

# Contents

<b>1 Geschicht</b>	<b>1</b>
1.1 Industrieroboter . . . . .	1
1.2 Serviceroboter . . . . .	1
<b>2 Software-Architekturen für mobile Robotersysteme</b>	<b>2</b>
2.1 Probleme und Anforderungen . . . . .	2
2.1.1 Umgebung mobiler Roboter . . . . .	2
2.1.2 Roboterkontroll-Architekturen . . . . .	2
2.1.3 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 . . . . .	4
2.2.4 Probabilistische Robotik . . . . .	5
2.2.5 Subsumption-Architektur in Bezug auf die Anforderungen des Robotersteuerungssystems . . . . .	5
2.3 Robot Operating System (ROS) . . . . .	6
2.3.1 Design Prinzipien . . . . .	6
2.3.2 Asynchrone Kommunikation . . . . .	7
2.3.3 Sychrone Kommunikation . . . . .	7
<b>3 Lokalisation autonomer mobiler Robotersysteme</b>	<b>8</b>
3.1 Varianten der Selbstlokalisierung . . . . .	8
3.2 Relative Lokalisierung versus Absolute Lokalisierung . . . . .	9
3.3 Transformation von Koordinatensystemen lokale ↔ globale . . . . .	9
3.4 Karten für statistische und dynamische Umgebungen . . . . .	10
3.4.1 Mapping Methoden . . . . .	10
3.4.2 Arten von Modellen . . . . .	11
3.4.3 Kontinuierliche Metrische Karten . . . . .	12
3.4.4 Grid Maps - Rasterkarten . . . . .	12
3.4.5 Adaptive Unterteilung . . . . .	12
3.4.6 Topologische Karten . . . . .	13
3.4.7 Hybrid Maps . . . . .	14
3.5 Passive und Aktive Selbstlokalisierung . . . . .	14
3.6 Landmarken . . . . .	15
<b>4 Fortbewegung, Lokalisierungsalgorithmen</b>	<b>16</b>
4.1 Relative Lokalisierung . . . . .	16
4.1.1 Dead Reckoning . . . . .	16
4.1.2 Odometrie . . . . .	17
4.1.3 2D-Scanmatching . . . . .	18
4.1.4 Weitere Lokalisierungsalgorithmen zur globalen Lokalisierung . . . . .	19

<b>5 Navigation</b>	<b>20</b>
5.1 Bekanntes vs. unbekanntes Terrain . . . . .	20
5.2 Navigation in unbekanntem Terrain . . . . .	20
5.2.1 Konturverfolgung . . . . .	20
5.2.2 Bug1 Algorithmus . . . . .	21
5.2.3 Bug2-Algorithmus . . . . .	21
5.2.4 Bug3-Algorithmus . . . . .	22
5.2.5 Labyrinthe . . . . .	22
5.3 Pfadplanung für mobile Roboter in bekanntem Terrain . . . . .	24
5.3.1 Konfigurationsraum . . . . .	24
5.4 Algorithmen und Methoden . . . . .	25
5.4.1 Dijkstra . . . . .	26
5.4.2 A* . . . . .	26
5.4.3 Wegsuche mit dem Sichtgraph-Algorithmus . . . . .	27
5.4.4 Voronoi-Diagramme . . . . .	29
5.4.5 Navigation in einer Rasterkarte . . . . .	29
5.4.6 Potentialfeld Methode . . . . .	30
<b>6 Probabilistische Methoden und Kartierungen</b>	<b>31</b>
6.1 Problemstellung . . . . .	31
6.2 Modellierung von Unsicherheit . . . . .	31
6.3 Umgebungsmodellierung mit Occupancy Grids . . . . .	31
6.3.1 Satz von Bayes . . . . .	31
6.3.2 Evidence Grids (Beweisraster) . . . . .	31
6.4 Bayes-Filter Algorithmus . . . . .	32
6.4.1 Algorithmus . . . . .	32
6.4.2 Beispiel Aufgabenstellung . . . . .	33
6.4.3 Beispiel Rechnung . . . . .	34
6.5 Markov Lokalisierung . . . . .	34
6.6 Monte Carlo Lokalisierung . . . . .	34
6.6.1 Partikelmengen . . . . .	35
6.7 Kalman-Filter . . . . .	36
6.7.1 Definition . . . . .	36
6.7.2 Vorgehen . . . . .	36
6.7.3 Einschränkungen . . . . .	36
6.8 Simultaneous Localization and Mapping (SLAM) . . . . .	37
6.8.1 Landmarkenbasiertes SLAM Problem . . . . .	37
6.8.2 Problemstellung . . . . .	37
6.8.3 Funktionsweise . . . . .	38
6.8.4 Hinzunahme neuer Landmarken . . . . .	38
6.8.5 Aufbau eines SLAM-Graphen . . . . .	39
6.8.6 Varianten von SLAM . . . . .	39
6.8.7 Bayesian Netzwerk für landmarkenbasiertes SLAM . . . . .	40
<b>7 Schwarmrobotik und Evolutionäre Robotik</b>	<b>41</b>
7.1 Schwärme und deren Verhalten in der Natur . . . . .	41
7.1.1 Computersimulation von Schwärmen - Algorithmus von Craig Reynolds . . . . .	41
7.2 Mechanismen zur Schwarmorganisation . . . . .	42
7.3 Schwarmintelligenz . . . . .	42
7.4 Ameisenalgorithmen . . . . .	43
7.4.1 Optimaler Weg bei futtersbeschaffenden Ameisen . . . . .	43

7.4.2	Ant Colony Optimization Algorithm (ACO) . . . . .	44
7.4.3	Traveling Salesman Problem . . . . .	44
<b>8</b>	<b>Locomotion</b>	<b>46</b>
8.1	Laufroboter . . . . .	46
8.1.1	Freiheitsgrade für Roboterbeine . . . . .	46
8.1.2	Laufverhalten . . . . .	47
8.1.3	Statisch stabiles Gehen . . . . .	47
8.1.4	Zero Moment Point und Pseudo-Dynamisches Gehen . . . . .	48
8.1.5	Steuerungssoftware . . . . .	48
8.2	Radroboter . . . . .	49
8.2.1	Stabilität von Radrobotern . . . . .	49
8.3	Kinematik mobiler Radroboter . . . . .	49
8.3.1	Holonomische Bewegung . . . . .	49
8.3.2	Kinematik und Positionsveränderung für Zweiradandtrieb . . . . .	50
8.3.3	Fortbewegung bei Omnidirektionalem Antrieb . . . . .	50
<b>9</b>	<b>Lego Mindstorms Code</b>	<b>51</b>
9.1	SubSumptionMain . . . . .	51
9.2	Arbitrator . . . . .	51
9.3	Wishes . . . . .	52
9.4	Behaviors . . . . .	52
9.5	Effectors . . . . .	54
9.6	Sensors . . . . .	56

# 1 Geschichte

## 1.1 Industrieroboter

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

Zeichen sich aus durch:

- Schnelligkeit
- Genauigkeit
- Robustheit
- Traglast

Einsatzgebiete sind unter anderen **Schweißen, Kleben, Schneiden, Lackieren**.

