Verfahren	17

3.7	Landmarken	17
3.7.1	Definition	17
3.7.2	Natürliche Landmarken	18
3.7.3	Künstliche Landmarken	18
4	Fortbewegung, Lokalisierungsalgorithmen	19
4.1	Relative Lokalisierung	19
4.1.1	Dead Reckoning	19
4.1.2	Odometrie	20
4.1.3	2D-Scanmatching	21
5	Navigation	23
5.1	Bekanntes vs. unbekanntes Terrain	23
5.2	Navigation in unbekanntem Terrain	23
5.2.1	Konturverfolgung	23
5.2.2	Sensorbasierte Planer - Navigation mit Hinderniskontakt	24
5.2.3	Labyrinth	26
5.3	Pfadplanung für mobile Roboter in bekanntem Terrain	27
5.3.1	Bewegungsplan für mobile Roboter	27
5.3.2	Konfigurationsraum	28
5.4	Algorithmen und Methoden	29
5.4.1	Dijkstra	30
5.4.2	A*	30
5.4.3	Wegsuche mit dem Sichtgraph-Algorithmus	30
5.4.4	Voronoi-Diagramme	32
5.4.5	Navigation in einer Rasterkarte	34
5.4.6	Potentialfeld methode	35
6	Probabilistische Methoden und Kartierungen	37
6.1	Problemstellung	37
6.2	Modellierung von Unsicherheit	37
6.3	Umgebungsmodellierung mit Occupancy Grids	37
6.3.1	Satz von Bayes	37
6.3.2	Evidence Grids	37
6.3.3	Anwendung des Satzes von Bayes	38
6.4	Bayes-Filter Algorithmus	38
6.4.1	Algorithmus	38
6.5	Markov Lokalisierung	39
6.5.1	Algorithmus	39
6.6	Monte Carlo Lokalisierung	40
6.6.1	Grundsätzliches Vorgehen	40
6.6.2	Partikelmengen	41
6.7	Kalman-Filter	41
6.7.1	Definition	41
6.7.2	Vorgehen	41
6.7.3	Einschränkungen	42
6.8	Simultaneous Localization and Mapping	42
6.8.1	Landmarkenbasiertes SLAM Problem	42
6.8.2	Problemstellung	43
6.8.3	Funktionsweise	43
6.8.4	Hinzunahme neuer Landmarken	44

6.8.5	Aufbau eines SLAM-Graphen	44
6.8.6	Varianten von SLAM	45
6.8.7	Bayesian Netzwerk für landmarkenbasiertes SLAM	45
7	Schwarmrobotik und Evolutionäre Robotik	47
7.1	Schwärme und deren Verhalten in der Natur	47
7.1.1	Computersimulation von Schwärmen - Algorithmus von Craig Reynolds	47
7.2	Schwarmintelligenz	47
7.3	Multi Robot Systems	48
7.4	Ameisenalgorithmen	48
7.4.1	Optimaler Weg bei futtersbeschaffenden Ameisen	48
7.4.2	Ant Colony Optimization Algorithm	49
7.4.3	Traveling Salesman Problem	50
8	Locomotion	51
8.1	Fortbewegungsarten	51
8.2	Laufroboter	51
8.2.1	Vorteile von Laufrobotern	51
8.2.2	Nachteile von Laufrobotern	51
8.2.3	Freiheitsgrade für Roboterbeine	51
8.2.4	Freiheitsgrade für Zweibeiner	52
8.2.5	Laufverhalten	52
8.2.6	Statisch stabiles Gehen	52
8.2.7	Zero Moment Point und Pseudo-Dynamisches Gehen	53
8.2.8	Steuerungssoftware	53
8.3	Radroboter	54
8.3.1	Stabilität von Radrobotern	54
8.4	Kinematik mobiler Radroboter	54
8.4.1	Holonomische Bewegung	54
8.4.2	Kinematik und Positionsveränderung für Zweiradandtrieb	55
8.4.3	Fortbewegung bei Omnidirektionalem Antrieb	55
9	Industrierobotik	56
9.1	Industrieroboter	56
9.1.1	Einsatz	56
9.1.2	Trends	56
9.2	Bauform und Komponenten	56
9.3	Freiheitsgrade	57
9.3.1	Definition	57
9.4	Bewegungssachsen	57
9.4.1	Rotatorisch	57
9.4.2	Translatorisch	57
9.4.3	Rotationsgelenk	58
9.4.4	Torsionsgelenk	58
9.4.5	Revolvergelenk	58
9.4.6	Kugelgelenk	59
9.5	Arbeitsraum	59
9.6	Grundtypen von Industrierobotern	59
9.6.1	Portalroboter	59
9.6.2	Horizontal-Knickarmroboter	59
9.6.3	Vertikal-Knickarmroboter	60

Contents

9.6.4	Parallele Roboter	60
9.6.5	Leichtbauroboter	60
9.7	Effektoren	60
9.7.1	Endeffektoren	60
9.7.2	Greifersysteme	61
9.7.3	Greifplanung	61
9.7.4	Greifprinzipien	61
9.8	Antrieb	62
9.8.1	Antriebsarten	62
9.9	Sensor	62
9.9.1	Interne Sensoren	62
9.9.2	Externe Sensoren	62
9.10	Kinematik	62
9.10.1	Kinematikmodul	63
9.10.2	Steuerung und Regelung von Industrierobotern	63
9.10.3	Punkt-zu-Punkt Bewegung	63
9.10.4	Überschleifen	64
9.10.5	Programmierung	64
9.11	Kinetic - Berechnungen	65
9.11.1	Rotationsmatrizen	65
9.11.2	Vorwärts Transformation	67
9.11.3	Modifizierte Denavit-Hartenbert Transformation	67
9.11.4	Inverses kinematisches Problem	67
9.11.5	Bestimmung der Achswinkel anhand der Position und Orientierung des Endeffektors	67
9.11.6	Koordinatensysteme	69
9.12	Safety	69
9.13	Roboterprogrammierungen	69
9.13.1	Programmierung in RAPID	69
9.13.2	Syntax	70
9.13.3	Bewegungsfunktionen	72
9.13.4	World Zones	73
9.13.5	Bildschirmein- und -ausgabe	73
9.13.6	Zeitmessung	74
9.13.7	Bewegung relativ	74
10	Lego Mindstorms Code	75
10.1	SubSumptionMain	75
10.2	Arbitrator	75
10.3	Wishes	76
10.4	Behaviors	76
10.5	Effectors	78
10.6	Sensors	80

1 Geschichte

1.1 Industrieroboter

Nach Definition der VDI-Richtlinie 2860 sind Industrieroboter universell einsetzbare Bewegungsautomaten mit mehreren Achsen, deren Bewegungen hinsichtlich Bewegungsfolge und Wegen bzw. Winkel frei programmierbar und sensorgeführt sind.

- Zeichnen sich durch **Schnelligkeit**, **Genauigkeit**, **Robustheit** und eine hohe **Traglast** aus.
- Einsatzgebiete: Schweißen, Kleben, Schneide, Lackieren

Zunehmend **kollaborative** Roboter, Cobots:

- Industrieroboter, die mit Menschen gemeinsam arbeiten
- Nicht mehr durch Schutzeinrichtungen im Produktionsprozess von Menschen getrennt
- Nimmt Menschen wahr, verursacht keine Verletzungen

1.2 Serviceroboter

1.2.1 Definition

- Ein **Serviceroboter** ist eine **frei programmierbare Bewegungseinrichtung**, die **teil- oder vollautomatisch** Dienstleistungen verrichtet.
- **Dienstleistungen** sind dabei Tätigkeiten, die nicht der direkten industriellen ERzeugung von Sachgütern, sondern der Verrichtung von **Leistungen für Menschen und Einrichtungen** dienen.
- Einteilung in zwei Klassen

1.2.2 Klassen

- Roboter, die für professionellen Einsatzbereich: **Rettung**, **Landwirtschaft**, **Medizin**
- Roboter für den Privaten gebrauch: **Staubsauger**, **Rasenmäher**, **Pfleger**

2 Software-Architekturen für mobile Robotersysteme

2.1 Probleme und Anforderungen

2.1.1 Definition mobile Roboter

‘Unter einem Roboter verstehen wir eine frei programmierbare Maschine, die auf Basis von Umgebungssensordaten in geschlossener Regelung in Umgebungen agiert, die zur Zeit der Programmierung nicht genau bekannt und/oder dynamisch und oder nicht vollständig erfassbar sind.’ ⇒ **Joachim Herzberg, Mobile Roboter**

2.1.2 Umgebung mobiler Roboter

Bei **mobilen Robotern** ist die Umgebung im Detail **nicht bekannt und generell nicht kontrollierbar**

- Alle Aktionen sind von der aktuellen Umgebung abhängig
- Details sind erst zum Zeitpunkt der Ausführung der Aktionen bekannt
- Mobile Roboter müssen in einer geschlossenen Regelung
 - die Umgebung mit Sensoren erfassen
 - die Daten auswerten
 - Aktionen daraus planen
 - Aktionen mittels Koordination der Aktuatoren umsetzen

2.1.3 Roboterkontroll-Architekturen

Herausforderungen

- Robotersystem besteht aus den Gebieten **Wahrnehmung, Planung und Handlung**
- Herausforderungen an eine Roboterkontroll-Architektur, sie muss:
 - Sensorwerte erfassen und auswerten
 - Pfade planen
 - Hindernisse vermeiden
 - Komplexe Algorithmen in langen Zeitzyklen ausführen

Probleme bei der Software-Erstellung zur Roboterkontrolle

- Roboter sind eingebettete Systeme, die in geschlossener Regelung laufen und die Sensorströme in **Echtzeit verarbeiten** müssen
- Unterschiedliche Aufgaben ⇒ Unterschiedliche Zeitzyklen
- Unterschiedlicher Zeitskalen ⇒ kein standardisierter Kontroll- oder Datenfluss den die Architektur abbilden könnten
- Für etliche algorithmische Teilprobleme sind **keine effizienten Verfahren** bekannt
- **Prozessorkapazität ist begrenzt**

2.1.4 Anforderungen an das Kontrollsysteem eines autonomen Roboters

Robustheit

- Die Umgebung des Systems kann sich ständig ändern
- Auf eine Umgebungsänderung sollte der Roboter sinnvoll reagieren und nicht verwirrt stehen bleiben.
- Verwendete Modelle der Umgebung sind ungenau.

Unterschiedliche Ziele

- Der Roboter verfolgt zu einem Zeitpunkt eventuell Ziele, die im Konflikt zueinander stehen.
- **Beispiel:** der Roboter soll ein bestimmtes Ziel ansteuern, dabei aber Hindernissen ausweichen.

Sensorwerte von mehreren Sensoren

- Sensordaten können verrauscht sein
- Sensoren können fehlerhafte oder inkonsistente Messwerte liefern, weil der Sensor z.B. außerhalb seines Bereichs misst für den er zuständig ist und dies nicht überprüfen kann.

Erweiterbarkeit

- Wenn der Roboter neue Sensoren erhält, sollte dies leicht in das Programm integriert werden können.

2.2 Mögliche Modelle

2.2.1 Klassisches Modell - der funktionale Ansatz

Das **klassische Modell** wird auch als hierarchisches Modell oder funktionales Modell bezeichnet. Ist ein Top-Down Ansatz, besteht aus drei Abstraktionsebenen

- Die unterste Ebene: **Pilot**
- Mittlere Eben: **Navigator**
- Oberste Ebene: **Planer**

Sense-Think-Act-Cycle oder **SMPA** (Sense - Model - Plan - Act).

- Sensordaten, die vom Fahrzeug geliefert werden, werden in den zwei unteren Ebenen vorverarbeitet.
- Konstruktion oder Aktualisierung eines Weltmodells
- **Planer** ist die Basis aller Entscheidungen basieren auf dem zugrundeliegenden Weltmodell
- Tatsächliche Fahrbefehle werden durch unterste Ebene ausgeführt

Zyklus wird ständig wiederholt \Rightarrow wenn alle Ebenen richtig funktionieren resultiert daraus ein intelligentes Verhalten und die Erfüllung der Aufgabe.

Nachteile

- **Sequentieller Ansatz, lange Kontrollzykluszeit**
- **Gesamtsystem anfällig**, \Rightarrow fällt ein Modul scheitert das Gesamtsystem
- Die Repräsentation der Umgebung muss alle notwendigen Informationen enthalten, damit ein Plan entwickelt werden kann. Planer hat nur Zugriff auf das Weltmodell \Rightarrow während Planer Aktionen ausarbeitet, könnte sich die Umwelt schon wieder geändert haben.

2.2.2 Verhaltensbasiertes Modell

Grundlegender Gedanke Intelligentes Verhalten wird nicht durch komplexe, monolithische Kontrollstrukturen erzeugt, sondern durch das Zusammenführen der richtigen einfachen Verhalten und deren Interaktion.

Definition

- Engere Verbindung zwischen **Wahrnehmung** und **Aktion**
- Jede **Roboterfunktionalität** wird in einem **Behavior** gekapselt
- Alle **Behaviors** werden **parallel ausgeführt**
- Jedes Behavior Modul operiert unabhängig von den anderen
- Alle Behaviors können auf alle Fahrzeugsensoren zugreifen und gewissermaßen die Aktuatoren ansteuern.

2.2.3 Hybrider Ansatz

- Nutzt die Vorteile der **Subsumption Architektur** und der **SMPA-Architektur**
- Der verhaltensbasierte Anteil ist nicht geeignet, auf längere Sicht zielgerichtet Aktionen zu koordinieren \Rightarrow SMPA-Anteil

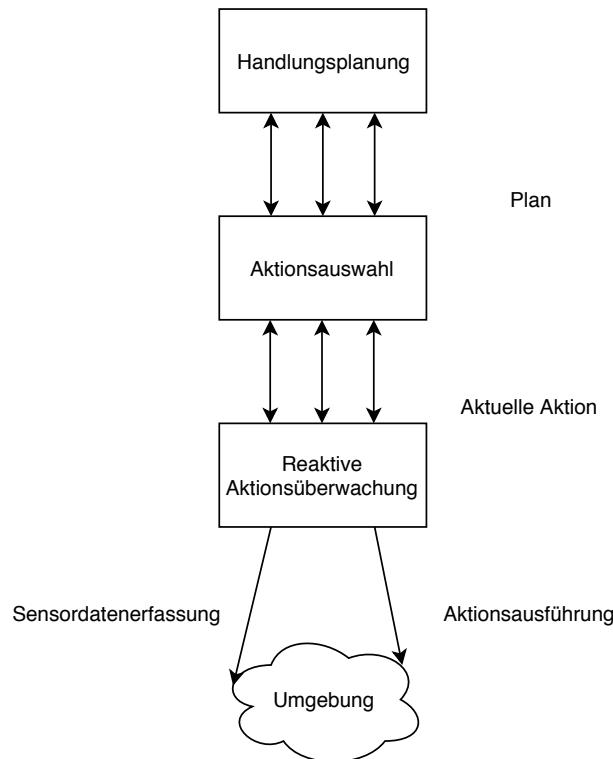


Figure 2.1: Schema der Hybridmodell Schichten

- Die **Handlungsplanung** arbeitet auf hoher, strategischer Stufe in langen Zeitzyklen
- die **mittlere Kontrollebene** hat die taktische Aufgabe, die jeweils **nächste Aktion aus dem Plan auszusuchen**, zu instanzieren und auf die Ebene der Verhaltensbausteine zu zerlegen. Des Weiteren muss die Rückmeldung von der Aktionsüberwachung interpretieren und entscheiden, ob eine Aktion erfolgreich abgeschlossen ist. \Rightarrow entscheiden, ob die Handlungsplanung einen anderen Plan erstellen muss
- Die **reaktive Aktionsüberwachung** enthält die Verhaltensbausteine auf operativer Ebene, die in schellen Zeitzyklen die physische Roboteraktion anstoßen und überwachen

Kritik

- Mittlere Komponente benötigt den größten konzeptuellen und programmiertechnischen Aufwand
- Das mittlere Teilproblem ist deutlich komplexer als die beiden anderen

2.2.4 Probabilistische Robotik

- Probabilistische Robotik berücksichtigt die **Unsicherheit der Wahrnehmung und der Aktionen**
- **Schlüsselidee:** Information in Form von Wahrscheinlichkeitsdichten repräsentieren

- Eine Lokalisierung der Roboter wird unter Verwendung von Wahrscheinlichkeitstheorie oder einer Wahrscheinlichkeitsverteilung eine Aussage über die Umgebung treffen
- **Probabilistische Wahrnehmung:** wenn man Sensorwerte schätzen kann, dann kann man mit Wahrscheinlichkeitstheorie eine Aussage über die Umgebung treffen
- **Probabilistisches Handeln:** aufgrund der Unsicherheit über die Umgebung ist auch das Handeln mit Unsicherheit behaftet. Mit probabilistischen Ansätzen besteht die Möglichkeit Entscheidungen trotz Unsicherheit zu treffen

Vorteil probabilistische Verfahren können auch mit weniger präzisen Umgebungsmodellen angewandt werden.

Nachteil weniger effizient wegen komplexer Berechnungen, Approximation erforderlich

2.2.5 Subsumption-Architektur in Bezug auf die Anforderungen des Robotersteuerungssystems

Robustheit

- Wenn einige Steuerungsmodule ausfallen, arbeiten bei der Subsumption-Architektur die restlichen Schichten einwandfrei ⇒ **eingeschränktes, aber sinnvolles Verhalten möglich**

Unterschiedliche Ziele

- Mehrere Teilsituationen können verschiedene Verhaltenselemente sinnvoll machen, die sich widersprechen können.
- Die Wichtigkeit einer Handlung hängt vom Kontext ab, d.h. höhere Ziele können niedrigere Ziele ersetzen.
- Alle zu einem Zeitpunkt möglichen Verhaltenselemente werden parallel bearbeitet.
- Das **resultierende Verhalten wird in Abhängigkeit von Umwelteinflüssen dynamisch bestimmt**
- Das Gesamtergebnis hängt nicht von einer übergeordneten Instanz ab

Sensorwerte von mehreren Sensoren

- Der Roboter muss auch bei inkonsistenten Informationen eine Entscheidung fällen
- Die Subsumption-Architektur sieht keine zentrale Verarbeitung und Speichung der Umwelt-daten vor
- Jedes Modul reagiert nur auf die Daten einzelner Sensoren, es muss **kein konsistentes Abbild der Umwelt erschaffen werden**

Erweiterbarkeit Das bestehende Verhalten kann jederzeit durch Hinzufügen weiterer Schichten um komplexe Funktionen erweitert werden

2.3 ROS - Robot Operating System

2.3.1 Entwicklung

- Das Architektschema für Roboterkontrollsoftware gibt es nicht ⇒ deshalb heute auch Unterstützung der Roboter-Softwareentwicklung durch Middleware wie ROS
- Zweck: soll die Entwicklung von Software für Roboter vereinfachen und wiederkehrende Aufgaben standardisieren
- Standard für Roboterkontrollsoftware

2.3.2 Design Prinzipien

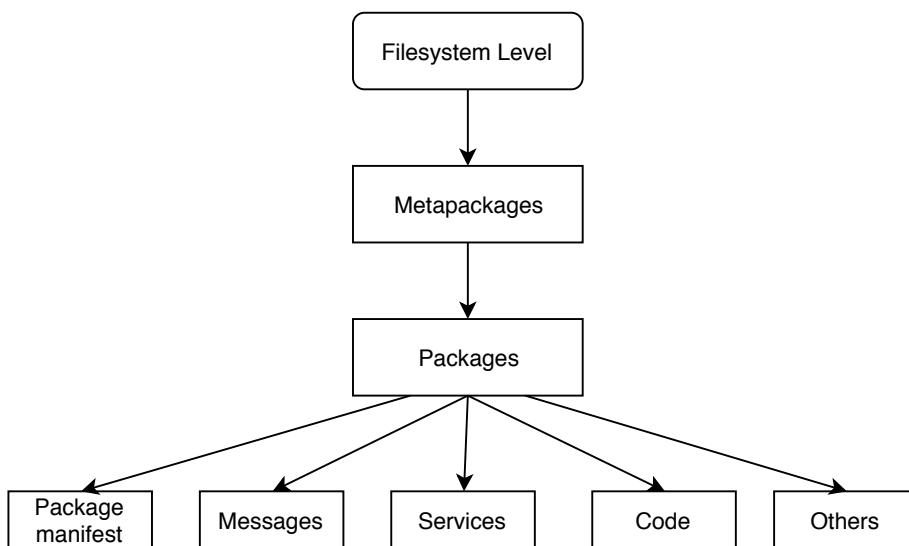


Figure 2.2: Filesystem, das ROS zugrunde liegt

- Ein Package beinhaltet die ROS Prozesse, welche auch Nodes genannt werden
- Komplexe Prozesse werden durch Netzwerke von Nodes bewerkstelligt
- Roboterkontrollprogramm besteht aus vielen Prozessen, die potentiell über mehrere Rechner verteilt sein können
- Wichtigster Knoten ist der Master ⇒ Abwicklung der internen Kommunikation
- Andere Knoten können nur starten, wenn ein Master existiert
- Nodes müssen sich beim Master anmelden
- Funktionalität (Kommandos, Ausführung v. Algorithmen) wird in eigenen Nodes realisiert
- Nodes sind in verschiedene Prozesse getrennt ⇒ fehlerhafte Knoten hat i.d. Regel wenig Auswirkungen auf die anderen
- Knoten werden über Publish-Subscribe verknüpft
- **Asynchrone Nachrichten** werden durch Topics ausgetauscht
- **Synchrone Nachrichten** werden durch Services ausgetauscht

2.3.3 Publish - Subscribe

Nodes sind Software-Module, die die Verarbeitung durchführen. Sie kommunizieren über Topics miteinander und tauschen dabei Nachrichten.