Zunehmend **kollaborative** Roboter, Cobots die mit Menschen **ohne Schutzeinrichtungen** im Produktionsprozess interagieren und diese **Wahrnehmen um Verletzungen zu vermeiden**.

## 1.2 Serviceroboter

Ein **Serviceroboter** ist eine **frei programmierbare Bewegungseinrichtung**, die **teil- oder vollautomatisch** Dienstleistungen verrichten.

**Dienstleistungen** sind dabei Tätigkeiten, die **nicht der direkten industriellen Erzeugung** von Sachgütern, sondern Verrichtung von **Leistungen für Menschen und Einrichtungen** dienen. Sie werden in zwei Klassen eingeteilt:

**Professionel** Einsatzbereiche sind beispielsweise:

- Rettung
- Landwirtschaft
- Medizin

**Privat** Einsatzbereiche sind beispielsweise:

- Rasenmäher
- Pfleger

# 2 Software-Architekturen für mobile Robotersysteme

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 Probleme und Anforderungen

### 2.1.1 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** und **Details sind nur während der Ausführung** der Aktion bekannt.

Mobile Roboter müssen in einer geschlossen Regelung:

1. Erfassung der Umgebung mit Sensoren
2. Auswertung der Daten
3. Planung der Aktionen
4. Umsetzung der Aktionen mittels Koordination der Aktuatoren

### 2.1.2 Roboterkontroll-Architekturen

#### Herausforderungen

Robotersysteme bestehen im allgemeinen aus den Gebieten **Wahrnehmung, Planung und Handlung**. Die Herausforderungen die hierdurch entstehen sind:

- Erfassung und Auswertung der Sensorwerte
- Planung der Pfade
- Vermeidung von Hindernissen
- Ausführung komplexer Algorithmen in langen Zeitzyklen

## Probleme bei der Software-Erstellung zur Roboterkontrolle

Roboter sind **eingebettete Systeme**, die in einem geschlossenen Regelkreis die Sensorströme in **Echtzeit verarbeiten**. Dadurch entstehen diese Probleme:

- Unterschiedliche Aufgaben  $\Rightarrow$  Unterschiedliche Zeitzyklen
- Unterschiedlicher Zeitskalen  $\Rightarrow$  Abbildung des Kontroll- und Datenflusses in der Architektur ist nicht standardisiert
- Für etliche algorithmische Teilprobleme sind **keine effizienten Verfahren** bekannt
- **Prozessorkapazität ist begrenzt**

### 2.1.3 Anforderungen an das Kontrollsysteem eines autonomen Roboters

**Robustheit** Die Umgebung des Systems kann sich ständig ändern. Auf diese muss der Roboter sinnvoll reagieren, obwohl die verwendeten Modelle der Umgebung ungenau ist.

**Unterschiedliche Ziele** Roboter verfolgt zu einem Zeitpunkt eventuell in Konflikt stehende Ziele.  
z.B.: Ziel ansteuern und Hinderniss ausweichen

**Sensorwerte von mehreren Sensoren** Sensordaten können verrauscht sein und damit fehlerhafte und inkonsiste Messwerte liefern. z.B. außerhalb seines Bereichs und dies nicht überprüfen kann.

**Erweiterbarkeit** Bei neuen Sensoren, sollte diese 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.

Dieses Modell hat unterschiedliche Namen:

- Sense-Think-Act-Cycle (STAC)
- Sense-Model-Plan-Act (SMPA)

**Sense:** Vorverarbeitung der Daten

**Model:** Konstruktion oder Aktualisierung eines Weltmodells

**Plan:** Alle Entscheidungen basieren auf dem Weltmodell

**Act:** Ausführung der Aktionen und Befehle

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

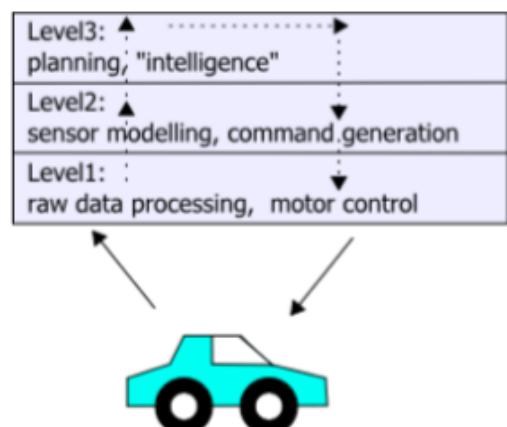


Figure 2.1: Oben nach unten: Planer, Navigator, Pilot

### Nachteile:

- **Sequentieller Ansatz** ⇒ lange Kontrollzykluszeit
- **Gesamtsystem anfällig** ⇒ Modulfehler führt zum scheitern des Gesamtsystems
- Weltmodell muss alle zur Planung notwendigen Informationen enthalten
- Planer hat nur Zugriff auf das Weltmodell ⇒ Während Planung kann sich Umwelt verändert 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 ⇒ SMPA-Anteil

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. ⇒ 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

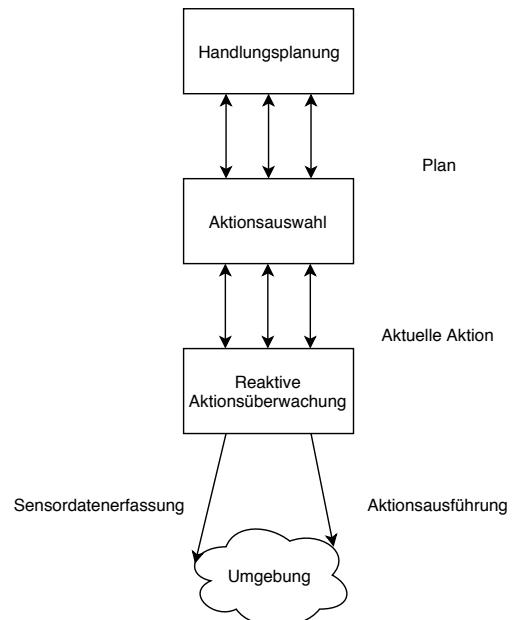


Figure 2.2: Schema des Hybridmodell

## 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

Berücksichtigung der **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  $\Rightarrow$  **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 niedere 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 komplexere Funktionen erweitert werden

## 2.3 Robot Operating System (ROS)

ROS bietet eine Standard für Roboterkontrollsoftware. **DAS Architektschema für Roboterkontrollsoftware** gibt es nicht ⇒ Unterstützung der Softwareentwicklung durch Middleware wie ROS.