Kommunikation Die Kommunikation basiert auf einem **Publish Subscribe Pattern**

- Wenn Daten weitergegeben werden sollen, wird ein Publisher erzeugt
- Publisher registriert sich beim Master und gibt Topics an
- Daten können in anderen Knoten abgerufen werden – dazu wird ein (oder mehrere) Subscriber angelegt
- Subscriber frägt beim Master bezüglich gewünschten Topics an
- Daten werden über TCP/IP Sockets übertragen

Topics

- Themen, zu denen die Nodes Messages versenden
- Topics sind einfach Strings
- Verschiedene Nodes können zu einem bestimmten Topic Nachrichten versenden
- Ein Node kann sich prinzipiell zu mehreren Topics einschreiben und mehrere Topics publizieren

Services

- Nachteil von Publish Subscribe wird durch Services geschlossen
- Sind eine weitere Art, wie Nodes kommunizieren können
- Synchroner Nachrichtenaustausch mithilfe von Requests, welche von anderen Nodes mit einer Response beantwortet werden
- Ein Knoten registriert eine Aktion (Service), namentlich beim Master
- Ein Service Caller kann die Ausführung eines Services anstoßen, sobald dieser verfügbar ist
- geeignet für RMI oder einmalige Anfragen

Messages

- werden von Nodes bei der Kommunikation
- Messages sind streng typisierte, möglicherweise verschachtelte Datenstrukturen, die aus den primitiven Typen int, float, bool bestehen
- Eine Message kann andere Messages oder Felder von Messages enthalten

2.3.4 Parameter Server und Konfigurationsdateien

- Der ParameterServer auf dem Master Knoten enthält eine Art Wörterbuch für Werte.
- Alle Ressourcen wie Knoten, Nachrichten oder Parameter existieren in einer hierarchischen Namensstruktur.

2.3 ROS - Robot Operating System

- Speichert z.B. Konfigurationsdateien

3 Lokalisation autonomer mobiler Robotersysteme

3.1 Abgrenzung: Lokalisation - Mapping - SLAM - Navigation

- **Lokalisation** Ermitteln der aktuellen Position des Roboters.
- **Kartenerstellung, Mapping oder Umgebungsmodellierung** hilft bei Entscheidung
Kartenerstellung bedeutet die Auswertung der vom Roboter mittels Sensoren erfassten Daten der Umgebung mit dem Ziel, ein Umgebungsmodell zu erzeugen oder zu vervollständigen.
- **Großes Problem** Mapping
- **Pfadplanung oder Navigation** beantwortet die Frage **Wie gelange ich dorthin?** Bewegungsplanung oder Pfadplanung bedeutet die Berechnung der Fahrroute und der daraus abgeleiteten Bahn vom aktuellen Punkt zum Zielpunkt
- Man unterscheidet zwischen Navigation in unbekannter und in bekannter Umgebung.
- Selbstlokalisierung und Kartenerstellung bedingen sich gegenseitig.

3.2 Varianten der Selbstlokalisierung

Lokale Selbstlokalisierung (position tracking)

- Die Startposition des Roboters ist ungefähr bekannt.
- Es handelt sich um **relative** Selbstlokalisierung
- Sobald sich der Roboter bewegt, muss aufgrund neuer Sensordaten die Position neu berechnet werden.
- Bezugspunkt ist der Startpunkt.
- **Methoden** Odometrie und Trägheitsnavigation

Globale Selbstlokalisierung

- Die Startposition ist unbekannt.
- Es handelt sich um **absolute Positionierung**
- An welcher Position der Roboter befindet, entscheidet er durch Auswerten seiner Sensordaten und durch erkennen von **signifikanten** Umgebungsmerkmalen
- **Mögliche Methode:** Triangulation

Kidnapped Robot Problem

- Die Position des Roboters ist anfangs bekannt
- Der Roboter wird willkürlich mit temporär deaktivierten Sensoren an eine beliebige andere Position versetzt, ohne darüber informiert zu werden.
- Auch dann muss das Verfahren robust die Position wiederfinden, zunächst muss der Roboter dies erkennen und sich dann relokalisieren
- Es muss eine erneute globale Lokalisierung durchgeführt werden

3.3 Relative Lokalisierung versus Absolute Lokalisierung

Relative Lokalisierung

- auch: **lokale, inkrementelle Lokalisierung** oder 'tracking'
- Relativ zu einer Startpose wird sukzessiv die Änderung der Pose an diskreten, aufeinanderfolgenden Zeitpunkten ermittelt und integriert

Absolute Lokalisierung

- auch als **globale Lokalisierung** bezeichnet
- Die Pose wird in Bezug auf ein externes Bezugssystem ermittelt, z.B. einer Karte oder einem globalen Koordinatensystem

Ziel Bestimme oder schätze die Position und Orientierung des Roboters in seiner Umgebung basieren auf

- der Eigenbewegung
- durch Messungen der relativen Position zu unterscheidbaren Objekten in der Umgebung in Roboterkoordinaten (Ultraschall, Laser, Kamera)

3.4 Transformation von Koordinatensystemen lokale <-> globale

Kinematik Die Kinematik ist die Lehre der Beschreibung von Bewegungen von Punkten im Raum. Dabei werden die Größen Weg, Geschwindigkeit und Beschleunigung betrachtet. Die Kinematik ist ein Teilgebiet der Mechanik.

Kinematische Robotermodell

- kreisförmiger Roboter
- Zweiradantrieb
- Bewegung in der Ebene

Lokales Koordinatensystem

- mit dem Roboter verbunden
- Ursprung in der Mitte der Antriebsachse
- x-Achse zeigt in Richtung des Roboterfrontteils

Festlegung der Roboterposition im globalen Koordinatensystem

- durch die Koordinaten $M(x_M, y_M)$ im globalen Koordinatensystem
- durch den Winkel θ zwischen der lokalen x-Achse und der globalen x-Achse
- Pose p gegeben durch: $p = (x_M, y_M, \theta)^T$

Transformation von lokalen in globale Koordinaten

- Koordinaten von P im lokalen Koordinatensystem: $p_l = (x_l, y_l)^T$
- Koordinaten von P im globalen Koordinatensystem: $p_g = (x_g, y_g)^T$
- Transformation von p_l nach p_g ($m = (x_M, y_M)^T$): $p_g = R(\theta)p_l$

Transformation von globalen in lokale Koordinaten

- Transformation von globalen in lokale Koordinaten $p_l = R(\theta)^{-1}(p_g - m) = R(-\theta)(p_g - m)$

3.5 Karten für statistische und dynamische Umgebungen

- Generell gilt: **Karten** sollen eine explizite Repräsentation des Raumes sein.
- Die Karten sind auf die Sensorik des Roboters zugeschnitten
- Die Karten sind nicht vorrangig für den menschlichen Betrachter bestimmt, sondern der Roboter soll sie effizient nutzen können.

Statische Umgebungen

- basierend auf der Annahme, dass sich zwar der Zustand des Roboters innerhalb der Umgebung, nicht jedoch die Umgebung selbst ändert.
- Karte spiegelt die wirkliche Umgebung wieder.

Dynamische Umgebungen

- Objekte können ihre Lage oder ihren Zustand ändern
- In der Karte registrierte Objekte können verschwinden, nicht registrierte Objekte auftauchen
- ‘Lernende’ Karten sind ein fundamentales Problem in der mobilen Robotik

3.5.1 Mapping Methoden

Weltzentriert

- Die Pose aller Objekte einschließlich des Roboters werden in der Umgebung in Bezug auf ein festes Koordinatensystem repräsentiert.
- **Indoor:** Ursprung kann eine Zimmerecke sein
- **Outdoor:** Benötigung eines globalen Koordinatensystems, wie z.B. die Längen- oder Breitengrade, i.d.R. nutzen von **WGRS**(World Geographic Reference System)

Roboterzentriert gebraucht um bspw. Kollisionen zu vermeiden

- Durch Koordinaten-Transformation kann zwischen den verschiedenen Referenz-Frames konvertiert werden

3.5.2 Arten von Modellen

- Die wichtigste Form von Umgebungsmodellen für mobile Roboter sind Umgebungskarten.
- Die folgende Ausführungen beziehen sich auf geeignet Karten für **mobile, autonome Landfahrzeuge**

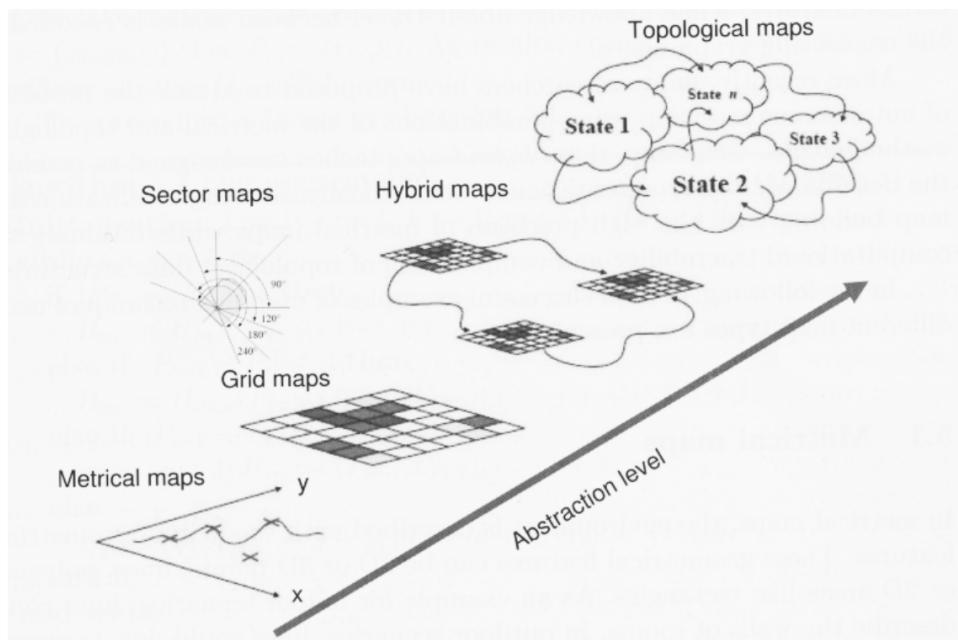


Figure 3.1

Arten von Umgebungsmodellen

- **kontinuierliche metrische Karten**, zweidimensional oder dreidimensional:
 - jedes Objekt wird assoziiert mit Koordinaten
- **diskrete metrische Karten, Grid Maps**, zweidimensional oder dreidimensional:
 - der Raum wird gleichmäßig oder ungleichmäßig aufgeteilt; Objekte werden mit Positionen innerhalb des Gitters assoziiert

- **Hybrid Maps**
- **Topologische Modelle** nur zweidimensional
 - im Vordergrund steht die Beziehung der Objekte zueinander

3.5.3 Kontinuierliche Metrische Karten

- Metrische Lokalisierung, beruht auf Ultraschall oder Laserscannern
- Exakte Beschreibung der Umgebung mit 2D oder 3D Modellen möglich
- Die Positionen von Objekten der Umgebung werden durch ein Koordinatensystem repräsentiert.
- **Vorteil:** detailliertes Bild der Umgebung
- **Nachteil:** große, unstrukturierte Datenmengen erschweren die Pfadplanung

3.5.4 Grid Maps - Rasterkarten

- Die Umwelt des Roboters wird in ein gleichmäßiges Raster oder Grid zerlegt.
- Jede Zelle enthält den Belegtheitsgrad der Zelle ⇒ zeigt an ob zelle mit einem Hindernis belegt ist oder nicht
- Verschiedene Modelle verwenden unterschiedliche Werte
 - zwei Werten
 - freie Zellen, belegte Zellen und Zellen mit Mischbelegung
 - Prozentsatz der Belegungswahrscheinlichkeit
- Notwendige Informatinoen sind z.B.:
 - x, y als Koordinaten (Zeile, Spalte) einer Zelle
 - Sensordaten des Roboters
 - ein boolscher Wert für den Zustand der Zelle
- Die Werte in den Zellen sind unabhängig voneinander
- Eine Steigerung der Messgenauigkeit der Sensoren führt dazu, dass die Rasterelemente immer kleiner werden
- Für eine kompakte Notation können Grid Maps im 2-dimensionalen Raum mit Quadtrees im 3-dimensionalen mit Octrees gespeichert werden

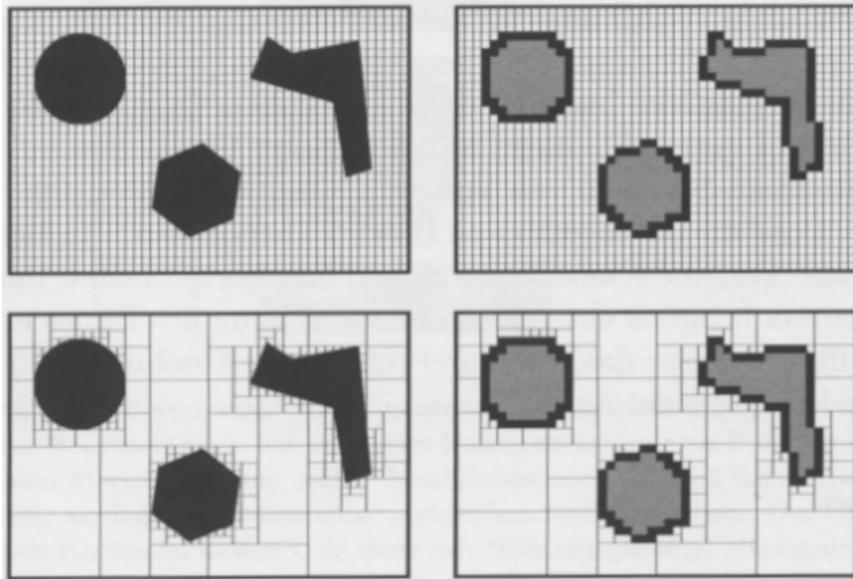


Figure 3.2

Gleichmäßige Gitterstruktur vs. Adaptiver Gitterstruktur

- links oben zeigt die Objekte in einer gleichmäßigen Gitterstruktur
- rechts oben zeigt die zugehörige Repräsentation über Belegtheiten der Zellen
- speicherintensiv bei gleichmäßiger Unterteilung des Raums \Rightarrow adaptive Unterteilung des Raums und Speicherung in Quadtrees oder bei 3. Dimension Octtrees

3.5.5 Adaptive Unterteilung

- Ausgangszustand: Rechteck mit Hindernissen
- Fläche wird unterteilt in 4 Rechtecke gleicher Größe
- Jedes Rechteck wird rekursiv wieder in 4 Rechtecke unterteilt \Rightarrow Quadtree
- Attributierung der Knoten:

Frei: Rechteck enthält keinen Teil eines Hindernisses

Belegt: Rechteck ist vollständig von Hindernis belegt

Gemischt: Rechteck enthält Punkte, die zu einem Hindernis gehören, sowie solche die es nicht tun

- Nur gemischte Knoten werden weiter unterteilt

Vorteile schnell und leicht feststellbar, ob Punkt in einem Hindernis liegt

Nachteile

- Konturen der Objekte und der Freiraum zwischen ihnen wird unpräzise repräsentiert
- Um die Datenfülle zu reduzieren, wird das Raster zu grob gewählt und dadurch ein möglicher Weg durch Mischpixel versperrt

Weiterer Verwendungszweck Neben der reinen Lokalisierung können die Karten auch dazu verwendet werden eine Fahrspur (Trakektorie) zu berechnen.

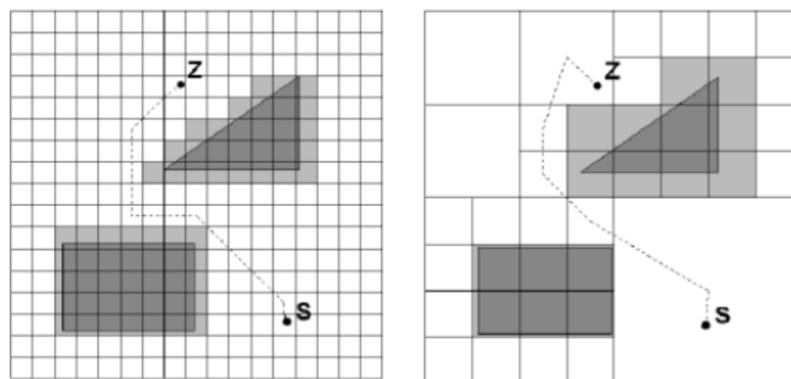


Figure 3.3

3.5.6 Weitere Beispiele für Umgebungskarten

- Laserscan Karten
- Bildbasierte Karten

3.5.7 Topologische Karten

Bedingt geeignet zur Lokalisation, Haupteinsatzgebiet ist die Pfadplanung.

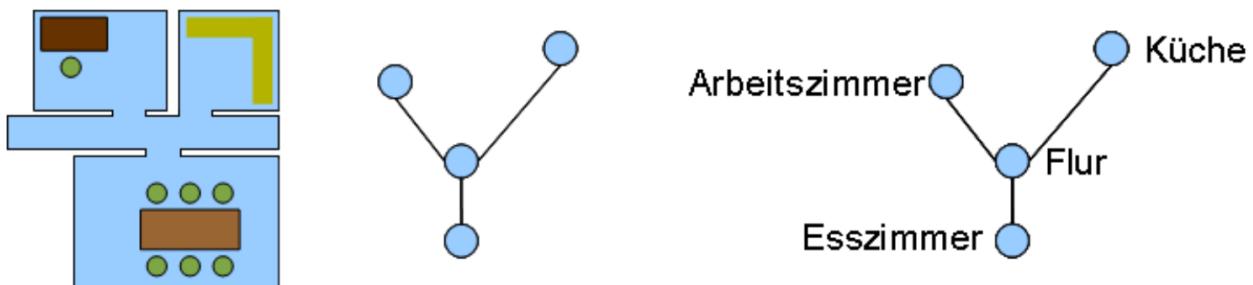


Figure 3.4

- Modelle bilden einen **Graphen**
- **Knoten** entsprechen Orten oder Bereichen der Umgebung
- Beziehungen zwischen den Orten werden durch **Kanten** modelliert.
- Zwei Knoten sind durch eine Kante verbunden, wenn sie unmittelbar voneinander erreichbar sind.
- **Gewichte**: Maß für die Länge der Wege
- Ist die Länge der jeweiligen Wegstücke bekannt, lässt sich der kürzeste Weg finden.

Vorteile

- Kompaktheit

- Gute Skalierbarkeit für welträumige Umgebungen.
- Es gibt viele schnelle Algorithmen auf Graphen, die gut zur Pfadplanung eingesetzt werden können

Nachteil Relevante Umgebungsmerkmale werden verdeckt. Landmarken werden schwerer erkannt.

3.5.8 Hybrid Maps

- Kombinieren metrische und topologische Ansätze
- Ermöglichen Lokalisation und Kantenerstellung mit hoher Präzision
- Erhalten die Kompaktheit der topologischen Ansätze

Abstraktions-basierter Ansatz

- Basis: konstruieren einer metrischen Karte der Umgebung
- ⇒ aufbau einer kompakten topologischen Repräsentation
- **Vorteil** Effizient Planung eines Pfads zu einem gegebenen Ziel aufgrund der Abstraktion.
- Die zugrunde liegende metrische Karte wird für Relokalisation und Hindernisvermeidung benötigt.

3.6 Passive und Aktive Selbstlokalisierung

3.6.1 Passive Verfahren

- bestimmen oder schätzen die Roboterposition mittels aktueller Sensorinformationen
- beeinflussen **nicht** die Bewegung und Orientierung des Roboters
- Lokalisierungsmodul beobachtet nur die Roboteroperationen
- Roboter bewegt sich zufällig hin und her bzw. führt die zu erledigende Aufgabe durch

3.6.2 Aktive Verfahren

- besitzen vollständige oder teilweise Kontrolle über die Bewegungen des Roboters und Ausrichtung der Sensoren
- fährt gezielt bestimmte Orte an um Mehrdeutigkeiten zwischen mehreren Orten aufzulösen

3.7 Landmarken

3.7.1 Definition

- Als Landmarken werden **eindeutig identifizierbare Charakteristiken der Umwelt** bezeichnet, die von entsprechenden Sensoren erkannt werden können.

Landmarke

- ihre Position im Weltmodell ist bekannt
- sichtbar von unterschiedlichen Positionen aus
- erkennbar unter verschiedenen Belichtungen und Blickwinkeln
- relative Position bestimmbar
- stationär, oder dem Navigationsmechanismus muss die Bewegung bekannt sein

Vorteil Navigation erfolgt mit der Umwelt selbst und nicht mittels erachteter Daten

3.7.2 Natürliche Landmarken

- Werden nicht zum Zweck der Positionsbestimmung aufgestellt, können aber dafür verwendet werden
- Grundsätzlich Passiv

3.7.3 Künstliche Landmarken

- Markante Objekte, eigens zum Zweck der Positionsbestimmung in der Umgebung installiert.