**Zweck:** soll die Entwicklung von Software für Roboter vereinfachen und wiederkehrende Aufgaben standardisieren

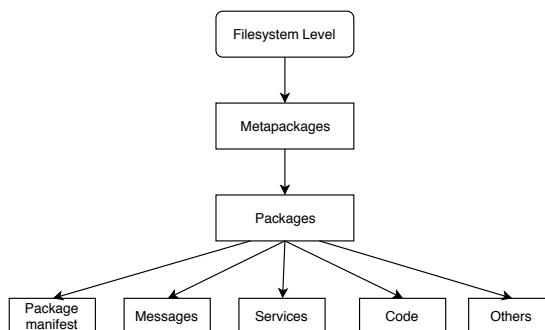


Figure 2.3: Filesystem, das ROS zugrunde liegt

### 2.3.1 Design Prinzipien

In ROS kommunizieren die verschiedenen Nodes über den Master, dabei können die verschiedenen Nodes und der Master auf verschiedenen Rechnern laufen. Einige Bedingung ist, dass die Nodes den Master erreichen können.

**Master** Dient zur Koordinierung und Kommunikation von Nodes. Er ist damit der wichtigste Knoten. Er bietet eine Serviceregistry für synchrone Kommunikation zwischen Nodes und ist Message Broker für asynchronen Nachrichtenaustausch.

**Node** Wird in ROS als Package publiziert. Ein Node muss sich beim Master registrieren und kann über ihn mit anderen Nodes kommunizieren. Ein Node ist in ROS ein Prozess mit einer bestimmten Funktionalität. Ist ein Node fehlerhaft hat dies in der Regel wenig Auswirkungen auf andere Nodes.

### Parameter Server und Konfigurationsdateien

Der Parameter Server auf dem Master enthält **eine Art Wörterbuch für Werte**. Alle Ressourcen wie Nodes, Messages oder Parameter existieren darin in einer hierarchischen Namensstruktur. In ihm werden z.B.: die Konfigurationsdateien gespeichert.

### 2.3.2 Asynchrone Kommunikation

Der asynchrone Nachrichtenaustausch funktioniert über das **Publish-Subscribe** Pattern. Hierbei kann ein Node kann entweder Nachrichten an den Master publiziere, dafür benutzt er ein bestimmtes Topic. Er kann jedoch auch verschiedene Topics abbonieren um Nachrichten zu empfangen sobald diese publiziert werden. Ein Node hat dabei kein Limit an Topics die er nutzen kann. Ein Topic ist hierbei ein einfacher String. Die Kommunikation findet dabei über TCP/IP Socket stat.

```

Node2                                         Master
|           register publisher
|----- topic: "farbsensor" ----->
|
|
Node1                                         Master
|           subscribe
|----- topic: "farbsensor" ----->
|
|
Node2                                         Master
|           publish
|-- topic: "farbsensor" message: { r: 0.1, g: 0.1, b: 0.1 } -->
|
|
Node1                                         Master
|<- topic: "farbsensor" message: { r: 0.1, g: 0.1, b: 0.1 } --- |

```

Die Nachrichten (**Messages**) die dabei übertragen werden können sind streng typisiert. Eine Message kann **andere Messages, oder Felder von Messages** enthalten. Die möglichen Datentypen sind hierbei die primitiven Typen **int, float und bool**.

### 2.3.3 Sychrone Kommunikation

Die synchrone Kommunikation wird durch Services gelöst. Ein Node kann beim Master einen Service registrieren. Ein zweiter Node kann daraufhin einen Request schicken und erhält von der entsprechenden Node eine Response. Dies ist besonders geeignet für RMI und einmalige Anfragen.

```

Node1                                         Master
|           register service
|----- service: "farbsensor" ----->
|
|
Node2                                         Master
|           call service
|----- getfarbe ----->
|
|
Node1                                         Master
|<----- getfarbe -----
|----- { r: 0.1, g: 0.1, b: 0.1 } ----->
|
|
Node2                                         Master
|           response
|----- { r: 0.1, g: 0.1, b: 0.1 } ----->

```

# 3 Lokalisation autonomer mobiler Robotersysteme

**Lokalisation** Ermitteln der aktuellen Position des Roboters.

**Pfadplanung oder Navigation** beantwortet die Fragen **Wie gelange ich dorthin?** Bewegungsplanung oder Pfadplanung bedeutet die Berechnung der Fahrroute und der daraus abgeleiteten Bahn vom aktuellen Punkt zum Zielpunkt.

**Unterscheidung zwischen unbekannter und bekannter Umgebung**

**Kartenerstellung, Mapping oder Umgebungsmodellierung** Die Auswertung der vom Roboter mittels Sensoren erfassten Daten der Umgebung mit dem Ziel, ein Umgebungsmodell zu erzeugen oder zu vervollständigen.

⇒ Großes Problem

⇒ Selbstlokalisierung und Kartenerstellung bedingen sich gegenseitig.

## 3.1 Varianten der Selbstlokalisierung

### Lokale Selbstlokalisierung (position tracking)

- Die Startposition des Roboters ist **ungefähr bekannt**.
- **Relative** Selbstlokalisierung
- Neuberechnung der Position mithilfe der Sensordaten bei Bewegung.
- Bezugspunkt ist der Startpunkt.
- **Methoden** sind Odometrie und Trägheitsnavigation

### Globale Selbstlokalisierung

- Die Startposition ist unbekannt.
- **absolute Positionierung**
- Lokalisation durch Sensordaten und erkennen von **signifikanten** Umgebungsmerkmalen
- **Methode** ist Triangulation

### Kidnapped Robot Problem

1. Die Position des Roboters ist anfangs bekannt
2. Der Roboter wird willkürlich mit temporär deaktivierten Sensoren an eine beliebige andere Position versetzt, ohne darüber informiert zu werden.

3. Auch dann muss das Verfahren robust die Position wiederfinden, zunächst muss der Roboter dies erkennen und sich dann relokalisieren
4. Es muss eine erneute globale Lokalisierung durchgeführt werden

## 3.2 Relative Lokalisierung versus Absolute Lokalisierung

### Relative Lokalisierung

Auch: lokale, inkrementelle Lokalisierung oder 'tracking'.

Relativ zu einer Startpose wird sukzessiv die Änderung der Pose an discreten, aufeinanderfolgenden Zeitpunkten ermittelt und integriert.

### Absolute Lokalisierung

Auch: globale Lokalisierung

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)

## 3.3 Transformation von Koordinatensystemen lokale $\leftrightarrow$ 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** Zweiradbetriebener Kreisförmiger Roboter und Bewegung in der Ebene.

**Lokales Koordinatensystem** An den Roboter verbunden mit Ursprung in der Mitte der Antriebsachse. Die x-Achse zeigt in Richtung des Roboterfront.

### Roboterposition im globalen Koordinatensystem

Globale Koordinaten  $M(x_M, y_M)$  und der Drehung von  $x'$  im Bezug zu  $x$  als Winkel  $\theta$ .

$$\Rightarrow \text{Pose } p \text{ mit } p = \begin{pmatrix} x_M \\ y_M \\ \theta \end{pmatrix}$$

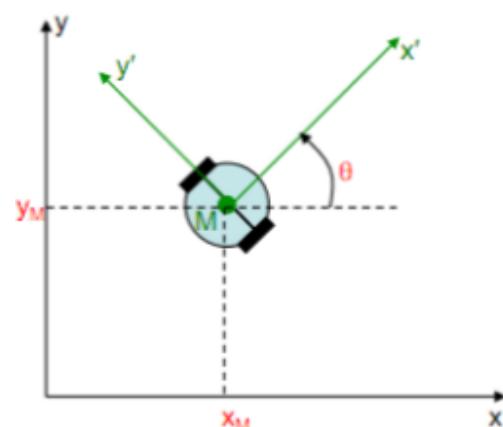


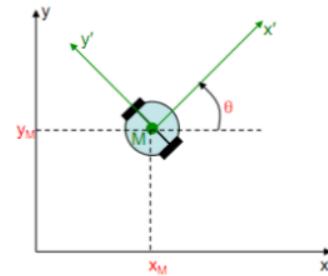
Figure 3.1

## Transformation von lokalen in globale Koordinaten

$$R_\alpha \mapsto \begin{pmatrix} \sin \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix}$$

$$\vec{m} = \begin{pmatrix} x_M \\ y_M \end{pmatrix}$$

$$\vec{t}_g = R_\theta \cdot \begin{pmatrix} x' \\ y' \end{pmatrix} - \vec{m}$$

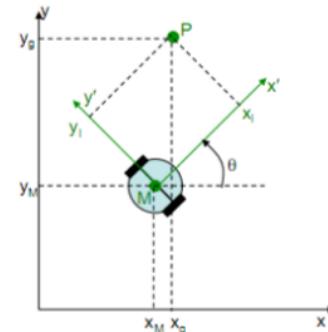


## Transformation von globalen in lokale Koordinaten

$$R_\alpha \mapsto \begin{pmatrix} \sin \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix}$$

$$\vec{m} = \begin{pmatrix} x_M \\ y_M \end{pmatrix}$$

$$\vec{t}_l = R_{-\theta} \cdot \left( \begin{pmatrix} x_g \\ y_g \end{pmatrix} - \vec{m} \right)$$



## 3.4 Karten für statistische und dynamische Umgebungen

Generell gilt: **Karten** sollen eine explizite Repräsentation des Raumes sein.

Diese sind auf die Sensorik des Roboters zugeschnitten und nicht vorrangig für den menschlichen Betrachter bestimmt.

### 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. Dazu gehört das verschwinden und auftauchen von bekannten oder unbekannten Objekten.

"Lernende" Karten sind ein fundamentales Problem in der mobilen Robotik

#### 3.4.1 Mapping Methoden

Durch Koordinaten-Transformation kann zwischen den verschiedenen Referenz-Frames beliebig gewechselt werden.

## Weltzentriert

Die Pose aller Objekte werden in der Umgebung in Bezug auf ein festes Koordinatensystem repräsentiert.

Dies kann Indoor eine Zimmerecke sein und Outdoor ein globales Koordinatensystem wie Längen- und Breitengrade, i.d.R. nutzen von **WGRS**(World Geographic Reference System)

## Roboterzentriert

Wird gebraucht um bspw. Kollisionen zu vermeiden. Diese nimmt als Bezugspunkt den Roboter.

### 3.4.2 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**

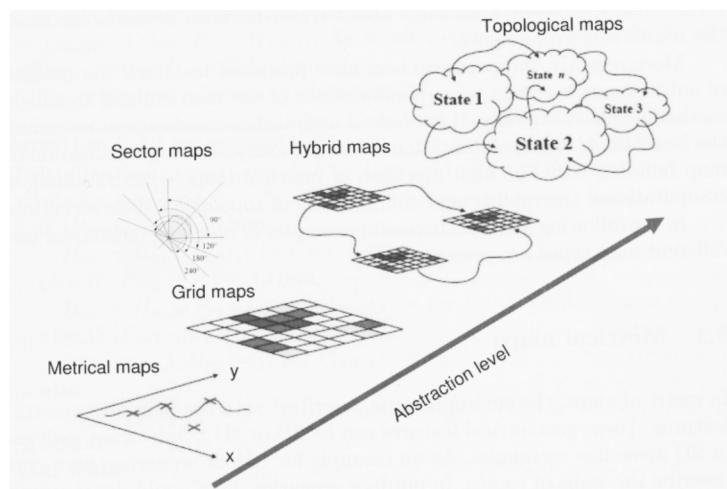


Figure 3.2: Die verschiedenen Umgebungsmodelle

## Arten von Umgebungsmodellen

### Kontinuierliche metrische Karten

2D oder 3D bei denen jedes Objekt eine Koordinate erhält.

### Diskrete metrische Karten, Grid Maps

2D oder 3D bei der Raum in gleichmäßige oder ungleichmäßige Teile aufgeteilt wird. Ein Objekt wird mit einem dieser Teile assoziiert.

### Hybrid Maps

z.B. Die Kontinuierliche metrische Anordnung von Grid Maps.

### Topologische Modelle

Nur 2D bei der die Beziehung der Objekte zueinander im Vordergrund steht.

### 3.4.3 Kontinuierliche Metrische Karten

Metrische Lokalisierung beruht auf Ultraschall oder Laserscannern, bei der eine exakte Beschreibung der Umgebung in 2D oder 3D möglich ist.

**Vorteil:** detailliertes Bild der Umgebung

**Nachteil:** große, unstrukturierte Datenmengen erschweren die Pfadplanung

### 3.4.4 Grid Maps - Rasterkarten

Die Umwelt wird in gleichmäßiges Raster oder Grid zerlegt. Wobei für jede Zelle mitgeführt wird ob sie belegt ist oder nicht. Dabei können je nach Modell unterschiedliche Werte verwendet werden.

- frei und belegt
- frei, belegt und Mischbelegung
- Belegungswahrscheinlichkeit

Notwendige Informationen sind z.B.:

- x, y als Koordinaten (Zeile, Spalte) einer Zelle
- Sensordaten des Roboters
- Belegungswert

Dabei sind die Zellen unabhängig voneinander. Je höher die Messgenauigkeit der Sensoren ist, desto kleiner können die Rasterelemente gewählt werden.