4 Fortbewegung, Lokalisierungsalgorithmen

4.1 Relative Lokalisierung

4.1.1 Dead Reckoning

- **Koppelnavigation oder Dead Reckoning** ursprünglich in der Nautik verwendet
- Mathematisches Verfahren - **Vorwärtsskinematik** - zur Positionsbestimmung
- Ausgehend von einer Startposition ist es dem Navigator möglich, seine **aktuelle Position zu berechnen** aufgrund der **zurückliegenden bekannten Kurs- und Geschwindigkeitswerte**

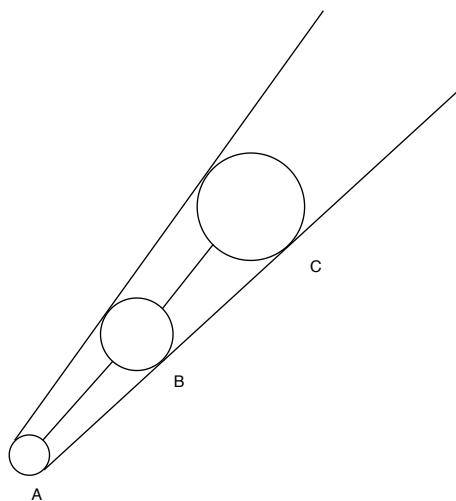


Figure 4.1

- A sei ein gegebener Ausgangspunkt
- Radius wird angegeben der zur Abweichung proportional ist, bsp. hier 0,5m.
- Der Radius spiegelt die mit der Zeit kumulierte Ungenauigkeit wieder
- Eine mögliche Roboterposition ist dann innerhalb des Sektors gegeben der durch die Linien eingegrenzt wird

Vorteile

- einfache Implementierung
- leichte Interpretation der Daten
- unkomplizierte Bedienung
- passable Kurzstreckengenauigkeit

Nachteile

- Startposition muss bekannt sein
- Genauigkeit nimmt mit zunehmender Länge der befahrenen Strecke drastisch ab

4.1.2 Odometrie

Odometrie ist die Wissenschaft der Positionsbestimmung eines Fahrzeugs durch die Beobachtung seiner Räder.

Grundlegendes verfahren

- Sensoren an Rädern messen Drehbewegung
- **Relative Positionsbestimmung:** Die **Bestimmung der Position** erfolgt ausgehend von einer bekannten Position durch Berechnung des zurückgelegten Weges und anhand von Daten über den Roboter selbst.
- Es wird Inkrementalgebern die Anzahl n der Radumdrehungen zwischen zwei Messpunkten gezählt. Aus dem bekannten Radumfang wird die wegendifferenz berechnet mit:

$$\Delta = \Pi \times d \times n$$
- **Ausrichtung** kann durch differentiale Odometrie erfolgen: es werden z.B. die unterschiedlichen Entfernung gemessen, die die linken und rechten Räder zurückgelegt haben.

Vorteile

- kostengünstig
- hohe Abtastraten
- passable Kurzzeitgenauigkeit

Fehlerquellen

- Fehlerhafte Messung des Raddurchmessers
- Raddurchmesser nicht gleich, Unrundheit des Radess

Fehlerberücksichtigung

- Die **Fehler** fließen in die Positions differenz ein, werden zur letzten bekannten Position hinzugefügt und **summieren sich mit jedem Messschritt**
- Fehlerellipse wächst mit zurückgelegtem Weg
- Odometrie als alleiniges verfahren nur für kurze Strecken geeignet
- Fehler lassen sich bei geringen Geschwindigkeiten und geringer Beschleunigung reduzieren

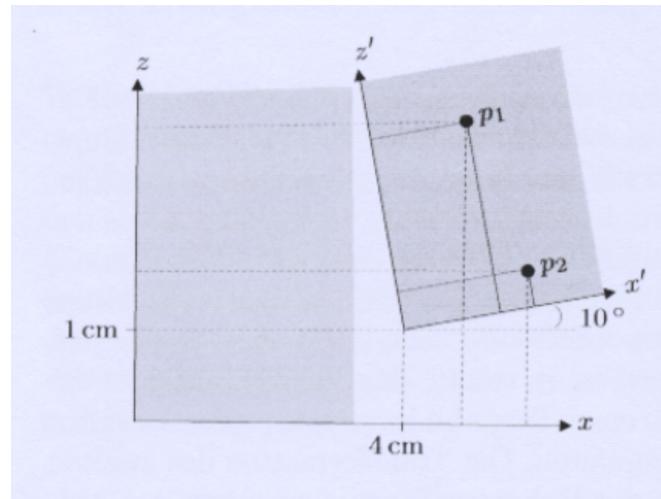


Figure 4.2

4.1.3 2D-Scanmatching

- Ausgangslage sind zwei Scans, ein Scan M (**Modell**) und ein zweiter Scan D (**Daten**)
- Es wird eine Transformation des einen Scans berechnet und zwar so, dass beide optimal überlagert werden
- Die Transformationen bestehen nur aus einer Rotation und einer Translation
- Die Überlagerung ist optimal, wenn Punkte, die in der realen Szene nahe beieinander liegen, auch in den registrierten Messdaten nahe beieinander liegen.
- **Ziel:** Fehlerfunktion minimieren \Rightarrow Abstände der Punkte des einen Scans zu ihren korrespondierenden Punkten des zweiten Scans
- Die Transformation des zweiten Scans entspricht dann der Bewegung des Roboters zwischen der Aufnahme der Daten; durch sukzessiven Vergleich kann damit die Bewegung des Roboters nachverfolgt werden

Iteratives Vorgehen

- **Annahme:** die korrespondierenden Punkte sind bekannt \Rightarrow eine **Transformation** kann berechnet werden, die diese Mengen aufeinander abbildet
- **Beispiel** zwei Scans mit einer Poseänderung des zweiten um $(4\text{cm}, 1\text{cm}, 10\text{deg})^T$ beide Scans sehen dieselben Raumpunkte p1 und p2
- Obige Annahme i.d.r. nicht erfüllt \Rightarrow **nicht eindeutig zu bestimmen, welche Punkte zwischen den beiden Scans korrespondieren**
- **Lösung:** **iteratives Vorgehen**, bei dem zunächst eine **Schätzung** der Punktpaarung stattfinden und die Pose des zweiten Scans unter dieser Paarung optimiert wird.
- Iterativ werden mit dem transformierten Scan neue Punktpaare berechnet, bis ein Abbruchkriterium erfüllt ist, d.h. bis sich die Transformation zwischen zwei Schritten nicht mehr signifikant ändert

Transformationsberechnung

- **Gesucht:** Mögliche Menge von Translationen und Rotationen, unter denen ein korrektes Matching möglich ist.
- $(t_x, t_z, \theta)^T$, die eine Translation um t_x in x-Richtung und t_z entlang der Z-Achse durchführt, sowie eine Rotation um den Winkel θ
- Der Scan M besteht aus einer Menge von Punkten $(m_i)_{i=1,2,\dots,N}$
- Der Scan D besteht aus einer Menge von Punkten $(D_i)_{i=1,2,\dots,N}$

Minimum der Funktion $E(\theta, t) = \sum_{i=1}^N \|p_i - (\mathbf{R}_\theta p'_i + \mathbf{t})\|^2$

Transformation zur minimierung der Fehlerfunktion E Folgende Transformation mit ggb. Parametern minimiert die Fehlerfunktion:

$$\theta = \arctan \left(\frac{S_{zx'} - S_{xz'}}{S_{xx'} + S_{zz'}} \right)$$

Hierbei ist

$$t_x = c_x - (c'_x \cos \theta - c'_z \sin \theta) \text{ und}$$

$$t_z = c_z - (c'_x \sin \theta + c'_z \cos \theta)$$

5 Navigation

5.1 Bekanntes vs. unbekanntes Terrain

Man unterscheidet zwischen Algorithmen für:

- **bekannte Umgebungen** (auch während der Fahrt ändert sich die Umgebung nicht)
- **unbekannte Umgebungen**
- Gebiet **vollständig bekannt** ⇒ Lösung der Suche mittels eines Graphen.
- **unvollständig oder gar nicht bekanntes Gelände** ⇒ Berechnungen erfolgen auf **lokalen Teilinformationen**

Neue Informationen ⇒ **inkrementelle Anpassung** über Sensoren

5.2 Navigation in unbekanntem Terrain

5.2.1 Konturverfolgung

- Eine **Freiraumfahrt** d.h. eine Fahrt durch ein Gelände, dessen Raum möglichst weit und frei von Hindernissen ist ⇒ nicht immer Zielführend
- Lösung: **Konturverfolgung**

Roboter wird nah an einem Objekt (Wand, Hinderniss) entlang bewegt

Es sollte möglichst ein gegebener Abstand d eingehalten werden

Regelung des Abstands d

IF (Distance to Wall > d)

THEN turn to wall

IF (Distance to Wall < d)

THEN turn away from wall

IF (Distance to Wall == d)

THEN Drive straight ahead

5.2.2 Sensorbasierte Planer - Navigation mit Hinderniskontakt

Bug1 Algorithmus

Fordert zwei bestimmte Verhalten:

- bewege dich auf einer geraden Linie
- folge einer Begrenzung in einem bestimmten Abstand

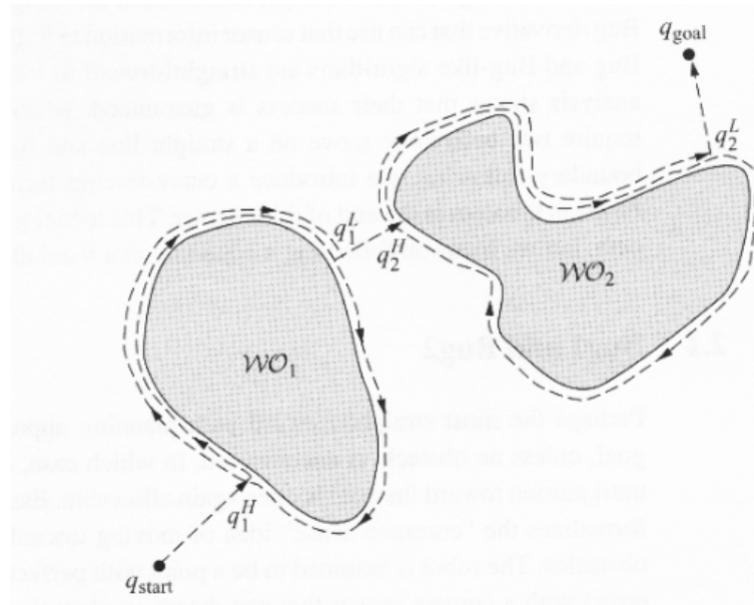


Figure 5.1: Bug1 Algorithmus Route

Voraussetzungen

- Roboter benötigt einen Sensor zur Erkennung eines 'Kontakts' mit einem Hindernis
- Roboter kann die Distanz zwischen zwei Punkten x und y messen
- der Arbeitsraum ist begrenzt

Algorithmus

- Roboter folgt Linie zum Ziel bis er beim Punkt q_1^H auf ein Hinderniss trifft
- Roboter umfährt das Hindernis bis er erneut beim gleichen Punkt auf das Hindernis trifft
- Roboter bestimmt den nächsten Punkt (Leavepoint) q_1^L vom Hindernis zum Ziel und fährt diesen Punkt an
- Von diesem Punkt fährt der Roboter geradewegs zum Ziel bis er das Ziel erreicht oder erneut auf ein Hindernis trifft

Exception Schneidet die Linie von q_1^L zum Ziel das **aktuelle Hindernis** gibt es keine Lösung

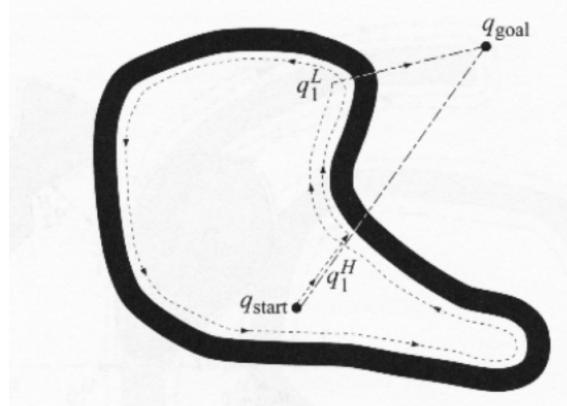


Figure 5.2

Bug2-Algorithmus

Idee:

- Linie vom Start S zum Ziel T konstruieren
- bewegt sich möglichst auf dieser Linie
- Roboter trifft auf Objekt \Rightarrow umfahren in einer bestimmten Richtung bsp. immer rechts herum
- Hindernis wird solange umfahren, bis der Roboter wieder auf einen Punkt auf der Linie ST trifft, der näher zum Ziel liegt als der ursprüngliche Kontaktpunkt mit dem Hindernis

Algorithmus:

-
1. Auf Geraden ST in Richtung T fahren bis:
 - a.) Ziel erreicht wird \rightarrow END
 - b.) Hindernis getroffen wird \rightarrow Schnittpunkt H_j setzen
 \rightarrow Schritt 2
 2. Umfare das Objekt bis:
 - a.) das Ziel erreicht wird \rightarrow END
 - b.) Gerade ST in einem Punkt Q getroffen wird mit Strecke QT kleiner als Strecke H_jT
 QT wird in Richtung T verlassen
 Q als Leavepoint L_j setzen
i um 1 erhöhen
 \rightarrow Schritt 1
 - c.) zum letzten Hitpoint H_j zurueckgekehrt wird
Algorithmus wird abgebrochen
Es gibt keinen Weg zu T
-

- Ev. wird kein Weg zum Ziel gefunden
- Bei Komplexen Hindernissen kann Ziel mit **Backtracking** gefunden werden

Bug3-Algorithmus



Abbildung 7.8: Die Bug 1-Strategie führt den Roboter vom Start zum Ziel. Jedes Objekt wird einmal vollständig umrundet.



Abbildung 7.9: Die Bug 3-Strategie führt den Roboter vom Start zum Ziel. Sobald die Richtung zum Ziel frei ist, verlässt er das Hindernis und fährt dorthin.

Figure 5.3

5.2.3 Labyrinthe

Grundprobleme

- Es wird davon ausgegangen, dass der Roboter berühren oder 'sehen' kann
- Zwei grundlegend Hauptprobleme:

Einen **Weg in ein Labyrinth** finden, um einen bestimmten Gegenstand oder **Schatz zu erreichen** sowie den **Rückweg zum Eingang**

Flucht aus einem Labyrinth von einer unbekannten Stelle aus.

- Enger Zusammenhang zwischen Labyrinth und Graphen \Rightarrow Jeder Korridor:= Kante und jede Kreuzung:= Knoten

Bei bekanntem Labyrinth \Rightarrow Suchproblem auf in Bäumen

Verlassen eines Labyrinths mit Pledge Algorithmus **Grundidee:** Vorsichtig geradeaus bis man auf ein Hindernis trifft und dann mit der 'linken' Hand immer an der Wand entlang bis zum Ausgang.

Problem: Enthält das Labyrinth eine Säule, läuft man für immer im Kreis

- **Lösung** \Rightarrow man folgt der Wand nur solange, bis man wieder in die alte Richtung schaut.

Allgemeingültige Lösung: Drehungen beim Abbiegen an den Ecken mitzählen. Bei jeder Linksdrehung wird der Umdrehungszähler inkrementiert, bei jeder Rechtsdrehung dekrementiert.

- Bewege den Roboter geradeaus bis eine Wand erreicht ist
- Folge der Wand bis Umdrehungszähler 0 ist

Verlassen eines Labyrinths mit Ariadenfaden

- **Ziel:** einen Weg zu einem versteckten Ziel im Labyrinth sowie wieder zurück zum Eingang finden ohne dass eine Karte des Labyrinths bekannt ist

- **Idee:** Wenn man ein Labyrinth betritt Faden ausrollen \Rightarrow zurückverfolgen bringt einen zurück zum Eingang.

Voraussetzungen und grundsätzliches Vorgehen

- einer Wand folgen
- Umdrehen
- Kreuzungen erkennen
- Ziel erkennen
- Faden auslegen und wieder einsammeln
- Faden am Boden erkennen
- Faden zur nächsten Kreuzung folgen

Tarry und Tremaux Algorithmus

- Beispiel für klassische Tiefensuche
- Richtung, in der sich das zu suchende Objekt befindet ist unbekannt.
- Graph kann zyklen enthalten
- Es wird ein zyklisch gerichteter Graph durch jede Kante konstruiert, wobei jede Kante nur einmal pro Richtung besucht wird

Algorithmus:

- Starte willkürlich an einem Knoten
- Folge einem möglichen Pfad, markiere die Kante, in welcher Richtung sie betreten worden ist
- Sind alle Kanten schon betreten, eine auswählen, die bis jetzt nur in die Gegenrichtung betreten wurde.
- Trifft man auf eine Sackgasse oder einen schon besuchten Gang, zurück zur letzten Kreuzung
- Es darf kein Pfad betreten werden, der schon in beide Richtungen besucht wurde.
- Algorithmus ist beendet, wenn der Startpunkt erreicht wird.

5.3 Pfadplanung für mobile Roboter in bekanntem Terrain

5.3.1 Bewegungsplan für mobile Roboter

Ziel der Navigation ist es, ein Fahrzeug in der Umwelt zu bewegen. Dies beinhaltet drei Unteraufgaben:

Globale Pfadplanung

- **Voraussetzung:** es gibt eine Karte
- suche eines Pfades von einem Start- zu einem Zielpunkt in vorhandenem Umgebungsmodell
- evtl. auch Suche nach Pfad mit geringsten Kosten

- Kompletter Pfad beschrieben durch Menge von Punkten

Lokale Pfadplanung berücksichtigt Fahrzeug-Dimension und kinematische Einschränkungen

Path Control generiert geeignete Steuerbefehle, um den vorberechneten Pfad zu folgen

5.3.2 Konfigurationsraum

Herleitung

- Abmessungen, Form, Bewegungsmöglichkeiten des Roboters werden für die Erstellung des Konfigurationsraums benötigt
- Konfiguration q eines Roboters beschreibt Lage und Ausrichtung im Bezugssystem des Umgebungsmodells
- Im zweidimensionalen Raum kann Position in x, y -Ebene und Orientierung ausgedrückt werden
- Konstruktionsbedingt sind einige Konfigurationen für den Roboter in seiner Umgebung nicht zulässig
- Problem; einfachere Darstellung:

Roboter als Punkt angenommen

Abmessungen des Roboters + Objektabmessungen \Rightarrow Konfigurations-Hindernisse

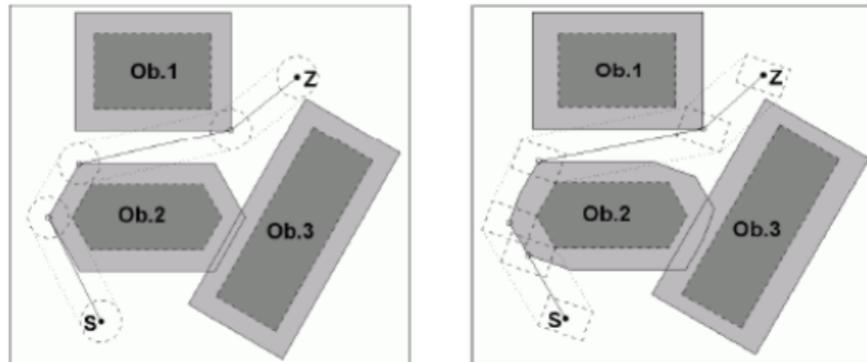


Figure 5.4

Definition

- Summe aller Konfigurations-Hindernisse bildet den **Konfigurationsraum**
- Konfigurationsraum ist Datenstruktur, die es dem Roboter ermöglicht, die Position und Orientierung von Hindernissen in der Umgebung zu definieren
- Der Konfigurationsraum dient als Basis der Wegplanung

Repräsentationen

- Graphen mit Knoten
- Reguläre Gitter

- Quad-Bäume oder Octal-Bäume oder als Voronoidiagramme

5.4 Algorithmen und Methoden

Für die folgenden Algorithmen und Methoden wird ausgeganen, dass Hindernisse bekannt sind und weder Position noch Form ändern

Zellzerlegungen das Umgebungsmodell wird in sich nicht überlappende Zellen unterteilt, die als besetzt oder frei markiert sind

Roadmaps

- Das entstehende Netzwerk muss **topologisch** alle zwischen den Hindernissen befahrbare Wege umfassen.
- Planer kann dann kollisionsfreien Pfad von Start- zu Zielpunkt erstellen

Auf dieses Netzwerk können Standardmethoden der Graphentheorie, wie sie auch in der Autonavigation Verwendung finden, anwendet werden:

- kürzeste Wegsuche mit A*, Dijkstra
- Wegsuche mit Umgehung von Hindernissen mit dem **Sichtbarkeitsgraph**
- Wegsuche mittels eines **Voronoiographen**, **Voronoidiagramms**

Potentialfeldmethoden beinhalten die physikalische Simulation des Roboters als Partikel in einem Feld.