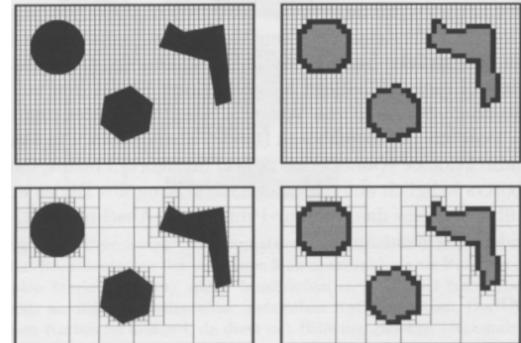


Figure 3.3: oben: gleichmäßiges Gitter,  
unten: adaptives Gitter

#### Gleichmäßige Gitterstruktur vs. Adaptiver Gitterstruktur

Für eine kompakte Notation können Grid Maps adaptive Unterteilt werden und im 2-dimensionalen Raum mit Quadtrees im 3-dimensionalen mit Octtrees gespeichert werden.

### 3.4.5 Adaptive Unterteilung

1. Ausgangszustand: Rechteck mit Hindernissen
2. Fläche wird unterteilt in 4 Rechtecke gleicher Größe
3. Jedes Rechteck wird rekursiv wieder in 4 Rechtecke unterteilt  $\Rightarrow$  Quadtree
4. Attributierung der Knoten:

**Frei:** Rechteck enthält keinen Teil eines Hindernisses

**Belegt:** Rechteck ist vollständig von Hindernis belegt

**Gemischt:** Rechteck enthält nur Teile eines Hindernisses

5. Nur gemischte Knoten werden weiter unterteilt

## Vorteile

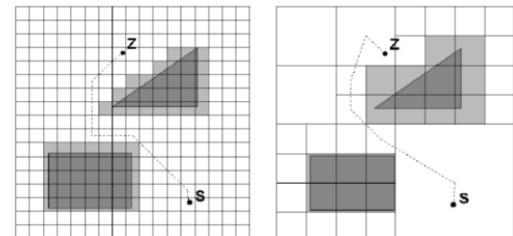
Schnell und leicht feststellbar, ob Punkt in einem Hindernis liegt

## 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.



## Weitere Beispiele für Umgebungskarten

- Laserscan Karten
- Bildbasierte Karten

## 3.4.6 Topologische Karten

Bedingt geeignet zur Lokalisation, Haupteinsatzgebiet ist die Pfadplanung.

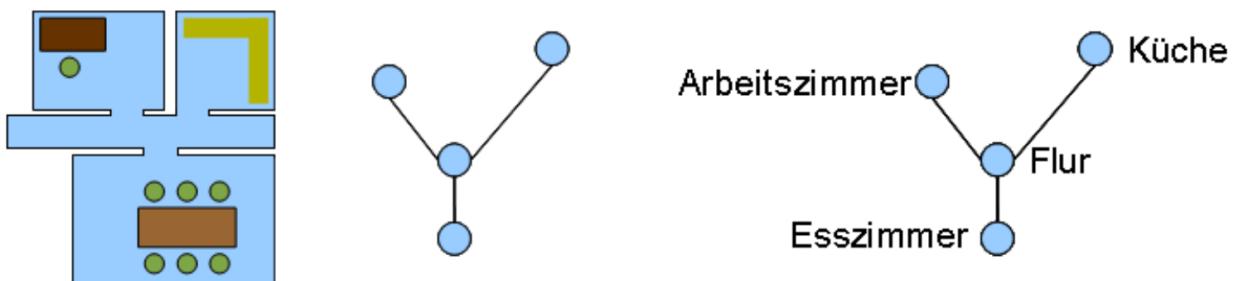


Figure 3.4

- 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.

### 3.4.7 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.

## 3.5 Passive und Aktive Selbstlokalisierung

### 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

### 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

## 3.6 Landmarken

### 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 errechneter Daten

### Natürliche Landmarken

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

### 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**

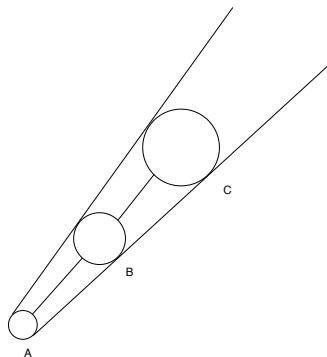


Figure 4.1

- 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 wegdifferenz berechnet mit:

$$\Delta = \Pi \times d \times n$$

- **Ausrichtung** kann durch differentiale Odometrie erfolgen: es werden bsw. die unterschiedlichen Entfernungsmessungen 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 Rades

#### 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

### 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

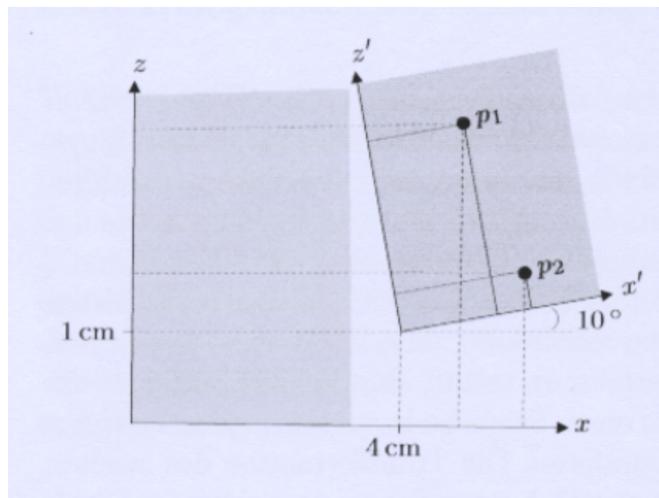


Figure 4.2

#### 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  $p_1$  und  $p_2$
- 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  $\theta$
- 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

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

## 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

$$\begin{aligned} t_x &= c_x - (c'_x \cos \theta - c'_z \sin \theta) \\ t_z &= c_z - (c'_x \sin \theta + c'_z \cos \theta) \end{aligned}$$

mit den Parametern:

$$\begin{aligned} c_x &= \frac{1}{N} \sum_i p_{x,i} & S_{xx'} &= \sum_i (p_{x,i} - c_x)(p'_{x,i} - c'_x) \\ c_z &= \frac{1}{N} \sum_i p_{z,i} & S_{xz'} &= \sum_i (p_{x,i} - c_x)(p'_{z,i} - c'_z) \\ c'_x &= \frac{1}{N} \sum_i p'_{x,i} & S_{zx'} &= \sum_i (p_{z,i} - c_z)(p'_{x,i} - c'_x) \\ c'_z &= \frac{1}{N} \sum_i p'_{z,i} & S_{zz'} &= \sum_i (p_{z,i} - c_z)(p'_{z,i} - c'_z) \end{aligned}$$

## 4.1.4 Weitere Lokalisierungsalgorithmen zur globalen Lokalisierung

- Wird beim 2D-Scanmatching nicht mit einem vorhergehenden Scan verglichen, sondern mit einer Karte, kann eine globale Lokalisierung realisiert werden
- Lokalisierung an Linien: Vergleich von Messung und Linienkarte
- Lokalisierung an visuellen SIFT (Scale-invariant feature transform) Merkmalen; ein Algorithmus zur Detektierung lokaler Merkmale, die größtenteils invariant gegenüber Rotation und Skalierung sind
- Verschiedene Algorithmen zur probabilistischen Lokalisierung in Karten

# 5 Navigation

## 5.1 Bekanntes vs. unbekanntes Terrain

Man unterscheidet zwischen Algorithmen für:

- **bekannte Umgebung:** auch während der Fahrt ändert sich die Umgebung nicht
- **unbekannte Umgebung:** entweder vollständig oder Teilweise unbekannte Umgebung

Ist das Gebiet **vollständig bekannt**, lässt sich die Suche mittels eines Graphen lösen. Ansonsten erfolgen die Berechnungen auf lokalen Teileinformationen von Sensoren mit denen **inkrementelle Anpassungen** vorgenommen werden.