5.4.1 Dijkstra

Funktionsweise

1. Alle Knoten werden als **unbesucht** markiert \Rightarrow kreiert ein set namens **unbesuchtes Set**
 2. Allen Knoten wird ein *temporärer Distanzwert* zugeordnet.
- Ursprungsknoten := 0
 Alle anderen Knoten := ∞
3. Setzen des Ursprungsknoten als *current* Knoten
 4. Für den *current* Knoten alle *unbesuchten* Nachbarn betrachten und deren *temporären Distanzwert berechnen*. Vergleichen des *temporären Distanzwert* mit dem *jetzigen Distanzwert* innerhalb des Nachbarknotens. Falls der temporäre Distanzwert kleiner ist als der jetzige Distanzwert innerhalb der nachbarn wird der temporäre Distanzwert eingetragen.
 5. Sobald alle Nachbarn überprüft wurden und alle werte eingetragen sind, wird der betrachtete Knoten auf **besucht** gesetzt. Somit wird er aus dem angelegten Set entfernt und wird nicht mehr überprüft.
 6. Sobald der Zielknoten als besucht markiert ist dann terminiert der Algorithmus, **else** wird der Knoten mit der niedrigsten temporären als neuer **current** Knoten gesetzt und schritt 4 wird wiederholt.

5.4.2 A*

Funktionsweise

5.4.3 Wegsuche mit dem Sichtgraph-Algorithmus

Visibility Map

- enthält Eckpunkte von Polygonen, den Ecken der Hindernisse
- zwei Knoten der Visibility Map teilen eine Kante, wenn die beiden Eckpunkte voneinander in Sichtweite sind

Visibility Graph

- ist die einfachste Visibility Map
- Knoten umfassen: **Startknoten**, **Zielknoten** und alle Eckpunkte der Hindernisse
- die Kanten sind Liniensegmente, die zwei Knoten in Sichtweite verbinden
- Hinderniskanten sind Teil des Sichtgraphen

Sichtgraphenalgorithmus

- verbinde **Start-**, **Zielpunkte** und die **Ecken** der Hindernisse durch Geraden, die nicht durch Hindernisse laufen dürfen
- Ergebnis ist ein Graph, dessen Knoten Orte sind und dessen Kanten mögliche Wegstücke zwischen diesen Knoten darstellen

- Kanten sind gewichtet mit der Entfernung zwischen den Knoten
- gesucht: Menge der möglichen Geraden, die den Startpunkt auf dem kürzesten Weg mit dem Zielpunkt verbindet
- **Nachteil:** führt unmittelbar an Hindernissen entlang, enthält noch viele nutzlose Kanten

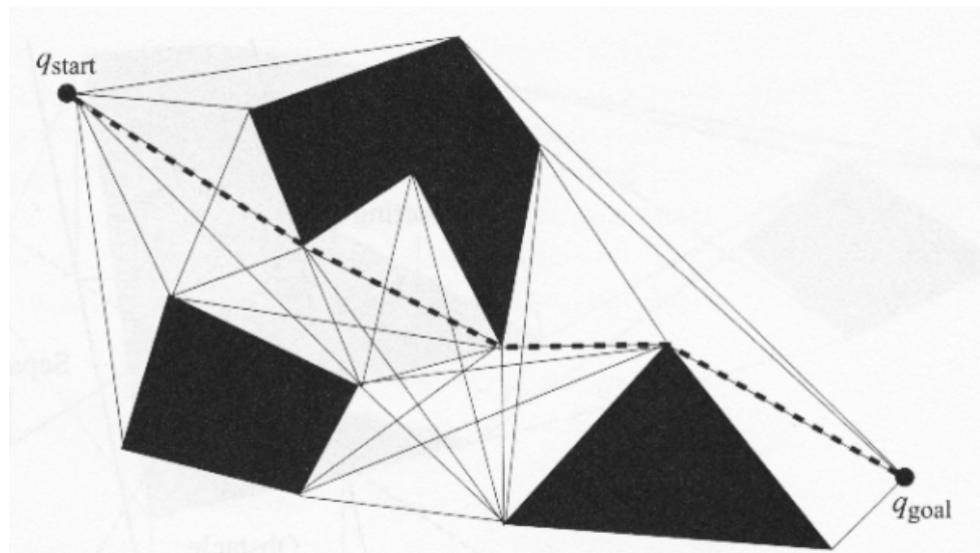


Figure 5.5

Reduzierter Visibility Graph Definition von **unterstützenden** und **trennenden** Kanten:

- **unterstützende Kante:** Tangente zu zwei Hindernissen, so dass die Hindernisse auf derselben Seite der Linie liegen
- **trennende Kante:** Tangente zu zwei Hindernissen, so dass die Hindernisse auf gegenüberliegenden Seiten der Tangente liegen

Der reduzierte Sichtgraph besteht nur aus unterstützenden und trennenden Kanten.

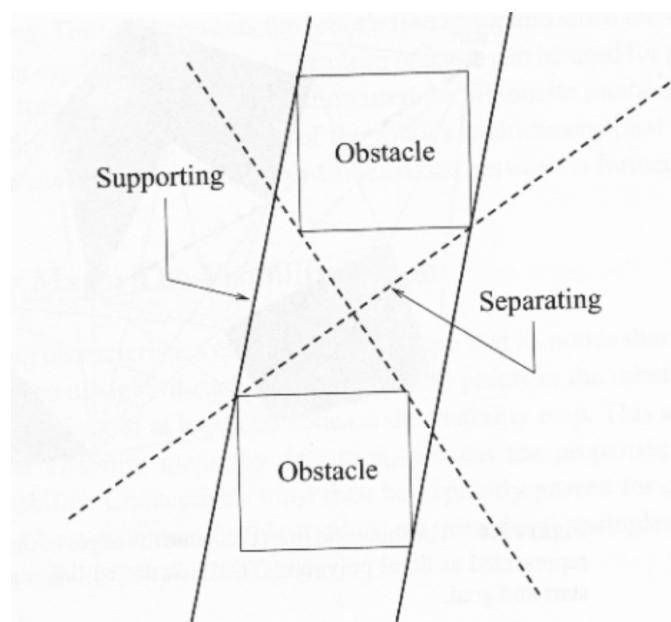


Figure 5.6

Konstruktion des Graphen

- Alle Liniensegmente vv_i mit $v \neq v_i$ müssen getestet werden, ob sie keinen Schnittpunkt mit einer Kante eines Hindernisses haben
- **Nachteil:** Komplexität $O(n^3)$
- **Effizienter:** Plane Sweep Algorithmus mit Komplexität $O(n^2 \log n)$

Algorithmus

- **Input:** A set of vertices v_i (whose edges do not intersect) and a vertex v
- **Output:** A subset of vertices from v_i that are within line of sight of v
- For each vertex v_i , calculate α , the angle from the horizontal axis to the line segment $\overline{vv_i}$
- Create the vertex list ϵ , containing the α_i 's sorted in increasing order.
- Create the active list S , containing the sorted list of edges that intersect the horizontal half-line emanating from v .
- **For all** α_i **do**
 - if** v_i is visible to v **then**
 - Add the edge (v, v_i) to the visibility graph.
 - end if**
 - if** v_i is the beginning of an edge, E , not in S **then**
 - Insert the E into S
 - end if**
 - if** v_i is the end of an edge in S **then**
 - Delete the edge from S
 - end if**
- **end for**

5.4.4 Voronoi-Diagramme

Allgemeines Voronoi-Diagramm

- Eine Fläche wird willkürlich mit Punkten besetzt \Rightarrow erzeugt Polygonflächen
- Alle Punkte einer Polygonfläche liegen am nächsten zu dem Punkt der im Zentrum dieses Polygons liegt
- Jede Zelle hat genau ein Zentrum
- Die Punkte des Gebiets, die zu mehreren Zentren den gleichen Abstand haben, bilden die Grenze zwischen den einzelnen Zellen
- Als abstandsfunktion wird der Euklidische Abstand verwendet:

$$dist(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$
- Das Voronoi-Diagramm ist die Menge solcher Grenzlinien

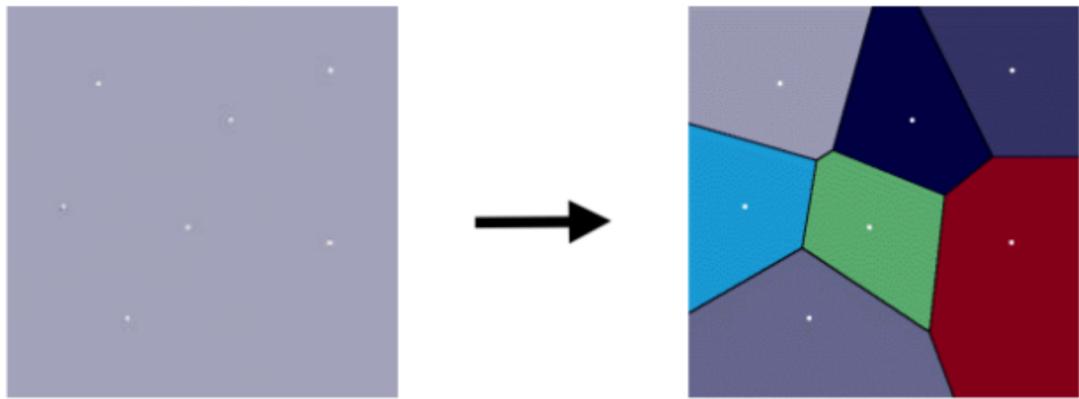


Figure 5.7

Immer dann gut einsetzbar, wenn ungenaue Sensoren zur Verfügung stehen oder die Umwelt ungenau geometrisch modelliert wurde oder sich dynamisch ändert.

Generalisierte Voronoi-Diagramme

- Bei der Navigation von Robotern wird eine verallgemeinerte Version des Voronoi Diagramms verwendet, das sogenannte **generalisierte Voronoi Diagramm**(GVD)
- **Generalisierung** betrifft die **Form der Zentren** und die Art der **Abstandsfunktion**
- **Zentren** können aus **komplexeren Formen** wie Linien, Kurven oder Polygonen **statt Punkten** bestehen
- Objekten der Umgebung werden als **Voronoi-Zentren** behandelt
- Die Menge aller **Voronoi-Kanten**, das GVD, stellt mögliche **kollisionsfreie Wegstücke** dar
- Falls sich der Roboter entlang einer **Voronoi-Kante** bewegt, kann er nicht mit Hindernissen kollidieren
- Beliebtes Verfahren zur Repräsentation des Konfigurationsraumes und Erzeugung eines Graphen.
- Punkte an denen sich **Voronoi-Kanten** schneiden, werden zu **Voronoi-Knoten**

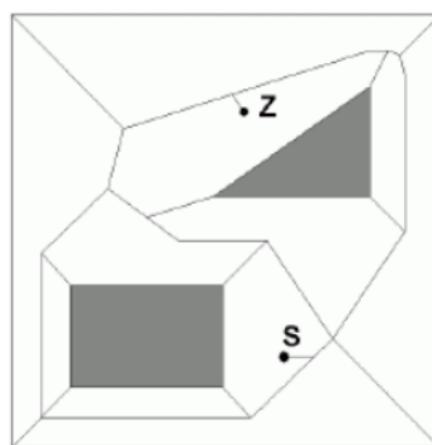


Figure 5.8

Start- und Zielpunkt des gesuchten Weges liegen normalerweise nicht auf dem Diagramm. Diese beiden Punkte werden mit der nächstliegenden Kante verbunden. Die Berechnung des optimalen

Weges auf dem Voronoi-Diagramm kann mit üblichen Graphenalgorithmen vorgenommen werden.

5.4.5 Navigation in einer Rasterkarte

- Die Karte ist durch ein binäres Raster mit freiem Platz und Hindernissen gegeben.
- Das Raster wird ausgehend vom Startpunkt: **geflutet** ⇒ Füllt den gesamten Hindernisfreien Raum
- In jeder Iteration haben alle Pixel auf einer Wellenfront dieselbe Pfadlänge im Raster bezogen auf den Zielpunkt
- Backtracking vom Zielpunkt zurück zum Startpunkt und erstellt dabei eine Liste der passierten Rasterpunkte
- Gewählt kann jeder Punkt im Raster werden, dessen Wert eins geringer ist, als der aktuell betrachtete
- Trifft dies auf mehrere Punkte zu, wird willkürlich einer gewählt

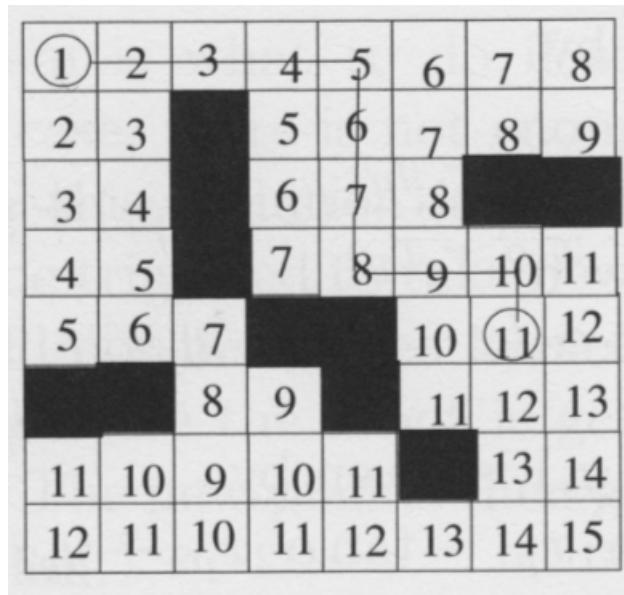


Figure 5.9

Fluten der Karte

- Ein **Rasterpunkt** besitzt vier Nachbarpunkte: 'N', 'O', 'S' , 'W'
- Ein Rasterpunkt z ist mit 0 für freien Raum belegt und -1 für ein Hindernis
- Wenn ein Rasterpunkt eine Nummer erhalten hat, behält er diese Nummer
- Der Algorithmus benutzt aus Performancegründen zwei Stacks S_0 und S_1 , die abwechselnd gefüllt und geleert werden. Die Stacks sind anfangs leer

Algorithm 7.7 Flooding a raster with thick walls

```

Initialization:  $i := 1; j := 0;$ 
starting cell  $z := 1$ ; push  $S_i$ ;
repeat
   $i := (i + 1) \bmod 2; j := (j + 1) \bmod 2$ 
  repeat
    pop stack  $S_i$ ; check surrounding of the cell taken out of stack:
    for all cells with  $z = 0$  do
      if  $\min_{z>0}(O,S,W,N) > 0$  then
         $z := \min_{z>0}(O,S,W,N) + 1$ ; push to stack  $S_j$ ;
      else
        do nothing;
      end if
    end for
    until stack  $S_i$  is empty
  until stack  $S_j$  is empty

```

Figure 5.10

Alternativer Wave Front Planer Siehe Ursprungsfolie Kapitel 5, Seite 29/30

5.4.6 Potentialfeld methode

Grundidee

- Vorbild: **Elektrisches Feld**
- Start- und Zielpositionen, die Positionen aller Hindernisse müssen bekannt sein.
- Der **Zielpunkt und Freiräume** erhalten ein **anziehendes Potential**
- Der **Startpunkt, die Hindernisse** und Wände erhalten ein **abstoßendes Potential**
- Es wird eine Karte generiert mit virtuell anziehenden und abstoßenden Kräften
- Kräfte nehmen linear mit dem Abstand zu den Objekten ab.
- **Ein Objekt bewegt sich nach der Methode des steilsten Abstiegs im Potentialfeld auf das Ziel zu.**

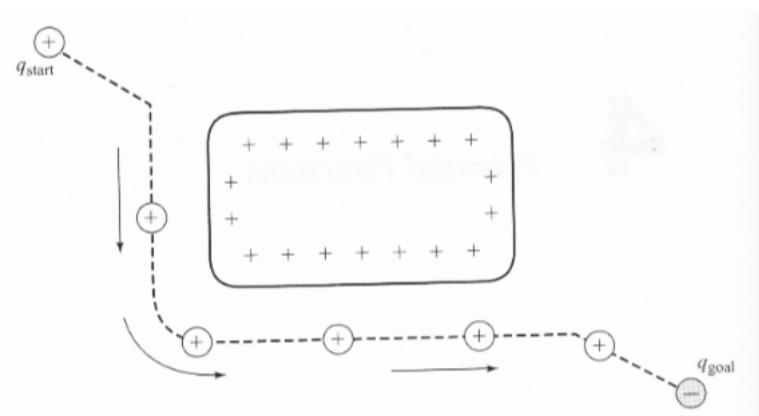


Figure 5.11

6 Probabilistische Methoden und Kartierungen

6.1 Problemstellung

Sensordaten Roboter empfängt regelmäßig Sensordaten, jedoch mit Unsicherheit behaftet.

Steuerdaten Roboter führt regelmäßig Bewegungen mit Steuerdaten u_k durch, aber diese entsprechen nicht exakt den vorgegebenen Steuerdaten.

Position und Umgebungsmodell Position schätzt seine globale Position aufgrund der Sensor- und Steuerdaten in seiner Umgebungskarte, jedoch auch mit Unsicherheit behaftet

6.2 Modellierung von Unsicherheit

- Viele Aussagen bei mobilen Robotern sind unsicher
- **Grundlegende Idee:** Modellierung von Unsicherheiten durch Wahrscheinlichkeiten
- Interpretierung der eigentlichen Lokalisierung als Wahrscheinlichkeitsdichte-Problem
- Auch bei weniger präzisen Umgebungsmodellen einsetzbar
- Sie erlauben es, den Zustand eines **dynamischen Systems** probabilistisch zu schätzen

6.3 Umgebungsmodellierung mit Occupancy Grids

6.3.1 Satz von Bayes

$$p(A|B) = p(B|A) * \frac{p(A)}{p(B)}$$

- $p(A)$ ist Wahrscheinlichkeit das die Aussage A zutrifft.
- $p(A|B)$ bezeichnet die Wahrscheinlichkeit $p(A)$ unter Voraussetzung dass B gilt

6.3.2 Evidence Grids

- **Problem:** reale Sensordaten erhalten häufig Rauschen; Rauschen bei Sensordaten führt zu Abweichungen des Idealwerts \Rightarrow schwerwiegende Fehler
- Umgebung wird in einem zweidimensionalen Gitternetz repräsentiert

- **Occupancy Grids (Belegungsraster)** speichern in jeder Zelle, ob der Weg für einen Roboter frei oder blockiert ist (**Binär**)
- **Evidence Grids (Beweisraster)** Untermenge der Occupancy Grids. Sammeln Beweismaterial und erstellen damit Karten (**Belegte Zellen nach Satz von Bayes**)
- Anhand von Sensordaten wird die Umgebung in einzelne Kartenzellen zergliedert, denen jeweil eine Besetzwahrscheinlichkeit der näheren Umgebung zugewiesen wird.

6.3.3 Anwendung des Satzes von Bayes

- Die Information über das Verhalten des Sensors wird mit einbezogen

$$z(x, y) = \frac{p(\text{Zelle belegt} \mid \text{Sensorwert})}{p(\text{Zelle nicht belegt} \mid \text{Sensorwert})}$$

- Die Anwendung des Satzes von Bayes auf diese Formel führt letztlich zu:

$$z(x, y) = \frac{p(\text{Sensorwert} \mid \text{Zelle belegt}) * p(\text{Zelle belegt})}{p(\text{Sensorwert} \mid \text{Zelle nicht belegt}) * p(\text{Zelle nicht belegt})}$$

- Die Wahrscheinlichkeit, dass eine Zelle belegt ist oder nicht belegt ist, wird zu Beginn mit 0.5 angenommen
- Dieser Wert wird fortlaufend mittels der aktuellen Sensorwerte aktualisiert.
- Die direkte Anwendung des Bayessischen Filters auf das Selbstlokalisierungsproblem ergibt sich die sogenannte **Markov-Lokalisierung**

6.4 Bayes-Filter Algorithmus

6.4.1 Algorithmus

- Vertrauenszustand (**belief**) spiegelt interne Wissen des Roboters über den Zustand seiner Umgebung wider
- Rekursiver Algorithmus: $bel(x_t)$ zum Zeitpunkt t wird berechnet aus dem belief $bel(x_{t-1})$ zum Zeitpunkt $t-1$
- z_t : letzte Information über den momentanen Umgebungszustand mittels Sensoren (**Observationsmesswert**)
- u_t letzte Kontrolldaten, Zustandsänderung im Zeitintervall $(t-1; t)$ (**Aktionsmesswert**)
- x_t Zustand zum Zeitpunkt t
- bei $(x_t) = p(x_t | z_1 : t, u_1 : t)$ ist die **Wahrscheinlichkeitsverteilung** über den **Zustand x_t zum Zeitpunkt t** , abhängig von allen vergangenen Sensorinformationen $Z_1 : t$ und Kontrolldaten $u_1 : t$

6.5 Markov Lokalisierung

6.5.1 Algorithmus

- Anwendung des Bayes-Filter auf das Lokalisierungsproblem erfordert eine **Karte als Input**
- Probabilistische Verfahren bedienen sich zur Schätzung der a posteriori Wahrscheinlichkeit der **Markov-Annahme** oder Unabhängigkeitsannahme. Nachfolgender Zustand hängt nur vom aktuellen Zustand und nicht von der Historie ab.
- Ziel der Markov Lokalisierung ist es, jeder möglichen Roboterposition einen Wahrscheinlichkeitswert zuzuordnen.