## 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, 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

---

Listing 5.1: Regelung des Abstands  $d$

```
if (Distance to Wall > d) then
    turn to wall
fi

if (Distance to Wall < d) then
    turn away from wall
fi

if (Distance to Wall == d) then
    Drive straight ahead
fi
```

---

## 5.2.2 Bug1 Algorithmus

### Voraussetzungen

- Bewegung auf einer geraden Linie
- Konturverfolgung
- 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

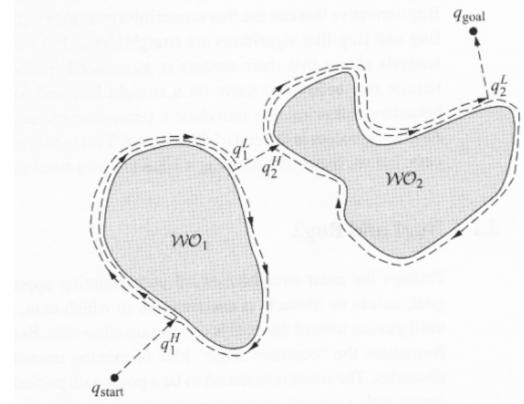


Figure 5.1: Bug1 Algorithmus Route

### Algorithmus

```

while ( noch nicht am Ziel ) {
    try {
        Folge gerader Linie zum Ziel
    } catch ( HindernisKontakt kontaktpunkt ) {
        Umfähre das Hindernis bis wieder bei $kontaktpunkt
        leavepoint = Punkt mit Lot vom Hindernis zum Ziel
        Fahre zu $leavepoint
    }
}

```

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

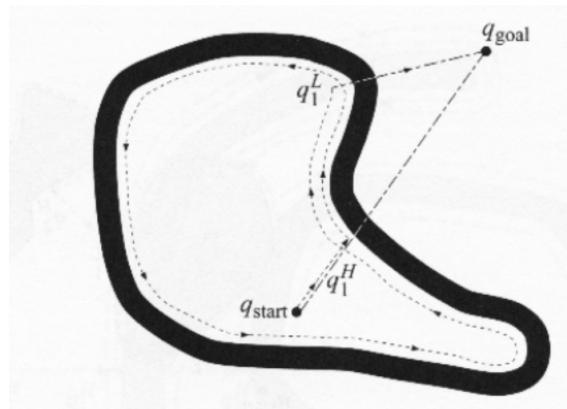


Figure 5.2

## 5.2.3 Bug2-Algorithmus

Schnellere Variante zu Bug1, da Hindernis nicht komplett umrundet wird. Eventuell keine mögliche Lösung wie bei Bug1. **Backtracking** und damit Veränderung der Umfahrrichtung eines Hindernisses können helfen.

Listing 5.2: Algorithmus ohne Backtracking

---

```
/**  
 * Start S  
 * Ziel T  
 * Derzeitige Position P = S  
 * Umfahrrichtung = rechts  
 */  
Konstruiere Strecke ST  
while (nicht bei T) {  
    try {  
        Fahre auf ST  
    } catch (HindernisKontaktPunkt H) {  
        while (P == Schnittpunkt mit ST && len(PT) < len(HT)) {  
            Umfahre das Objekt rechtsherum  
        }  
    }  
}  
}
```

---

## 5.2.4 Bug3-Algorithmus

Unterschied zu Bug2 ist, dass er nicht der Strecke ST strikt folgt sondern jede Möglichkeit nutzt näher ans Ziel T zu kommen. Auch hier Problematik keine Lösung zu finden und Verbesserung durch Backtracking möglich.

Listing 5.3: Algorithmus ohne Backtracking

---

```
/**  
 * Start S  
 * Ziel T  
 * Derzeitige Position P = S  
 * Umfahrrichtung = rechts  
 */  
while (nicht bei T) {  
    try {  
        Fahre zu T  
    } catch (HindernisKontakt) {  
        while (PT nicht frei) {  
            Umfahre das Objekt rechtsherum  
        }  
    }  
}
```

---

## 5.2.5 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 ⇒ Jeder Korridor:= Kante und jede Kreuzung:= Knoten
  - Bei bekanntem Labyrinth ⇒ Suchproblem 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** 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 ⇒ 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

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

**Global 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 geeignet Steuerbefehle, um den vorberechneten Pfad zu folgen

### 5.3.1 Konfigurationsraum

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

#### 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

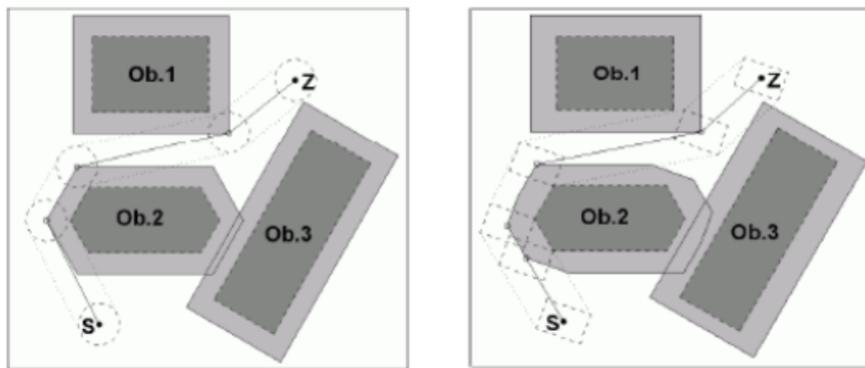


Figure 5.3

### Repräsentationen

- Graphen mit Knoten
- Reguläre Gitter
- Quad-Bäume oder Octal-Bäume
- Voronoidiagramme

## 5.4 Algorithmen und Methoden

Für die folgenden Algorithmen und Methoden wird ausgesagte, 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

**Straßenkartenmethoden oder Roadmaps** Eine Roadmap ist ein Graph mit Knoten und Kanten, bei dem alle befahrbaren Wege hinterlegt sind. Daraus kann ein kollisionsfreier Pfad vom Start- zum Zielpunkt erstellt werden.