Position völlig unbekannt \Rightarrow Gleichverteilung

Position bereits durch Odometrie oder ähnliches eingeschränkt \Rightarrow Verteilung auf eingegrenzten Bereich verteilt

Wenn die Position bekannt ist, folgt die Wahrscheinlichkeit für diese Position = 1

- Markov Lokalisierung benötigt möglichst genaues Umgebungsmodell

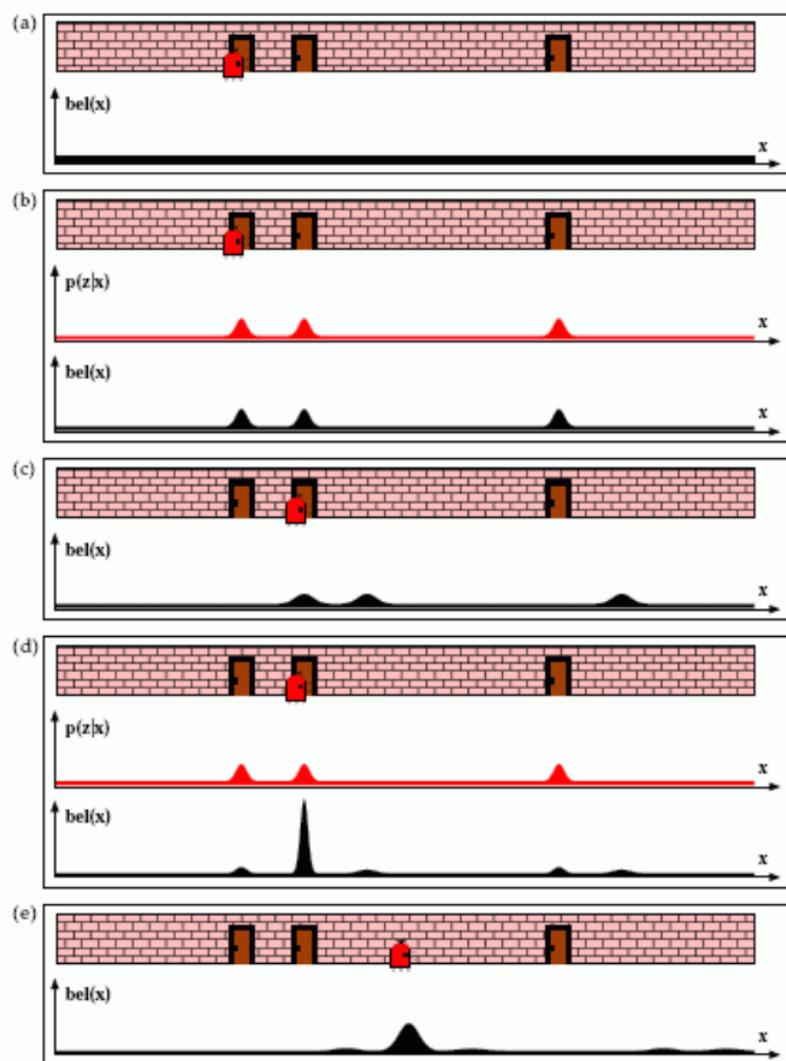


Figure 6.1: Markov Algorithmus

6.6 Monte Carlo Lokalisierung

6.6.1 Grundsätzliches Vorgehen

- Stichprobenbasierendes Approximationsverfahren
- Spezialform von Markov Lokalisierung

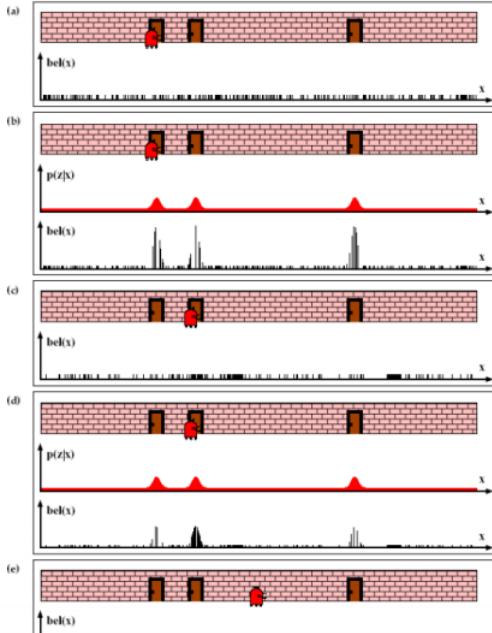


Figure 6.2: Monte Carlo Lokalisierung

Definition

- iterativer Bayesscher Filter, welcher als Schätzer für die zukünftige Wahrscheinlichkeitsverteilung der Roboterposition verwendet wird
- Unterscheidet sich im Gegensatz zu gridbasierten Lokalisierungsverfahren in der Betrachtung und in der Verarbeitung

Praxis: viele der Gitterzellen besitzen Wahrscheinlichkeit von 0

Miteinbeziehung dieser Gitterzellen in Berechnungen \Rightarrow ineffizient

diese können vernachlässigt werden

fokussieren der Gitterzellen, die die wahrscheinlichsten Positionen widerspiegeln

Funktionsweise

- Positionsschätzung bei (x_k) wird durch eine Menge von **Partikeln** dargestellt
- Es besteht keine Information über die Anfangsposition; Partikel sind **zufällig verteilt**
- Durch **Sensormessung** z werden die **Gewichte** (Strichhöhe) verändert
- **Resampling:** Aus der Partikelmenge werden zufällig aber entsprechend ihrem Gewicht Partikel gezogen \Rightarrow integrierung des Steuerbefehls (u_k) integriert.
- Es erfolgt eine erneute Gewichtung mit einem neuen Sensorwert

- Anschließend ein erneutes Resampling und Integrierung des Steuerbefehls

Vorteil

- zur Laufzeit kann die Größe der Stichprobenmenge variabel sein
- je unsicherer die Roboterposition ist, desto größer ist die Stichprobenmenge

6.6.2 Partikelmengen

- Jeder Partikel stellt eine **Hypothese** für den **Zustand** x dar
- Generierung einer Partikelmenge X aus einer Wahrscheinlichkeitsichtete p :

```
Algorithm generateParticle( $p$ ):
     $\chi = \emptyset;$ 
     $i = 0;$ 
    while  $i < M$  do
        generiere Zufallszahl  $x$  aus  $[a,b]$ ;
        generiere Zufallszahl  $q$  aus  $[0,c]$ ;
        if  $q < p(x)$  then
             $i = i+1;$ 
             $\chi = \chi \cup \{x\};$ 
        endif
    endwhile
return  $\chi$ ;
```

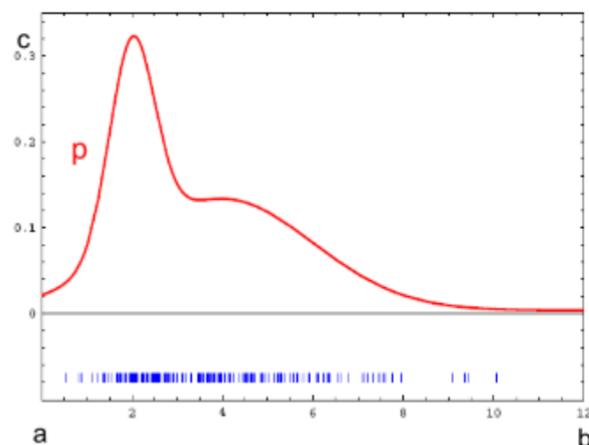


Figure 6.3: Partikelmengen

6.7 Kalman-Filter

6.7.1 Definition

- Zustandsschätzer für dynamische Systeme
- Spezielle Version eines Bayes-Filters
- Dient zur Fusion von zwei unterschiedlichen, stochastisch unabhängigen Informationsquellen

Anwendung fusion der Odometriedaten mit externen Messungen

6.7.2 Vorgehen

- $Bel(x_t)$ wird durch seinen Erwartungswert μ sowie die Kovarianz Σ_t approximiert.
- Zu jedem Zeitpunkt wird eine **Zustandsschätzung** geliefert, die aus einer Schätzung des aktuellen Zustandes und aus einer Vorhersage des Nachfolgezustandes nach Ausführung einer Aktion besteht.
- In die **Zustandsschätzungen** werden unabhängige Sensormessungen integriert

- In der **Vorhersagephase** benutzt der Kalman Filter die Zustandsschätzung vom vorhergehenden Zeitschritt um Zustandsschätzung für den aktuellen Zeitschritt zu erzeugen
- In der **Update oder Korrekturphase** werden die Messinformationen des aktuellen Zeitschritts verwendet, um die Vorhersage zu verbessern.
- Das Fehler Modell der Schätzung soll optimal aktualisiert werden auf Basis vorhandener Informationen

6.7.3 Einschränkungen

- Fehlermodelle sind Gaußverteilungen
- Die Zustandsverteilung ist eine Gaußverteilung

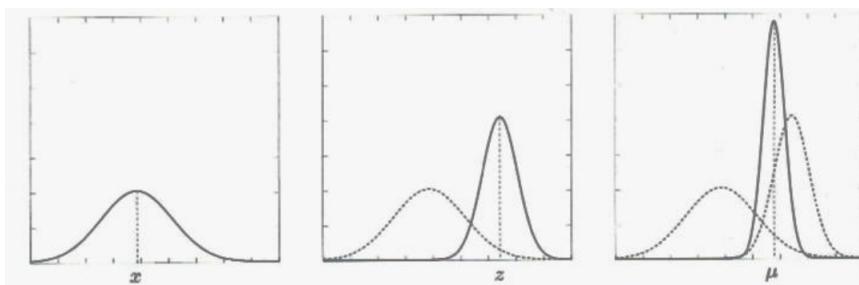


Figure 6.4: Kalman Filter

- **Links:** Unsicherheit im aktuellen Zustand x
- **Mitte:** eine unabhängige Messung z liefert konkurrierende Informationen (*Mittelwert und Varianz*)
- **Rechts:** Fusion beider Daten liefert eine Mittelung, gewichtet mit der Sicherheit der Informationen, sowie reduzierte Varianz, d.h. eine größere Sicherheit in dem gefilterten Zustand

6.8 Simultaneous Localization and Mapping

6.8.1 Landmarkenbasiertes SLAM Problem

SLAM Simultaneous Localization and Mapping

- **Ausgangspunkt** Roboter exploriert eine unbekannte, statistische Umgebung
 - Roboter kennt seine Pose (Position und Orientierung) nicht genau
 - Es existiert keine Karte der Umgebung
- **Bekannt** sind Sensor- und Steuerdaten: $d = u_1, z_1, u_2, z_2 \dots u_k, z_k$
- **Gesucht** Karte m mit M Landmarken: $m = l_1, x, l_1, y, \dots, l_M, x, l_M, y$
- Weg des Roboters $x_1, x_2, \dots x_k$

Probabilistische Algorithmen Ungenauigkeiten in den Messdaten werden durch Wahrscheinlichkeitsverteilungen modelliert

- bekannt sind die **Roboter Bewegungsbefehle** (die Kontrolldaten, die Steuerkommandos u_t)

- bekannt sind die **Beobachtungen** z_t der nahe gelegenen **Landmarken** bestehend aus Entfernung und Winkel
- die Sensorik kann sowohl die Beobachtungsrichtung als auch die beobachtete Entfernung einer Landmarke zur Verfügung stellen
- gesucht ist eine Schätzung der Karte der Merkmale, der Landmarkenpositionen, sowie der Pfad des Roboters, d.h. seine aktuelle und frühere Posen

6.8.2 Problemstellung

- Roboterpfad und Positionen der Landmarken in der Karte sind unbekannt
- Die Zuordnung von Messdaten zu Landmarken sind i.d. Regel unbekannt
- Roboter muss entscheiden, ob Messdaten einer bereits beobachteten Landmarke zugeordnet werden können oder einer noch nicht gesehenen Landmarke
- Problematik der Zuordnung wird durch Unsicherheit in der Roboterposition verstärkt

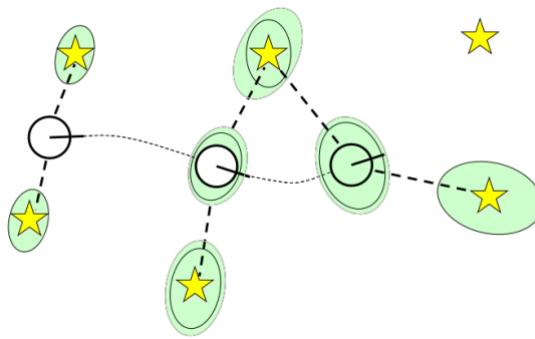


Figure 6.5

6.8.3 Funktionsweise

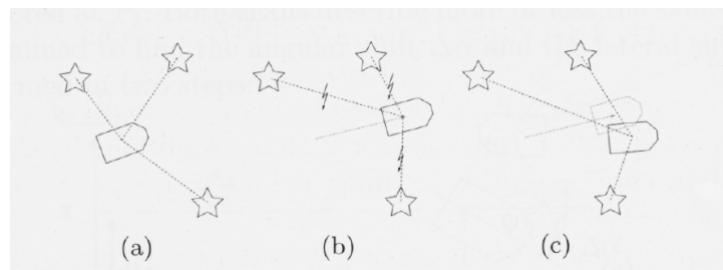


Figure 6.6: SLAM Darstellung

- (a) Roboter misst Distanzen zu den Landmarken
- (b) Roboter schätzt seine neue Position anhand von Odometriedaten; Odometrie-basierte Roboterpose kann zur Schätzung der neuen Distanzen zu den in (a) verwendeten Landmarken herangezogen werden
- Nach Vergleich zwischen Roboter und Landmarken kann der Roboter seine Position (c) korrigieren

6.8.4 Hinzunahme neuer Landmarken

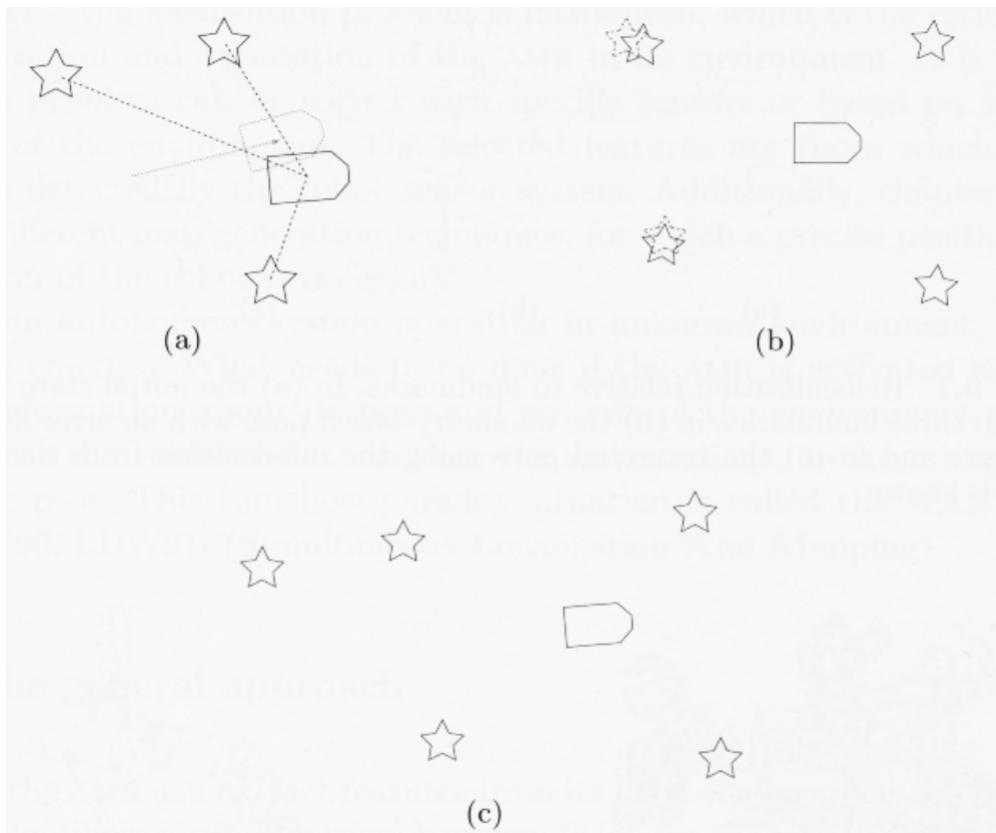


Figure 6.7

- (a) entspricht der Situation aus der vorhergehenden Folie.
- In (b) lokalisiert sich das Fahrzeug nach einer Bewegung erneut anhand zweier 'alter' Landmarken und der letzten lokalen Karte
- Führt aufgrund der Fehler zu einer ungenauen, globalen Roboterposition
- Die Position der neuen Landmarken wird relativ zu der vermeintlich bekannten, korrekten Position der alten Landmarken bestimmt
- Positionsannahme ist inkorrekt
- Verschiebung der Position der 'alten' Landmarken wird geschätzt und korrigiert sowie auf die Position der neuen Landmarken angewandt

6.8.5 Aufbau eines SLAM-Graphen

- Sämtliche Messvorgänge sind fehlerbehaftet, auch die Positionsbestimmung der Landmarken
- Die Unsicherheit wird in der folgenden Abbildung als Fehlerellipse dargestellt.
- Roboter schätzt die Landmarken **A** und **B**
- nach Bewegung des Roboters nimmt die Genauigkeit der Lokalisierung ab
- Die Unsicherheit der Positionsschätzung der Landmarken **C** und **D** steigt
- Der Roboter erkennt eine bereits zuvor gesehene Landmarke

- Eine Verknüpfung mit der früheren Information über die Landmarke reduziert die Unsicherheit bei der Positionsbestimmung und damit auch die Unsicherheit über die zugehörigen früheren Roboterpositionen

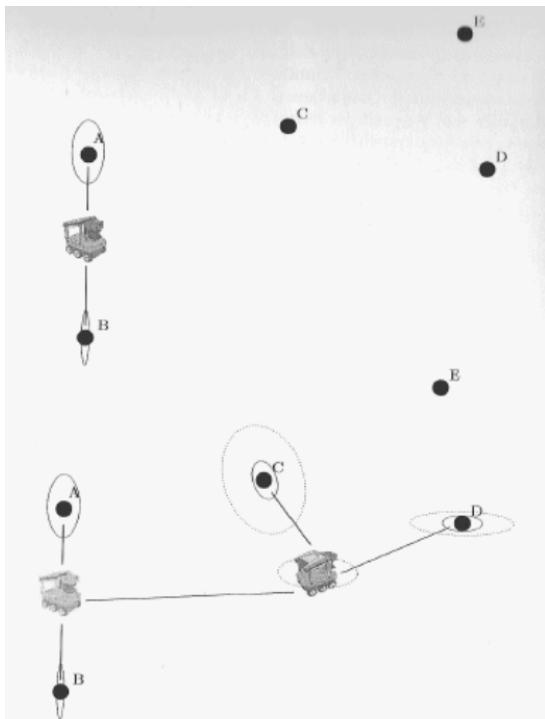


Figure 6.8

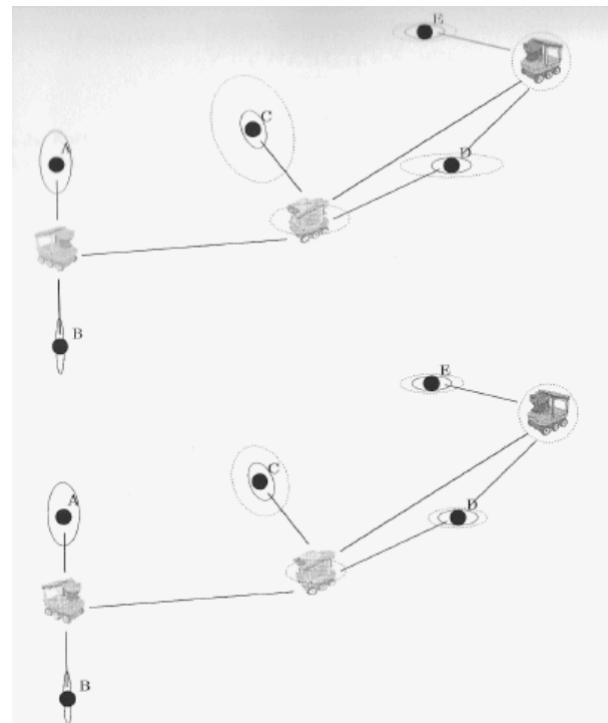


Figure 6.9

6.8.6 Varianten von SLAM

Vollständiges SLAM

- Roboter schätzt eine Umgebungskarte m
- Roboter schätzt seine aktuelle Pose x_t und **alle** zurückliegenden Posen x_{t-1} bis x_1
- Grundlage sind die bisher wahrgenommenen Sensordaten $z_1 : t$
- Sowie alle ausgeführten Aktionen $u_1 : t-1$
- Es muss die Verteilung $P(m, x_1 : t | z_1 : t, u_1 : t-1)$

Inkrementelles Slam

- Roboter schätzt nur die Karte m sowie die aktuelle Position x_t
- Es muss die Verteilung $P(m, x_t | z_1 : t, u_1 : t-1)$ geschätzt werden

6.8.7 Bayesian Netzwerk für landmarkenbasiertes SLAM

- Die einzelnen Landmarken sind unabhängig
- Gegeben sind die Roboterposen

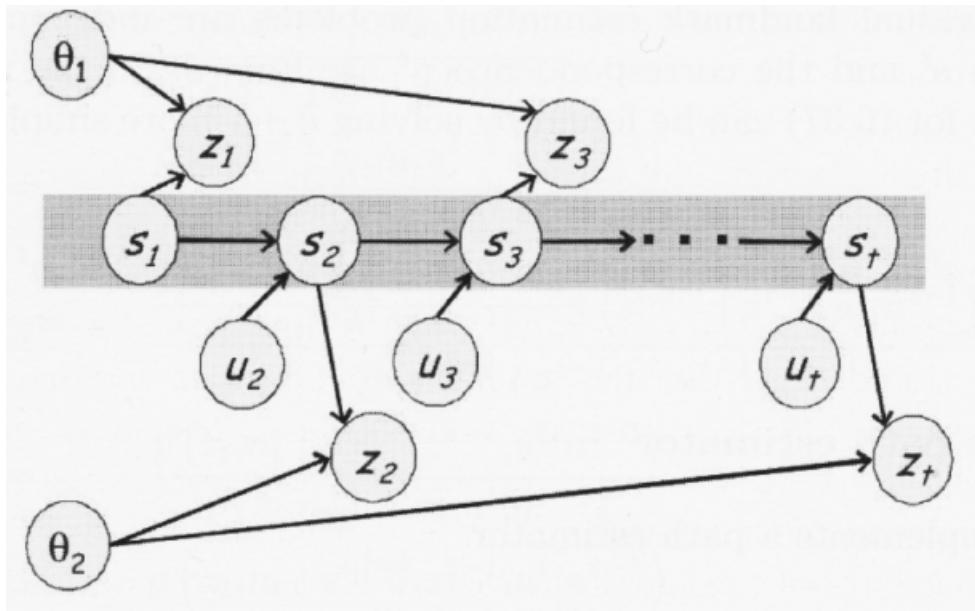


Figure 6.10: Bayes Netzwerk

Modell von Variablen und deren Abhängigkeiten als dynamisches Bayes-Netzwerk.

- Kern des Modells bilden die
 - Zeitreihe der Roboterzustände s_1, s_2, \dots, s_t
 - die Positionen der Landmarken θ_k
 - die Kontrollvariablen u_t
 - und die gemessenen, beobachteten Landmarken Positionen z_t
- Der Roboter bewegt sich von s_1 nach s_t mit einer Folge Kontrolleingaben u_2, \dots, u_t
- Der Roboterzustand s_t zum Zeitpunkt t ist lediglich vom Roboterzustand s_{t-1} zum vorhergehenden Zeitpunkt und dem ausgeführten Steuerkommando u_t des Roboters abhängig
- Zum Zeitpunkt $t = 1$ beobachtet der Roboter die Landmarkenpositionen θ_1 mittels z_1 zum Zeitpunkt $t = 2$ beobachtet er θ_2 via z_2 und zum Zeitpunkt $t = 3$ wieder θ_1
- Die Beobachtung z_t ist abhängig von der globalen Position der Landmarke θ_k und dem aktuellen Roboterzustand s_t
- **FastSLAM** zerlegt das Problem
 - in die **Lokalisation** (Wissen über den vom Roboter zurückgelegten weg s_1, \dots, s_t)
 - und einer Sammlung von einzelnen **Landmarken-schätzungen** z_k , die von der geschätzter Roboterpose abhängen
- Zeitkomplexität von **FastSLAM** ist $O(fM)$
 - f konstanter Faktor
 - M Anzahl der Landmarken

7 Schwarmrobotik und Evolutionäre Robotik

7.1 Schwärme und deren Verhalten in der Natur

Schwarmdefinition Der Begriff **Schwarm** bezeichnet einen Verband von fliegenden oder schwimmenden Lebewesen, der sich koordiniert bewegt. Im Unterschied zu anderen Gruppen zeigt er ein sogenanntes **Schwarmverhalten**

7.1.1 Computersimulation von Schwärmen - Algorithmus von Craig Reynolds

Die einzelnen Individuen agieren in Abhängigkeit von der Position und der Geschwindigkeit der benachbarten Boids nach folgenden Regeln:

Separation Bewege dich weg sobald dir andere zu nahe kommen

Alignment Bewege dich in die gleiche Richtung wie deine Nachbarn

Cohesion Bewege dich zum Mittelpunkt der benachbarten Vögel

Voraussetzung Reynolds setzte voraus, dass **alle** Vögel innerhalb eines fixen gegebenen Radius interagieren. Die Nachbarschaft ist bei Reynolds charakterisiert durch einen Abstand vom Zentrum des Vogels und durch einen bestimmten Winkel ausgehend von der Flugrichtung. Tiere außerhalb dieser Nachbarschaft werden ignoriert.

7.2 Schwarmintelligenz

Unter **Schwarmintelligenz** versteht man Systeme bestehend aus vielen primitiven, mobilen Agenten, die:

- gemeinsam agieren
- miteinander kommunizieren können
- im Kollektiv ein komplexes Problem lösen
- ohne zentrale Steuerung sich selbst organisieren

Kollektive Intelligenz Die Individuen agieren ziemlich beschränkt, die Gesellschaften dagegen sind ungemein leistungsfähig. Geeignet zur **Lösung schwieriger Optimisierungsprobleme**

7.3 Multi Robot Systems

Einsatz von simplen Robotern, deren Handlungsmöglichkeiten und Wahrnehmungssysteme stark begrenzt sind.

Vorteile

- **robust**
- **skalierbar**
- **flexibel**
- **Kosten** statt eines euren einzelnen Roboters, der eventuell nicht so leicht oder schnell zu ersetzen ist, werden billige Komponenten eingesetzt.
- **Verlässlichkeit** wenn ein einzelner Roboter oder Softwareagent ausfällt übernehmen andere Roboter dessen Aufgabe und fügen sich neu ins Kollektiv
- **Flexibilität** viele kleine kooperierende Roboter können bei sinnvoller Zusammenarbeit Probleme bewältigen, die ein großer Roboter alleine eventuell nicht bewältigen kann.

Problem Das Schwarmverhalten ist aufgrund des Einzelverhaltens nur schwer vorhersagbar.

7.4 Ameisenalgorithmen

7.4.1 Optimaler Weg bei futtersbeschaffenden Ameisen

Funktionsweise

- Eine Ameise verlässt den Bau, das nest (N) und sucht Futter auf einem **zufälligen Weg**
- Es gibt mehrere Zweige zur Futterquelle
- Weg wird mit **Pheromon**, einer chemischen Substanz markiert.
- Findet die Ameise Futter, schleppt sie das Futter auf dem gleichen oder einem anderen Weg zurück, der Weg wird dabei ev. ein weiteres Mal markiert.
- Weitere Ameisen, die zur Futtersuche starten, **orientieren sich** bei der Futtersuche **an den Pheromonspuren**
- Ameisen folgen **beworzugt**, aber nicht immer den markierten Wegen
- Auf langen Wegen ist die Ameisendichte wegen der größeren Entfernung geringer, das Pheromon verdunstet schneller.
- Wenn ein Ausreißer einen kürzeren Weg findet und einige andere Ameisen beginnen diesem Weg zu folgen, wird die Ameisendichte auf dem längeren Weg immer geringer und der kürzere Weg setzt sich durch

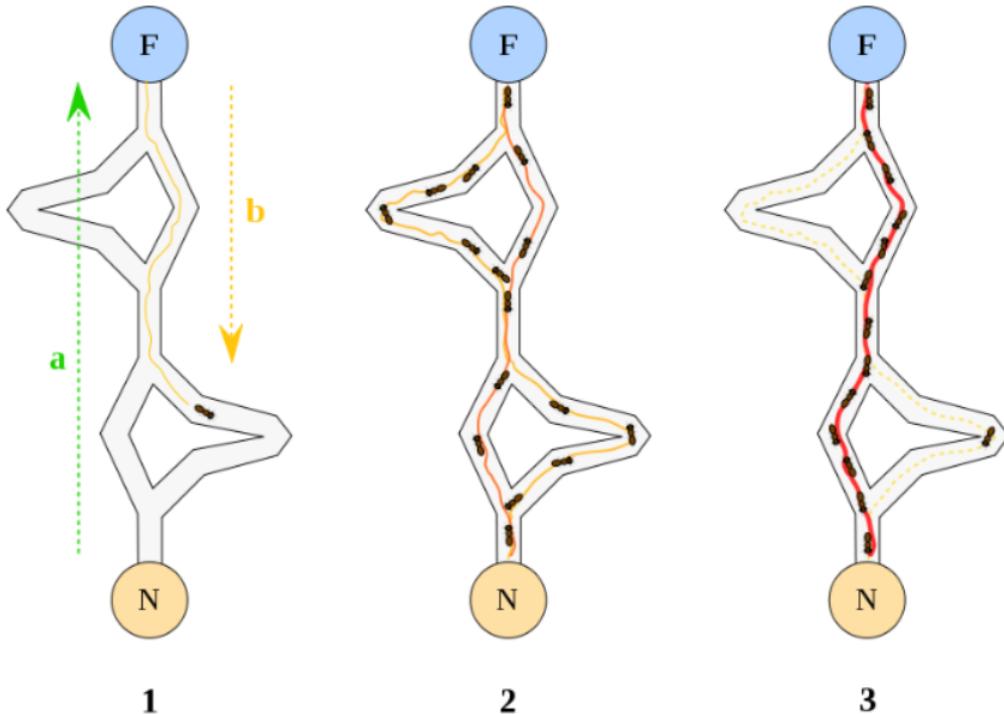


Figure 7.1: Darstellung des Ameisenalgorithmus Prinzips

7.4.2 Ant Colony Optimization Algorithm

- **Ant Colony Optimization(ACO)** ist der Überbegriff für Ameisen-basierende Algorithmen

Kategorien

- **Tourenplanung (Routing)**: ⇒ Travelling Salesman Problem
- **Zuordnung (Assignment)**: optimale Zuordnung von Personen oder Betriebsmitteln auf Stellen oder Aufgaben
- **Ablaufplanung (Scheduling)**: Verteilung von knappen Ressourcen auf Prozesse die zeitlich begrenzt sind
- **Teilmengen Problem (Subset)**: aus einer Menge von Objekten muss eine Teilmenge gefunden werden, damit eine vorgegebene Bedingung erfüllt und eine Zielfunktion optimiert wird.

Eigenschaften

- optimaler Weg ist der **kürzeste Weg** zwischen zwei Punkten
- **globale Information**: Belegung mit künstlichen Pheromonen als zentrale Idee
- Wahrscheinlichkeits-gestützte, **lokale Entscheidungen** - Ameise erkennt unmittelbare Nachbarschaft

Funktionsweise

1. Ameisen laufen entlang des Graphen
2. Eine Ameise erzeugt eine Lösung gemäß lokaler Information und Pheromon

3. Update beinhaltet neu aufgetragene Pheromone und Verdunstung bereits vorhandener Pheromone
4. Operationen, die globales Wissen vorraussetzen und damit nicht von einzelnen Ameisen bewerkstelligt werden können
 - Diskretisierung der Zeit t : in einem Zeitschritt erzeugen alle Ameisen eine vollständige Lösung.
 - Eine **Pheromon-Matrix** enthält die Intensität der Pheromone $T_{ij}(t)$ enthält die Intensität der Pheromone auf einer Kante vom Knoten i zum Knoten j im Graphen
 - Eine Matrix für lokale Informationen enthält die Sichtbarkeit der Stadt (d.h. die jeweils reziproke Distanz): $n_{ij} = \frac{1}{d_{ij}}$

7.4.3 Traveling Salesman Problem

- Vorhanden: **vollständiger gerichteter Graph**
- Gesucht: **Rundtour durch alle Städte**

Ameisen laufen durch den Graphen, die Kolonie ermittelt den Optimalen Weg.

- Es erweist sich als vorteilhaft, für jede Ameise eine andere zufällig gewählte Stadt als Ausgangspunkt für die Tour zu nehmen

```

for  $t \leftarrow 1, \dots, t_{\max}$  do
  for each Ameise  $k = 1, \dots, m$  do
    Wähle Ausgangsstadt;
    for each unbesuchte Stadt  $i$  do
      Wähle Stadt zufällig gemäss  $p_{ij}^k$ ;
      Trage Pheromonspur auf Pfad auf;
    Verdampfe Pheromone;
  
```

Figure 7.2: Ameisenalgorithmus - Schritte

Algorithmus - Schritte

Entscheidung für nächste Stadt

- Jede Ameise besitzt eine Liste mit gültiger Nachbarschaft N
- Die Entscheidung in einem Knoten bzgl. der nächsten Stadt fällt gemäß folgender Formel:

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in J_i^k} [\tau_{il}(t)]^\alpha \cdot [\eta_{il}]^\beta} & \text{für } j \in J_i^k \\ 0 & \text{für } j \notin J_i^k \end{cases}$$

- α und β steuern das Verhältnis zwischen Anteil der Pheromone und lokaler Information
- für α hat man den klassischen Greedy Ansatz

8 Locomotion

8.1 Fortbewegungsarten

- Fortbewegungsarten für Roboter sind oft durch die Natur inspiriert.
- Roboter mit Beinen benötigen in der Regel mehr Freiheitsgrade und sind mechanisch komplexer als Roboter auf Rädern.

8.2 Laufroboter

8.2.1 Vorteile von Laufrobotern

- Können auf unregelmäßigen Untergrund laufen
- Natürliche Fortbewegung
- Keine Umweltveränderung erforderlich

8.2.2 Nachteile von Laufrobotern

- Komplizierte Mechanik
- Anspruchsvolle Steuerungssoftware
- Mehrere Freiheitsgrade pro Bein
- Vielzahl an Sensoren erforderlich
 - interne Sensoren wie Potentiometer, Inkrementalsensoren, ...
 - externe Sensoren wie Stoßdämpfer, Stereo-Kamera, Infrarot, Ultraschall, ...

8.2.3 Freiheitsgrade für Roboterbeine

- Um ein Bein bewegen zu können, sind mindestens zwei Freiheitsgrade notwendig
- Die meisten Laufroboter haben mehrgliedrige Beine mit mindestens drei Freiheitsgraden pro Bein

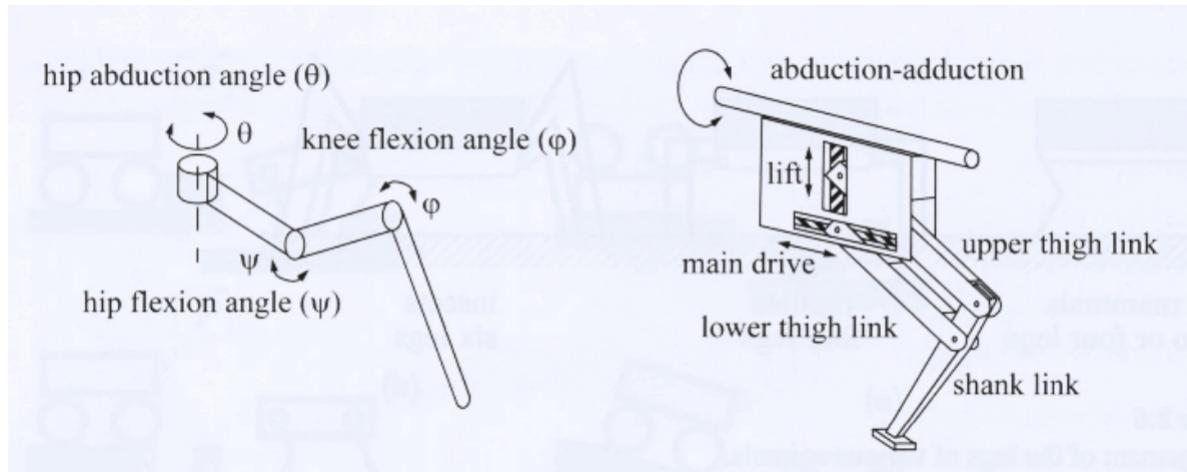


Figure 8.1: Freiheitsgrade für Roboterbeine

8.2.4 Freiheitsgrade für Zweibeiner

- Zweibeiner haben meist sechs Freiheitsgrade pro Bein
- Menschliches Bein hat 7 Freiheitsgrade
- Die Zahl der unterschiedlichen Gangarten hängt von der Zahl der Beine ab.
- Ein Zweibeiner (mit $k = 2$) hat $N = (2 \cdot k - 1)! = 6!$ unterschiedliche Gangarten

8.2.5 Laufverhalten

- Das Geh- und Laufverhalten kann in **statisch und dynamisch stabile Gangarten** eingeteilt werden
- Funktioniert nur für sehr langsame Bewegungen

Definition: Statisches stabiles Gehen bedeutet, dass sich der Roboter zu jedem Zeitpunkt in einem statischen Körperschwerpunkt ist so über den Füßen, dass der Körper nicht fallen kann.

Definition: Dynamisch stabiles Gehen bezeichnet Laufbewegungen, mit weniger als drei Füßen in Kontakt mit dem Boden. Dynamisches Gehen arbeitet mit Körperschwingungen: der Körperschwerpunkt befindet sich die meiste Zeit außerhalb von der aufgespannten Gleichgewichtszone.

8.2.6 Statisch stabiles Gehen

- Statische Stabilitätsbedingung: der Masseschwerpunkt (*Center of Gravity*) ist immer über dem Stützpolygon

Stützpolygon ist ein minimales Polygon, das alle Kontaktstellen des Roboters mit dem Boden enthält.

möglich für alle Beinanzahlen \geq zwei

nur langsame Bewegungen möglich

Dreifußgang

- Je drei Beine sind in der Stemmphase und in der Schwingphase
- Verbindet man die drei Beine in der Stemmphase ergibt sich ein Dreieck

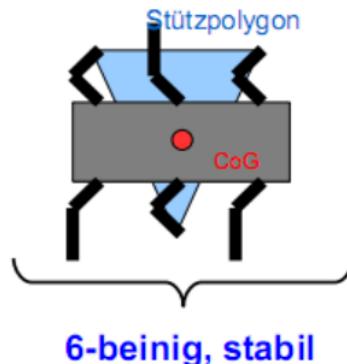


Figure 8.2

8.2.7 Zero Moment Point und Pseudo-Dynamisches Gehen

- Zero Moment Point (ZMP) ist der Punkt auf dem Boden, an dem die Kippmomente null werden
- ZMP dominiert seit 40 Jahren das zweibeinige Gehen
- Bewährt bei glatten Flächen
- Bei zweibeinigen Robotern wird das Stützpolygon ersetzt durch die konvexe Hülle der Bodenkontaktflächen
- **Nachteile des ZMP**
 - ZMP geregelter Gang kann nicht schneller sein als die Sensorik
 - ZMP erfordert die genaue Kenntnis der Massenverteilung im Roboter und alle Gelenkstellungen

8.2.8 Steuerungssoftware

Neuronale Netze

- Ähnlich zu biologischen Vorbildern
- Training durch Simulation

Mechatronik und Regelung

- Regelung von Gelenkwinkel und -Geschwindigkeiten
- Trajektorienplanung und -interpolation
- Explizite regelbasierte Laufplanung
- **Vorteil:** Nutzung gut bekannter mechatronischer Prinzipien

- Eigenschaften wie Stabilität sind gut bekannt
- Lernen einfacher Steuerungszusammenhänge im Gegensatz zu Neuronalen Netzen nicht notwendig

Verhaltensbasierte Steuerungen

- Modellierung von Basisverhalten durch Aufbau von Sensor/Motor-Verbindungen
- Zerlegung von komplexem Verhalten zu einer Vielzahl einfacher Ebenen mit zunehmend abstraktem Verhalten
- Verhalten können durch ein überlagertes Verhalten beeinflusst werden
- Mehrere Basisverhalten können zusammen zu einem völlig neuartigem Verhalten führen
- **Jedoch:** Schwierigkeiten bei der Realisierung komplexer Verhalten.

8.3 Radroboter

8.3.1 Stabilität von Radrobotern

- Natürliche Bewegungen wie Kriechen, Laufen, Springen oder Gehen technisch schwer zu imitieren.
- Mehrheit der mobilen Roboter auf Rädern oder Raupen unterwegs
- Alle Räder haben in der Regel Kontakt zum Untergrund
- für **statisch stabile Fahrzeuge** braucht man mindestens drei Räder:

Zwei angetriebene Räder auf einer Achse mit einem oder zwei passiv mitlaufenden Stützrädern, die sich frei drehen können

Steuerung erfolgt durch verschiedene Geschwindigkeit angetriebener Räder

kinematisches Zentrum oft in der Mitte des Fahrzeugs

- **dynamische Stabilität erfordert Bewegung**

8.4 Kinematik mobiler Radroboter

8.4.1 Holonomische Bewegung

Eine Bewegung heißt **holonomisch**, wenn das Objekt seine **Orientierung und seine Position unabhängig voneinander ändern** kann.