Auf dieses Netzwerk können Standardmethoden der Graphentheorie, wie sie auch in der Autonavigation Verwendung finden, angewendet 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

---

```
/**  
 * PriorityQueue Open  
 */  
struct Knoten {  
    int cost  
    Knoten* predecessor  
}  
Knoten start = {0, null};  
push start on Open  
  
while (Open is not empty) {  
    Knoten n = pop lowest cost from Open  
    if (ist n Ziel?) {  
        konstruiere weg  
        return sucess  
    }  
    for (Knoten nx : nachfolger von n) {  
        newcost = n.cost + wegkosten von n nach nx  
        if (n ist in Open && nx.cost <= newcost) {  
            continue;  
        }  
        nx.predecessor = n  
        nx.cost = newcost  
        if (nx is not in Open) {  
            push nx in Open  
        }  
    }  
}  
return failure
```

---

### 5.4.2 A\*

---

```
/**  
 * PriorityQueue Open  
 * List Closed  
 */  
struct Knoten {  
    int total          // total cost  
    int est            // estimated cost  
    int known         // known cost  
    Knoten* predecessor  
}  
Knoten start;  
start.total = 0;  
start.est = GoalDistEstimate(start);  
start.known = start.total + start.est;  
start.predecessor = null;  
  
push start on Open
```

```

while (Open is not empty) {
    Knoten n = pop lowest total cost from Open
    if (ist n Ziel?) {
        konstruiere weg
        return sucess
    }
    for (Knoten nx : nachfolger von n) {
        newcost = n.known + wegkosten von n nach nx
        if ((n ist in Open || n ist in Closed) && nx.known <= newcost) {
            continue;
        }
        nx.predecessor = n
        nx.known = newcost
        nx.est = GoalDistEstimate(nx)
        nx.total = nx.known + nx.est
        if (nx is in Closed) {
            remove nx from Closed
        }
        if (nx is not in Open) {
            push nx in Open
        }
    }
    push n onto Closed
}
return failure

```

---

### 5.4.3 Wegsuche mit dem Sichtgraph-Algorithmus

**Visibility Map** Die Eckpunkte der Polygone (Hindernisse) werden zu Knoten, die eine Kante teilen wenn sie das Polygon nicht schneiden.

**Visibility Graph** Einfachste Visibility Map, bei der Start- und Zielknoten mitinbegriffen sind. **Hinderniskanten sind Teil des Sichtgraphen.**

#### Sichtgraphalgorithmus

1. Erzeuge Visibility Graph  $\Rightarrow$  Kanten = Wegstücke & Knoten = Orte
2. Kanten werden mit Entfernung gewichtet.
3. Suche: Suche die Geraden, die den kürzesten Weg zum Ziel verbinden.

Nachteile:

- Weg tangiert Hindernisse
- Graph enthält viele nutzlose Kanten

## Reduzierter Visibility Graph

Der reduzierte Sichtgraph besteht nur aus unterstützenden und trennenden Kanten.

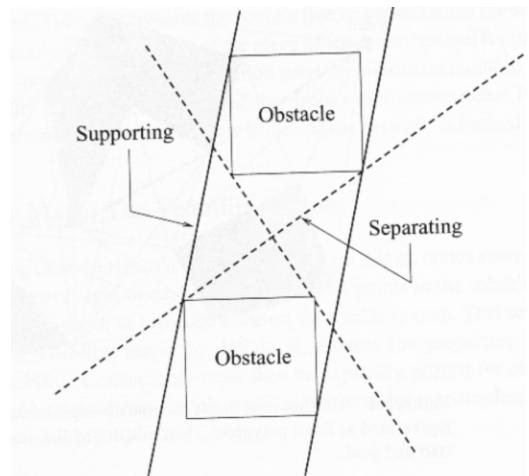
**unterstützende Kanten:** Tangente zu zwei Hindernissen, so dass die Hindernisse auf derselben Seite der Linie liegen

**trennende Kanten:** Tangente zu zwei Hindernissen so, dass die Hindernisse auf gegenüberliegenden Seiten der Tangente liegen.

Konstruktion: Alle Liniensegmente  $vv_i$  mit  $v \neq v_i$  dürfen keinen Schnittpunkt mit einer Hinderniskante haben.

⇒ **Nachteil:** Komplexität  $O(n^3)$

⇒ **Effizienter:** Plane Sweep Algorithmus  $O(n^2 \log n)$

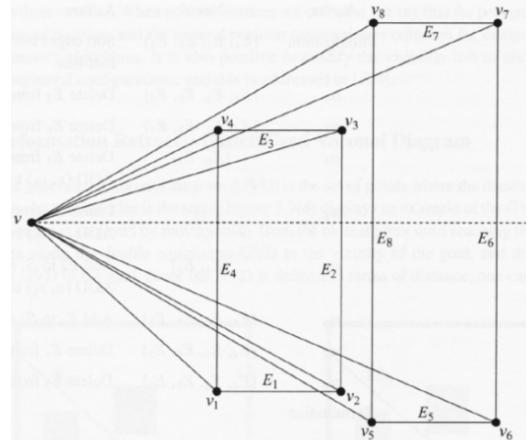


Listing 5.4: Sweep Line Algorithmus

---

```
/**  
 * Input: Set of vertices (whose edges do not  
 *         intersect) and a vertex v  
 * Output: A subset of vertices from the  
 *         input set that are within the line  
 *         of sight of v  
 */  
  
Angles = vertices.map(vi ->  
    angle between segment vvi horizontal line  
    from v  
)  
  
Angles.sort(ASC)  
  
List S // containing edges that intersect the  
// horizontal line from v  
  
for (angle : Angles) {  
    if (vi is visible to v) {  
        add edge (v, vi) to the visibility graph  
    }  
    if (vi ist the beginning of an edge, E, not in S) {  
        insert E into S  
    }  
    if (vi is the end of and edge in S) {  
        delete the edge form S  
    }  
}
```