Pose System

- Position eines Objekts auf einer Ebene kann mittels zweier Koordinaten angegeben werden
- Die absolute Orientierung des Objektes wird mit einer dritten Koordinate angegeben, was zusammen ein Pose-System ergibt
- Koordinaten des Pose Systems \Rightarrow 2 für die Position in x,y Ebene; 1 für die absolute Orientierung

8.4.2 Kinematik und Positionsveränderung für Zweiradantrieb

- **Voraussetzung:** kreisförmiger Roboter mit Zweiradantrieb, Bewegung ist nur in der Ebene möglich
- Position: Koordinaten x, y , Orientierung α
- Roboter misst zu diskreten Zeitpunkten t den mit dem linken bzw. mit dem rechten Rad zurückgelegten Weg
- ΔL und ΔR werden von Radwegsensoren ermittelt
- Der Drehwinkel $\Delta\alpha$ ist die Wegdifferenz des rechten und des linken Rades geteilt durch den Radabstand D

$$\Delta\alpha = \frac{\Delta R - \Delta L}{D}$$
- Die Wegstrecke ist der mittlere Weg

$$\Delta s = \frac{\Delta L + \Delta R}{2}$$
- Positionsänderung bei Geradeausfahrt mit $\Delta L = \Delta R$

$$\Delta x = \Delta R * \cos(\alpha)$$

$$\Delta y = \Delta L * \sin(\alpha)$$

8.4.3 Fortbewegung bei Omnidirektionalem Antrieb

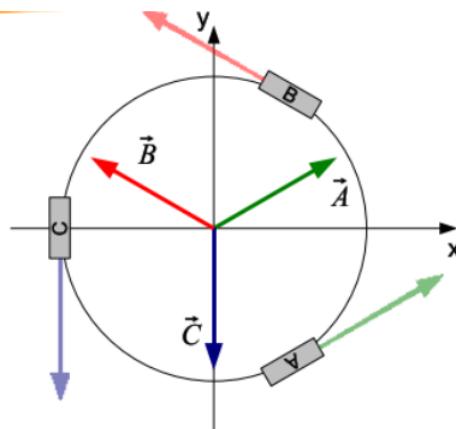


Figure 8.3: Einheitsvektoren

- **Einheitsvektoren** geben die Richtungen an, in denen die Omniwheels von den drei Motoren angetrieben werden.
- **Geschwindigkeitsvektoren** sind die Einheitsvektoren
- Raddrehmomente α, β, γ proportional zur Winkelgeschwindigkeit des jeweiligen Rades
- **Bewegungsvektor** $\vec{V} = \alpha \cdot \vec{A} + \beta \cdot \vec{B} + \gamma \cdot \vec{C}$

9 Industrierobotik

9.1 Industrieroboter

- **Roboterarm** mit Achsen, Gelenken und Antrieben
- Steuerung in einem **Steuerschrank** mit **Bedienerkonsole**
- **Handsteuergerät**

9.1.1 Einsatz

- In wohldefinierter Umgebung, definierte Hindernisse, bekannte Umwelt
- Größtes Einsatzgebiet in der Automobilindustrie und deren Zulieferern (*Schweißen, Lackieren, Beschichten ...*)

9.1.2 Trends

- Industrie 4.0, KI, IoT
- Human-Robot collaboration
- Mobile Roboter und einfachere Bedienung

9.2 Bauform und Komponenten

Roboterarm

- Armelemente bzw. Glieder die über Bewegungssachsen miteinander verbunden sind.
- Ausgleichszylinder ⇒ Motoren werden nicht überlastet, geringere Kräfte sind nötig

Endeffektor, Hand

- Das Arbeitsorgan ⇒ Werkzeug
- Üblicherweise wird die Werkzeugspitze **Tool Center Point (TCP)** genannt

Fahrzeug

- optional
- meist bodengebunden
- ein bis drei Freiheitsgrade
- Bsp.: Verfahrschlitten

9.3 Freiheitsgrade

9.3.1 Definition

Der Freiheitsgrad (DOF) bezeichnet in der Physik die Anzahl der frei wählbaren, voneinander unabhängigen Parameter eines physikalischen Systems, die dessen Zustand eindeutig kennzeichnen.

- In der Mechanik drückt der DOF die Möglichkeit aus im Raum voneinander unabhängig Bewegungen auszuführen
- Anzahl der Freiheitsgrade drückt die translatorischen und rotatorischen Bewegungsmöglichkeiten eines Körpers aus
- Anzahl der Gelenkkachsen \neq Anzahl der Freiheitsgrade

9.4 Bewegungsachsen

9.4.1 Rotatorisch

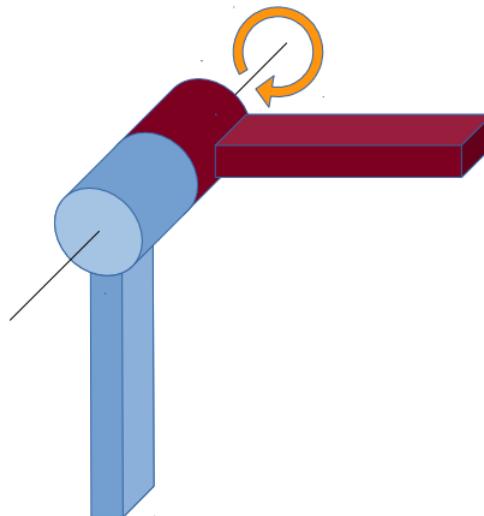


Figure 9.1: Rotatorische Achse

- Für Drehbewegungen
- Freiheitsgrad: 1

9.4.2 Translatorisch

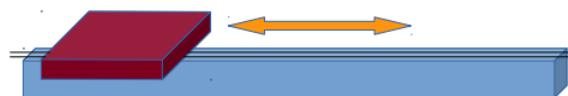


Figure 9.2: Translatorische Achse

- Für Schubbewegungen
- Freiheitsgrad: 1

Hauptachsen

- Zum Positionieren des Effektors im Raum
- Beeinflussen Position des TCP
- Rotatorisch oder Translatorisch

Nebenachsen

- Kopf- oder Handachsen
- Rufen nur kleine Positionsänderungen hervor
- Orientierung des Werkzeugs
- Meist rotatorisch

9.4.3 Rotationsgelenk

Die Drehachse bildet einen rechten Winkel mit den Achsen der beiden angeschlossenen Glieder.

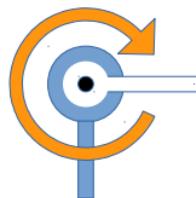


Figure 9.3: Rotationsgelenk

9.4.4 Torsionsgelenk

Die Drehachse verläuft parallel zu den Achsen der beiden Glieder.

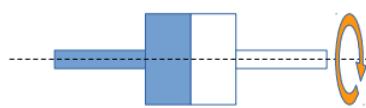


Figure 9.4: Torsionsgelenk

9.4.5 Revolvergelenk

Das Eingangsglied verläuft parallel zur Drehachse, das Ausgangsglied steht im rechten Winkel zur Drehachse.

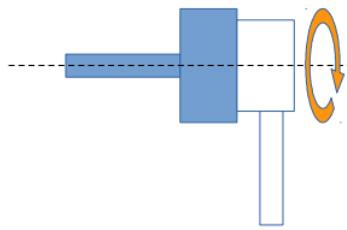


Figure 9.5: Revolvergelenk

9.4.6 Kugelgelenk

Besitzt 3 Freiheitsgrade, wobei die Beweglichkeit stark eingeschränkt ist. Arbeitsbereich der X- und Y-Achse wird durch die Konstruktion auf unter 180 grad limitiert. Die Einkerbung an der Vorderseite lässt an dieser stelle einen 90 grad Winkel zu.



Figure 9.6: Kugelgelenk

9.5 Arbeitsraum

Definition Punkte im 3D-Raum, die von der Roboterhand angefahren werden können.

9.6 Grundtypen von Industrierobotern

9.6.1 Portalroboter

- Steife Struktur \Rightarrow sehr große Arbeitsräume möglich
- Sehr hohe Genauigkeit
- Einfache Steuerung in kartesischen Koordinaten

9.6.2 Horizontal-Knickarmroboter

- **SCARA:** Selective Compliance Assembly Robot Arm
- Aufbau: Ähnlich des menschlichen Arms, horizontaler Gelenkarmroboter
- In der Regel 4 Achsen und Freiheitsgrade $f = 4$

- Hohe Steifigkeit in vertikaler Richtung
- Hohe Geschwindigkeiten möglich

9.6.3 Vertikal-Knickarmroboter

- Klassischer, universell einsetzbarer Industrieroboter
- 4-6 rotatorische Achsen und Freiheitsgrade
- Drei Hauptachsen und drei Nebenachsen

9.6.4 Parallele Roboter

- Bekannt von Flugsimulatoren, Nachbildung der Beschleunigung durch Kippen einer Plattform gegen den Erdbeschleunigungsvektor
- Geschlossene Führungsketten mit fest montierten Antrieben
- Paralleler Roboter, da die Antriebsachsen parallel auf den Endeffektor wirken

Hexapod

- Hohe Stabilität aber geringe Geschwindigkeit
- Aus 6 Linearachsen aufgebaut

Delta-Roboter

- Schnelle Handhabung
- Drei bis vier Gelenkachsen mit stationären Antrieben
- Evtl. Erweiterung mit einer 3-Achs Hand

9.6.5 Leichtbauroboter

- Typ. Massenverhältnis Industrieroboter Last/Eigenmasse: 1:10
- DLR Leichtbauroboter III: Last/Eigenmasse = 1:1

9.7 Effektoren

9.7.1 Endeffektoren

- Arbeitsorgan am Ende eines Roboterarms, mit dem direkt auf ein Werkstück eingewirkt werden kann.
- Angebracht an der entferntesten Stelle einer kinematischen Kette
- **Typen:**

Werkzeuge

Greifer

Prüfmittel

- Zustände der Effektoren werden mit Sensoren erfasst.

9.7.2 Greifersysteme

Ein Griff muss **stabil** sein und **kollisionsfrei** ausgeführt werden können. Objekt darf nicht im Greifer abgleiten oder sich verschieben.

Typen

- **Mechanisch**
- **Vakuum**
- **Magnetisch**

9.7.3 Greifplanung

Ein technischer Griff unterteilt sich in die Sequenzen:

- Herstellen des Kontakts zwischen Greifobjekt und Greifer
- Sichern des Haltens während der Bewegung in allen Richtungen
- Genaues Ablegen des Objekts in der Zielposition und in kürzester Zeit

9.7.4 Greifprinzipien

Formschlüssiger Griff

- Das Objekt liegt lose zwischen den Greifbacken
- Geringe Kräfte auf das Objekt
- Positionserhaltung durch passende Geometrie der Greifbacken

Stoffschlüssiger Griff

- Zwischen Greifer und Objekt wirken Kräfte in Form von Klebstoffen oder Flüssigkeitsbrücken

Kraftschlüssiger Griff

- Kontakt zwischen Greifer und Objekt durch Punkt- oder Flächenkräfte
- Reibkräfte, Vakuum oder Magnetkräfte
- Aneinander pressen, so dass die Reibungskräfte ein Verschieben der Bauteile gegeneinander verhindern

9.8 Antrieb

9.8.1 Antriebsarten

- Antrieb muss Kräfte und Momente durch das Gewicht der Roboterarme und der Objekte im Effektor kompensieren
- Antrieb besteht aus Motor, Getriebe und Wegmesssystem
- Drei Antriebsarten:
 - pneumatisch
 - hydraulisch
 - elektrisch

9.9 Sensor

Umwandeln einer mechanischen, physikalischen oder chemischen Größe in ein Signal. Gegenteil eines Aktors.

9.9.1 Interne Sensoren

Erfassen interner Zustände des Roboters

- Weg- und Winkelmessung
- Geschwindigkeit
- Batteriespannung

9.9.2 Externe Sensoren

Erfassen von Eigenschaften aus der Umwelt des Roboters.

Passive Sensoren

- Keine störenden Einflüsse in der Umwelt, dafür aber ungenau und umgebungsabhängig

Aktive Sensoren

- Genaue Messungen unter wohldefinierten Bedingungen dafür aber evtl. störende Einflüsse

9.10 Kinematik

Definition Die Kinematik beschreibt nur wie sich ein Körper bewegt und wird daher auch als Bewegungsgeometrie bezeichnet.

9.10.1 Kinematikmodul

- Das Kinematikmodul ist für die Positionierung der Gelenke
- Grundaufgaben

Vorwärtsrechnung: Angeben der Gelenkwinkel durch den Benutzer

Rückwärtsrechnung: Angeben der Pose durch den Benutzer, die der Roboter dann anfährt

Teachen von Punkten und Bahnen

9.10.2 Steuerung und Regelung von Industrierobotern

- Robotersteuerung = Hard- und Software zum korrekten Ansteuern des Manipulators
- Zwei Komponenten: Bewegungssteuerung und Gelenkregelung
 - Steuerung von Effektoren, Berechnung, Steuerung und Überwachung (**Bewegungssteuerung**)
 - Regelung von Position, Geschwindigkeit und Beschleunigung (**Gelenkregelung**)

Anforderung an die Regelung

- Möglichst schnelle Armbewegung
- Kein Abdriften im Ziel
- Kein Überschwingen in Kurven oder im Ziel

9.10.3 Punkt-zu-Punkt Bewegung

Beschreiben einer undefinierten Bahn, die aber bei jeder Wiederholung identisch sein wird.
Schnellste Bewegung zwischen zwei Punkten

Asynchrone PTP: Vollständig unabhängiges Verfahren; Achsen erreichen zu verschiedenen Zeiten das Ziel

Synchrone PTP: Leitachse = Achse mit größter Bahndauer; Geschwindigkeit der anderen Achsen werden so vermindert, dass alle Achsen gleichzeitig im Ziel angekommen; Ziel: Reduktion der mech. Belastung

Vollsynchronre PTP: Gleiche Bahn-, Beschleunigungs- und Bremszeiten für jedes Gelenk.

Linearbewegung

- Angabe von Anfangs- und Endpunkt
- Kürzeste Verbindung von zwei Punkten, aber nicht die Schnellste.

Kurvenbahn

- Angabe von Anfangs-, Zwischen- und Endpunkt
- 3 Stützpunkte, die exakt angefahren werden, beschreiben eine eindeutige Kurvenbahn.
- Exakte Kreisbahn kann problemlos beschrieben werden

9.10.4 Überschleifen

- Mechanischer Verschleiß wird verringert
- Reduktion des Energieverbrauchs

Geschwindigkeitsüberschleifen Das Verfahren zum nächsten Bahnsegment wird erst begonnen, wenn eine definierte Geschwindigkeit überschritten wird.

Positionsüberschleifen Das Verfahren zum nächsten Bahnsegment wird erst begonnen, wenn eine definierte Entfernung zum Zwischenpunkt unterschritten wird.

Positioniergenauigkeit Abweichung der Ist-Position von der Soll-Position: Zielsicherheit; absolutes Genauigkeitsmaß

Wiederholgenauigkeit Streuung der Ist-Position bei mehrmaliger Anfahrt aus derselben Richtung; relatives Genauigkeitsmaß.

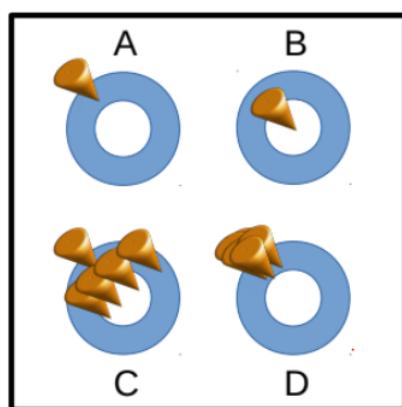


Figure 9.7: Absolut- vs. Wiederholgenauigkeit. A = schlechte AG, B = gute AG, C = schlechte WG, D = gute WG \Rightarrow Kombination aus B und D

9.10.5 Programmierung

Direkte Verfahren - Online Programmierung

- Teach-In Verfahren
- Play-back Verfahren; Programmierer fährt eine Bahn ab, die dann vom Roboter wiederholt wird;
- Master-Slave System; Bediener führt einen Masterarm; Bewegung wird vom Slave simultan kopiert
- CAD-Basiert

Indirekte Verfahren - Offline Programmierung

- Textuelle Programmierung

- Graphische Simulation auf Basis CAD
- Akustische Programmierung, daher Sprachsteuerung
- Aufgabenorientierte Programmierung; Roboter kann Aufgaben erledigen; Programmierer startet Aufgaben

9.11 Kinetic - Berechnungen

9.11.1 Rotationsmatrizen

$$R_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix}$$

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix}$$

Figure 9.8

Rotationsmatrix:

$${}^j R_i = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

ZYX-Euler Winkel (α, β, γ) :

$${}^j R_i = \begin{pmatrix} c(\alpha)c(\beta) & c(\alpha)s(\beta)s(\gamma) - s(\alpha)c(\gamma) & c(\alpha)s(\beta)c(\gamma) + s(\alpha)s(\gamma) \\ s(\alpha)c(\beta) & s(\alpha)s(\beta)s(\gamma) + c(\alpha)c(\gamma) & s(\alpha)s(\beta)c(\gamma) - c(\alpha)s(\gamma) \\ -s(\beta) & c(\beta)s(\gamma) & c(\beta)c(\gamma) \end{pmatrix}$$

Quaternionen (e_0, e_1, e_2, e_3) :

$${}^j R_i = \begin{pmatrix} 1 - 2(e_2^2 + e_3^2) & 2(e_1e_2 - e_0e_3) & 2(e_1e_3 + e_0e_2) \\ 2(e_1e_2 + e_0e_3) & 1 - 2(e_1^2 + e_3^2) & 2(e_2e_3 - e_0e_1) \\ 2(e_1e_3 - e_0e_2) & 2(e_2e_3 + e_0e_1) & 1 - 2(e_1^2 + e_2^2) \end{pmatrix}$$

Figure 9.9

ZYX-Euler Winkel (α, β, γ):

$$\beta = \text{Atan2}(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2})$$

$$\alpha = \text{Atan2}\left(\frac{r_{21}}{\cos(\beta)}, \frac{r_{11}}{\cos(\beta)}\right)$$

$$\gamma = \text{Atan2}\left(\frac{r_{32}}{\cos(\beta)}, \frac{r_{33}}{\cos(\beta)}\right)$$

Quaternionen (e_0, e_1, e_2, e_3):

$$e_0 = \frac{1}{2}\sqrt{1 + r_{11} + r_{22} + r_{33}}$$

$$e_1 = \frac{r_{32} - r_{23}}{4e_0}$$

$$e_2 = \frac{r_{13} - r_{31}}{4e_0}$$

$$e_3 = \frac{r_{21} - r_{12}}{4e_0}$$

Figure 9.10

Umwandlung ZYX-Euler Winkel in Quaternionen und zurück

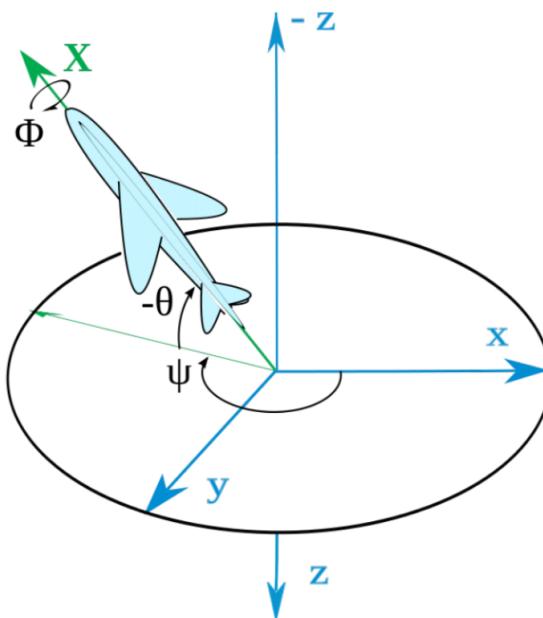


Figure 9.11

Euler Winkel Die Pose des Effektors besteht aus:

- XYZ-Koordinaten
- Orientierung (z.B. im Euler Format)

Homogene Transformation //TODO

9.11.2 Vorwärts Transformation

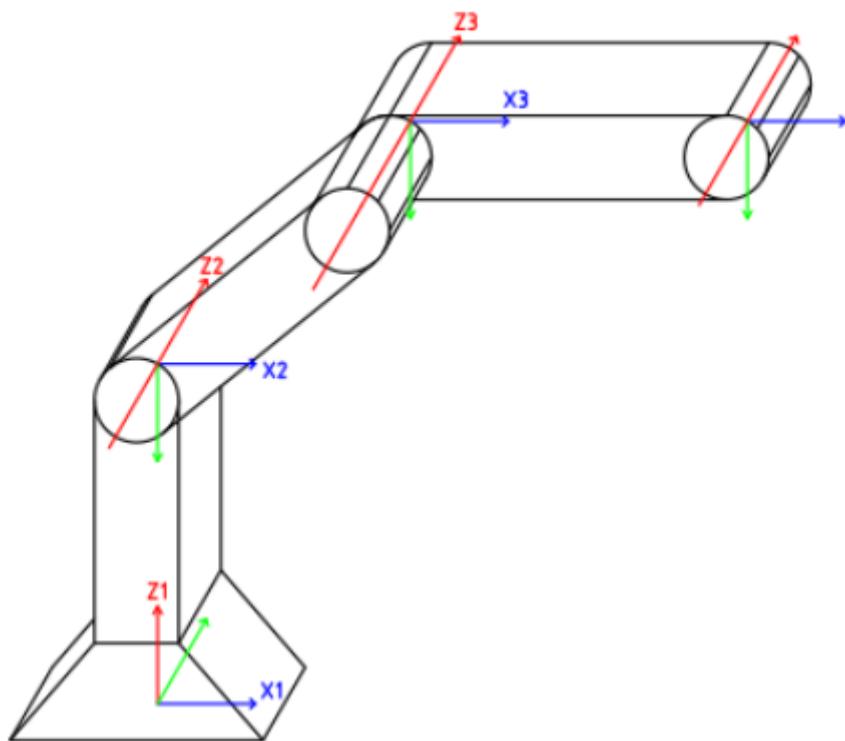


Figure 9.12: Franzkescher 3-Achs Roboter

Bestimmung der Position und Orientierung des Endeffektors anhand der Achswinkel Annahmen:

- XYZ_0 liegt auf XYZ_1
- Die z_i -Achse verläuft entlang der Gelenkachse
- Die x_{i-1} entlang der Normalen zwischen z_{i-1} und z_i
- Die Koordinatensysteme werden seriell transformiert
- ${}^0T_3 = {}^0T_1 \times {}^1T_2 \times {}^2T_3$

9.11.3 Modifizierte Denavit-Hartenbert Transformation

TODO Really?

9.11.4 Inverses kinematisches Problem

Es gibt mehrere Achsstellungen, die zur gleichen Pose führen würden.

9.11.5 Bestimmung der Achswinkel anhand der Position und Orientierung des Endeffektors

- Bei Robotern mit mehr als sechs Freiheitsgraden ist jede Stellung mehrdeutig.

- Es muss geprüft werden, ob eine gewünschte Position erreichbar ist:
 - Stellung nicht erreichbar, da Punkt im Raum zu weit entfernt ist.
 - Unzulässige Positionen, die außerhalb eines zulässigen Achsbereichs liegen
 - Kollision mit Objekt
- Singularitäten: Berechnung der Achsstellungen nicht möglich

Konfigurationssingularität Beispielweise kann die Drehung einer Achse durch eine andere kompensiert werden. Daher Abhilfe durch Vermeidung von 180deg Stellungen.

Bewegungssingularität Beispielweise müsste beim Verfahren durch einen Punkt eine Achsdrehung mit unendlich hoher Geschwindigkeit erfolgen.

Algebraische Methoden Finden von passenden Gleichungssystemen für jedes Gelenk, beispielsweise der Form: $A\cos(\theta_i) + B\sin(\theta_i) + C = 0$ $A\sin(\theta_i) - B\cos(\theta_i) + D = 0$

Geometrische Methoden Lässt sich in der Robotergeometrie ein Punkt finden, an dem Position und Orientierung getrennt betrachtet werden können.

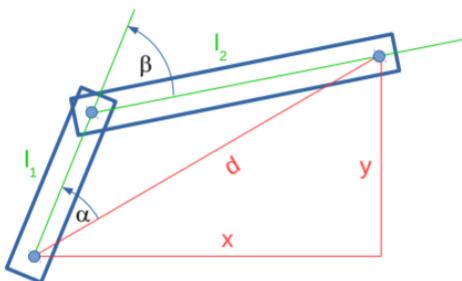


Figure 16: Beispiel zur geometrischen Rückwärtsrechnung mit zwei Achsen

Lösung mit Hilfe des Cosinussatz:

$$a^2 = b^2 + c^2 - 2bc \cos(\alpha)$$

$$b^2 = a^2 + c^2 - 2ac \cos(\beta)$$

Hier:

$$l_2^2 = l_1^2 + d^2 - 2l_1d \cos(\alpha)$$

$$l_1^2 = l_2^2 + d^2 - 2l_2d \cos(\alpha')$$

Mit α' liegt gegenüber α
 $\beta' = 180^\circ - \alpha - \alpha'$

Figure 9.13

Voraussetzungen für die Anwendung dieser Methoden

- drei aufeinander folgende, sich in einem Punkt schneidende Rotationsachsen
- drei aufeinander folgende parallele Rotationsachsen (z.B. SCARA)

Numerische Methoden

- Berechnung der Achstellungen mit Hilfe von Näherungsverfahren
- Von einem Startpunkt müssen die Koordinaten und Gelenkkoordinaten bekannt sein

- Mit Hilfe der **Jacobi-Matrix** lässt sich anschließend die Gelenkkonfiguration eines naheliegenden Punktes bestimmen
- Verfahren zur Rückwärtsrechnung in Echtzeit eher ungeeignet

9.11.6 Koordinatensysteme

Alle Koordinatensysteme werden als kartesische Rechtssysteme angenommen.

- Weltkoordinaten ⇒ Fest mit der Welt verbunden
- Basiskoordinaten ⇒ Fest mit dem Sockel des Roboters verbunden.
- Handflanschkoordinaten ⇒ Relativ zu den Basiskoordinaten
- Werkzeugkoordinaten ⇒ Relativ zu den Handflanschkoordinaten
- Anwenderkoordinaten ⇒ Relativ zu Welt- oder Basiskoordinaten
- Werkstückkoordinaten ⇒ Mit Werkstück verbunden, relativ zu Anwenderkoordinaten

9.12 Safety

9.13 Roboterprogrammierungen

9.13.1 Programmierung in RAPID

Ein RAPID Programm besteht aus einem oder mehreren Tasks:

- Mindestens ein **Motion Task**
- Eventuell zusätzliche Background Tasks

Konfigurationsmöglichkeiten für einen Task

- **Task in foreground:** Der Task wird erst ausgeführt, wenn der Eltern-Task im Zustand Idle ist
- **Type:**

NORMAL: Task reagiert auf START/STOP und NOT-Aus

STATIC: Läuft bei Neustart an der aktuellen PZ-Position weiter; Reagiert nicht auf START/STOP und NOT-Aus

SEMISTATIC: PZ Rücksetzen bei Neustart; Sonst wie STATIC

9.13.2 Syntax

```

MODULE HAWMain

    PROC main()
        VAR num chosenFunction; !Var decl. am Anfang!!!
        ...
    ENDPROC
    ...
ENDMODULE

```

Figure 9.14

Routinen können Prozeduren, Funktionen oder Traps sein.

Parameterübergabe

- Reguläre Par: **PROC moveToXY(num par1)**
- Optionale Par.: **PROC moveToXY(
num par1)**

Switches

- Switches sind immer optional
- Mehrere Switches sind möglich

```

PROC SwitchTest(\switch on \switch off \switch middle)
    IF Present(middle) THEN
        TPWrite "Middle on";
    ENDIF
    IF Present(off) THEN
        TPWrite "Off on";
    ENDIF
    IF Present(on) THEN
        TPWrite "On on";
    ENDIF
ENDPROC

```

Figure 9.15

Prozedurdeklaration

```

PROC arrayCopy(VAR num arr1{*}, VAR num arr2{*})
    FOR i FROM 1 TO dim(arr1, 1) DO
        arr2{i} := arr1{i};
    ENDFOR
ENDPROC

```

Figure 9.16

Funktionsdeklaration

```

FUNC num sum(num arg1, num arg2)
    RETURN arg1 + arg2;
ENDFUNC

```

Figure 9.17

Trapdeklaration

```

TRAP exceptionHandler
    handleExcptionTheRightWay;
    RETURN;
ENDTRAP

```

Figure 9.18

Routinen Aufruf

```

VAR num a1{3} := [1, 2, 3];
VAR num a2{3} := [0, 0, 0];

TPWrite NumToStr(a2{1}, 2);
TPWrite NumToStr(a2{2}, 2);
TPWrite NumToStr(a2{3}, 2);

arrayCopy a1, a2;

TPWrite NumToStr(a2{1}, 2);
TPWrite NumToStr(a2{2}, 2);
TPWrite NumToStr(a2{3}, 2);

TPWrite NumToStr(sum(12,24), 2);

```

Figure 9.19

```

VAR num globaleZahl := 42;
TASK VAR num imTaskGueltigeZahl := 23;
LOCAL VAR num lokaleUnwichtigeZahl := 13;
VAR pos pHome1 := [300, 300, 300];

```

Figure 9.20

Datentypen

- Jeder Datentyp ist Value- oder Non-Value
- Atomarer Datentyp
- Zusammengesetzter Datentyp
- Alias
- Signal - Ausnahme Semi-Value

Abfragen Wertbasiert, daher 0,1

Initialisierung und Zuweisung: Set, Reset, PulseDO, SetGO

Persistente Bsp.: *PERS pos pHome1 := [300, 300, 300];*

- Bleiben auch unter folgenden Umständen erhalten
 - Reboot
 - Open, Close, New Programm
 - Start - Move PP to main
 - Start - Move PP to routine
- Können nur auf Modulebene deklariert werden
- Können auf einen Wert initialisiert werden, aber keine erneute Initialisierung bei Reboot

Konstanten Bsp.: *CONST num euler := 2.7182;*

9.13.3 Bewegungsfunktionen

Move Absolute Joint

- Syntax: MoveAbsJ ToJointPos, Speed, Zone, Tool
- Beispiel: MoveAbsJ pHome1, vMax, fine, tool0;

Move Joint

- Syntax: MoveJ ToPoint, Speed, Zone, Tool
- Beispiel: MoveJ p10, vMax, fine, tool0;

Move Linear

- Syntax: MoveL ToPoint, Speed, Zone, Tool
- Beispiel: MoveL p10, vMax, fine, tool0;

Move Circularly

- Syntax: MoveC CirPoint, ToPoint, Speed, Zone, Tool
- Beispiel: MoveC p10, p20, vMax, fine, tool0;

9.13.4 World Zones

Es ist nötig den Roboter vor Programmausführung in eine definierte Position zu bringen. Programmierer ist für die einschränkung der Bewegungen des Roboters verantwortlich, sodass kein schaden entsteht bspweise durch Kollision. Diese Zone kann sein:

- eine Kugel
- ein Zylinder
- Quader/Box
- festgelegte Bereiche bei den Achswinkeln

Bei Eintritt des Tool Center Points (TCP) in den definierten Weltzonenbereich kann ein Signal gesetzt werden, das mit Hilfe der Software weiterverwendet werden kann. Die benötigten Befehle für eine Kugelbereich sind *WZSphDef* und *WZDOSet*. Das gesetzte Signal ist hier *dolnHome1*, das vorher als IO-Signal definiert wurde.

```
MODULE BGTask
```

```
VAR wzstationary service;
CONST robtarget pHome1:=[[374.00,0.00,630.00],[0.707107,2.28879E-15,0.707107,1.6149E
[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]]];

PROC startup()
    VAR shapedata volume;
    WZSphDef \Inside, volume, pHome1.trans, 50;
    WZDOSet \Stat, service \Inside, volume, dolnHome1, 1;
ENDPROC

ENDMODULE
```

Figure 9.21

9.13.5 Bildschirmein- und -ausgabe

Die TP-Funktionen bieten die Möglichkeit einer einfachen Bildschirmausgabe und eingabe. Mögliche Befehle sind **TPReadDnum**, **TPReadFK**, **TPReadNum** und **TPWrite**.

```

PROC MoveZero()
    VAR num diffTime;
    ClkReset timer1;
    ClkStart timer1;
    MoveAbsJ jzero, vmax, fine, tool0;
    ClkStop timer1;
    diffTime := ClkRead(timer1);
    TPWrite "Gemessene Zeit: " + NumToStr(diffTime, 2) + "s"
ENDPROC

```

Figure 9.22

9.13.6 Zeitmessung

- ClkReset - Zurücksetzen / Löschen eines Timers
- ClkStart
- ClkStop
- ClkRead - Auslesen eines (abgelaufenen) Timers

9.13.7 Bewegung relativ

Um die Anzahl der Teachpunkte möglichst gering zu halten, empfiehlt es sich, wenn möglich, weitere Punkte relativ zu anderen Punkten anzugeben.

⇒ Befehl **Offs**, mit dem die Positionskomponente eines Punktes einfach modifiziert werden kann.
 VAR robtarget p10 := Offs(pHome, X, Y, Z)

10 Lego Mindstorms Code

10.1 SubSumptionMain

```
public class SubsumptionMain {  
    public static void main(final String[] args) throws InterruptedException {  
        final Effector[] effectors = {  
            Effector.make("Lamp"),  
            Effector.make("Chassis")};  
        final Arbitrator arbitrator = new Arbitrator(effectors);  
        int priority = 1;  
        final Behavior[] behaviors = {  
            Behavior.make(Behavior.BLINK, arbitrator, priority++),  
            Behavior.make(Behavior.STEER_RIGHT, arbitrator, priority),  
            Behavior.make(Behavior.STEER_LEFT, arbitrator, priority++),  
            Behavior.make(Behavior.TURN_LEFT, arbitrator, priority++),  
            Behavior.make(Behavior.GO, arbitrator, priority++),  
            Behavior.make(Behavior.STOP, arbitrator, priority++),  
            Behavior.make(Behavior.KILLER, arbitrator, priority++)  
        };  
        final SubSensor[] sensors = {  
            SubSensor.make(SubSensor.BUTTON_SENSOR, behaviors),  
            SubSensor.make(SubSensor.COLLISION_SENSOR, behaviors),  
            SubSensor.make(SubSensor.ULTRASOUND_SENSOR, behaviors)  
        };  
        for (final SubSensor sensor : sensors) {  
            assert sensor != null;  
            sensor.start();  
        }  
        arbitrator.waitUntilDie();  
    }  
}
```

10.2 Arbitrator

```
public class Arbitrator {  
    private final int MAX_PRIO = 10;  
    private final Wish[] wishes = new Wish[MAX_PRIO];  
    private final Effector[] effectors;  
  
    Arbitrator(Effector[] effectors) { this.effectors = effectors; }  
  
    public synchronized void accept(final Wish wish, final int priority) {  
        if (wish == Wish.DIE) {  
            justDieAlready();  
        } else if (wish == Wish.NOTHING) {  
            wishes[priority] = null;  
        }  
    }  
}
```

```

} else {
    wishes[priority] = wish;
    boolean wishHasTopPriority = true;
    for (int p = priority + 1; p < MAX_PRIO; p++) {
        if (wishes[p] != null) {
            wishHasTopPriority = false;
            break;
        }
    }
    if (wishHasTopPriority) {
        for (final Effector effector : effectors) {
            effector.accept(wish);
        }
    }
}
public void justDieAlready() { notifyAll(); }
public void waitUntilDie() throws InterruptedException { wait(); }
}

```

10.3 Wishes

```

public enum Wish {
    FORWARD, STOP, LEDGREEN, LEDOFF, TURN_LEFT, STEER_LEFT, STEER_RIGHT, NOTHING, DIE
}

```

10.4 Behaviors

```

public abstract class Behavior {
    private final Arbitrator arbitrator;
    private final int priority;
    private final Reading type;
    private int readingValue;

    public Behavior(final Arbitrator arbitrator, final int priority, final Reading
        type) {
        this.arbitrator = arbitrator;
        this.priority = priority;
        this.type = type;
    }

    public synchronized void accept(final Reading type, final int readingValue) {
        if (this.type == type) {
            this.readingValue = readingValue;
            onAccept(readingValue);
        }
    }
}

```

```

abstract void onAccept(final int readingValue);

void sendWish(final Wish wish) {
    //System.out.println("sending wish: " + wish);
    arbitrator.accept(wish, priority);
}

}

```

```

public class Go extends Behavior {

    Go(Arbitrator arbitrator, int priority) {
        super(arbitrator, priority, Reading.Button);
    }

    @Override
    void onAccept(final int buttonValue) {
        if (buttonValue != Button.ID_ENTER) {
            sendWish(Wish.FORWARD);
            sendWish(Wish.NOTHING);
        }
    }
}

```

```

public class Killer extends Behavior {

    public Killer(final Arbitrator arbitrator, final int priority) {
        super(arbitrator, priority, Reading.Button);
    }

    @Override
    void onAccept(final int readingValue) {
        if (readingValue == Button.ID_ESCAPE) {
            sendWish(Wish.DIE);
        }
    }
}

```

```

class SteerLeft extends Behavior {

    SteerLeft(Arbitrator arbitrator, int priority) {
        super(arbitrator, priority, Reading.Distance);
    }

    @Override
    void onAccept(final int distanceValue) {
        if (distanceValue == UltrasoundSensor.TOO_CLOSE) {
            sendWish(Wish.STEER_LEFT);
        }
    }
}

```

```

    }
}
}
```

```

class Stop extends Behavior {

    Stop(final Arbitrator arbitrator, final int priority) {
        super(arbitrator, priority, Reading.Button);
    }

    @Override
    void onAccept(final int buttonValue) {
        if (buttonValue == Button.ID_DOWN) {
            sendWish(Wish.STOP);
            sendWish(Wish.LEDOFF);
            sendWish(Wish.NOTHING);
        }
    }
}
```

10.5 Effectors

```

public class Chassis extends Effector {
    private static final double STEERING_FACTOR = 0.5;
    private static final int STEERING_TIME = 40;
    private static final float FRICTION = 1.5f;

    private final RegulatedMotor motorA = new EV3LargeRegulatedMotor(MotorPort.A);
    private final RegulatedMotor motorD = new EV3LargeRegulatedMotor(MotorPort.D);

    @Override
    public void accept(@NotNull Wish command) {
        if (command == Wish.FORWARD) {
            forward();
        } else if (command == Wish.STOP) {
            stop();
        } else if (command == Wish.TURN_LEFT) {
            turnLeft();
        } else if (command == Wish.STEER_LEFT) {
            steerLeft();
        } else if (command == Wish.STEER_RIGHT) {
            steerRight();
        }
    }

    private void steerRight() {
        //System.out.println("steerRight");
        int originSpeed = motorA.getSpeed();
        motorA.setSpeed((int) (originSpeed * STEERING_FACTOR));
```

```

try {
    Thread.sleep(STEERING_TIME);
} catch (InterruptedException e) {
    e.printStackTrace();
}
motorA.setSpeed(originSpeed);
}

private void steerLeft() {
//System.out.println("steerLeft");
int originSpeed = motorD.getSpeed();
motorD.setSpeed((int) (originSpeed * STEERING_FACTOR));
try {
    Thread.sleep(STEERING_TIME);
} catch (InterruptedException e) {
    e.printStackTrace();
}
motorD.setSpeed(originSpeed);
}

private void stop() {
motorA.stop();
motorD.stop();
}

private void forward() {
motorA.forward();
motorD.forward();
}

private void turnLeft() {
motorA.backward();
motorD.backward();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

motorA.rotate((int) (360 * FRICTION));
motorD.stop();

forward();
}
}

```

```

class Lamp extends Effector
{
@Override
public void accept(Wish command)
{

```

```

switch (command){
    case LEDGREEN:
        Button.LEDPattern(1);
        break;
    case LEDOFF:
        Button.LEDPattern(0);
        break;
    default: //not my command
        break;
}
}
}

```

10.6 Sensors

```

public abstract class SubSensor extends Thread {

    public static final String BUTTON_SENSOR = "ButtonSensor";
    public static final String COLLISION_SENSOR = "CollisionSensor";
    public static final String ULTRASOUND_SENSOR = "UltrasoundSensor";
    private final @NotNull Behavior[] behaviors;

    private final @NotNull Reading type;

    public SubSensor(@NotNull Behavior[] behaviors, @NotNull Reading
        type) {
        this.behaviors = behaviors;
        this.type = type;
        setDaemon(true);
    }

    void send(@NotNull int value) {
        for (@NotNull Behavior behavior : behaviors) {
            behavior.accept(type, value);
        }
    }
}

```

```

public class ButtonSensor extends SubSensor {

    public ButtonSensor(@NotNull Behavior[] behaviors) {
        super(behaviors, Reading.Button);
    }

    public void run() {
        while (true) {
            int button = Button.waitForAnyPress();

```

```

        send(button);
    }
}

```

```

public class CollisionSensor extends SubSensor {
    public static final int COLLISION = 1;
    public static final int NO_COLLISION = 0;
    private final float[] floats;

    private SensorModes sensor1 = new EV3TouchSensor(SensorPort.S1);
    private SampleProvider touch1 = sensor1.getMode("Touch");
    private SensorModes sensor2 = new EV3TouchSensor(SensorPort.S4);
    private SampleProvider touch2 = sensor2.getMode("Touch");

    public CollisionSensor(Behavior[] behaviors) {
        super(behaviors, Reading.Collision);
        floats = new float[1];
    }

    public void run() {
        while (true) {
            touch1.fetchSample(floats, 0);
            if (floats[0] == COLLISION) {
                send(COLLISION);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            touch2.fetchSample(floats, 0);
            if (floats[0] == COLLISION) {
                send(COLLISION);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

```

public class UltrasoundSensor extends SubSensor {
    private final float[] floats;
    public static final int TOO_CLOSE = 0;
    public static final int PERFECT = 1;
    public static final int TOO_FAR = 2;

```

```

public static final int NEAR_BORDER = 5;
public static final int DISTANT_BORDER = 8;
public static final int SCALE_TO_CM = 100;

private SensorModes sensor3 = new EV3UltrasonicSensor(SensorPort.S3);
private SampleProvider ultrasonic = sensor3.getMode("Distance");

public UltrasoundSensor(Behavior[] behaviors) {
    super(behaviors, Distance);
    floats = new float[1];
}

public void run(){
    float distance = 10;
    float sample[] = new float[ultrasonic.sampleSize()];
    while (true){
        ultrasonic.fetchSample(sample, 0);
        distance = sample[0] * SCALE_TO_CM;

        if (distance <= NEAR_BORDER) {
            send(TOO_CLOSE);
        } else if (distance <= DISTANT_BORDER) {
            send(PERFECT);
        } else {
            send(TOO_FAR);
        }

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```