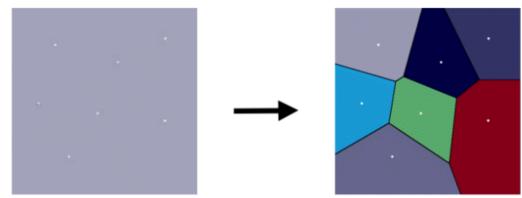
---



#### 5.4.4 Voronoi-Diagramme

Eine Fläche wird willkürlich mit Punkten besetzt. Diese Punkte bilden die Zentren der Polygonflächen. Jeder Punkt der Fläche gehört zu dem Zentrum das ihm am nächsten liegt  $\Rightarrow$  Gehört zur Polygonfläche dieses Zentrums. Wenn ein Punkt zu min. 2 Zentren den selben Abstand hat bildet er die Grenze zwischen den Polygonflächen dieser Zentren. Der Abstand wird mit der Euklidischen Abstandsfunktion bestimmt:

$$dist(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$



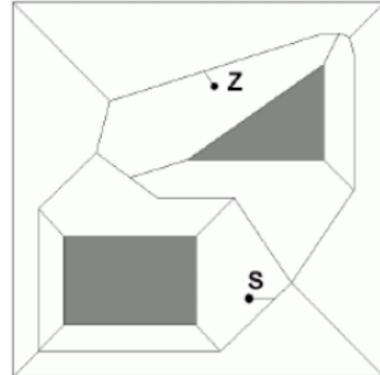
Die Menge der Grenzlinien nennt man Voronoi-Diagramm

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 dem sogenannten generalisierten Voronoi Diagramm (GVD) ist die **Form der Zentren** und die Art der **Abstandsfunktion** nicht festgelegt. Dadurch können Zentren komplexere Formen statt nur Punkte sein. Die damit entstehenden Voronoi-Kanten bilden mögliche kollisionsfreie Wegstücke.

In der Robotik werden Konfigurationsräume gerne damit repräsentiert und die Hindernisse werden zu Zentren des GVD gemacht. Bei der Wegfindung befinden sich Start- und Zielpunkt nicht auf dem Diagramm, diese werden mit der nächstgelegenen Kante verbunden.



#### 5.4.5 Navigation in einer Rasterkarte

In einer Rasterkarte gibt es nur die Bewegungsrichtungen (N, O, S, W). Sie besteht aus einem binären Raster. Es können zwei ähnliche Algorithmen zur Navigation verwendet werden:

- Fluten vom Startpunkt aus
- Fluten vom Zielpunkt aus (Wave Front Planer)

Bei beiden Algorithmen werden in jeder Iteration die Werte des Nachbarn des Randes der Welle, die kein Hindernis darstellen, auf den Wellenwert + 1 gesetzt.

Daraufhin kann der gewünschte Weg durch Absteigen (Backtracking) ermittelt werden. Die beiden Algorithmen unterschieden sich in der Startbelegung.

1	2	3	4	5	6	7	8
2	3		5	6	7	8	9
3	4		6	7	8		
4	5		7	8	9	10	11
5	6	7			10	11	12
			8	9		11	12
						13	14
11	10	9	10	11			
12	11	10	11	12	13	14	15

	Fluten vom Startpunkt	Wave Front Planer
Start der Welle	Startpunkt	Zielpunkt
Hindernis Wert	-1	1
Frei Startwert	0	0
Wellen Startwert	1	2
Anmerkung		Einfache Potentialfeldmethode

### Nachteile

- Weg führt ev. nah am Hindernis vorbei
- Der gesamte Raum muss durchsucht werden

**Vorteil** auch im 3D Raum anwendbar mit 6 Nachbarfeldern.

---

#### 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 5.4: Flooding Algorithmus

### 5.4.6 Potentialfeld Methode

Vorbild ist das elektrische Feld. Dabei müssen Start- und Zielposition und die Positionen aller Hindernisse bekannt sein. Der Zielpunkt und Freiraum erhalten ein anziehendes Potential. Der Startpunkt, Hindernisse und Wände ein abstoßendes Potential. Eine Karte mit den verschiedenen anziehenden und abstoßenden Kräften wird erstellt, diese nehmen linear mit dem Abstand zum Objekt ab.

Der Roboter bewegt sich nach der Methode des steilsten Abstiegs im Potentialfeld auf das Ziel zu.

**Problem** Der steile Abstieg ist ein greedy Algorithmus und kann dazu führen, dass ein lokales Minimum erreicht wird und somit in einer Sackgasse landet.  $\Rightarrow$  Potentialfunktion wählen, dass es nur ein Minimum im Ziel gibt.

# 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

Die Unsicherheiten werden mit Wahrscheinlichkeiten modelliert. So kann der Zustand von dynamischen Systemen probabilistisch geschätzt werden.

**Beispiel: Localization beim Roboter** Dabei wird die Position ermittelt an der sich der Roboter am wahrscheinlichsten befindet. Die Lokalisierung wird somit zu einem Wahrscheinlichkeitsdichte Problem bei dem alle vorangegangenen Sensorwerte berücksichtigt werden.

## 6.3 Umgebungsmodellierung mit Occupancy Grids

### 6.3.1 Satz von Bayes

$p(A)$  = Wahrscheinlichkeit das die Aussage  $A$  zutrifft

$P(A|B)$  = Wahrscheinlichkeit, dass  $p(A)$  unter der Voraussetzung das  $B$  gilt

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

### 6.3.2 Evidence Grids (Beweisraster)

**Problem:** reale Sensordaten erhalten häufig Rauschen. Rauschen bei Sensordaten führt zu Abweichungen des Idealwerts  $\Rightarrow$  schwerwiegende Fehler.

Im Gegensatz zu Occupancy Grids nutzen Evidence Grids keine Binärwerte um die Belegung darzustellen. Sie sammeln Beweismittel und geben die Wahrscheinlichkeit für die Belegung jeder Zelle an:

$$Z(x, y) = p(Z_{x,y,\text{belegt}}|\text{Sensorwert})/p(Z_{x,y,\text{frei}}|\text{Sensorwert})$$

Da die Wahrscheinlichkeit für die Belegung unter einem bestimmten Sensorwert schwer zu berechnen ist wird der Satz von Bayes benutzt:

$$Z(x, y) = \frac{p(\text{Sensorwert}|Z_{x,y,\text{belegt}}) * p(Z_{x,y,\text{belegt}})}{p(\text{Sensorwert}|Z_{x,y,\text{frei}}) * P(Z_{x,y,\text{frei}})}$$

Beim Start der Erstellung wird angenommen, dass die Wahrscheinlichkeit für die Belegung einer Zelle bei 50% liegt. Dannach wird die Wahrscheinlichkeit fortlaufend mit den aktuellen Sensorwerten aktualisiert.

Die direkte Anwendung des Bayessischen Filters auf das Selbstlokalisierungsproblem ergibt sich die sogenannte **Markov- Lokalisierung**

## 6.4 Bayes-Filter Algorithmus

Der Vertrauenszustand (**belief**) spiegelt internes Wissen des Roboters über den Zustand seiner Umgebung wieder.

### 6.4.1 Algorithmus

Ein Rekursiver Algorithmus der aus dem vorangegangenen Status den neuen berechnet:

$z_t$  = Observationsmesswert, aktuelle Sensorwerte der Umgebung

$u_t$  = Aktionsmesswert, aktuelle Zustandsänderungen im Intervall  $]t - 1, t]$

$x_t$  = Zustand zum Zeitpunkt  $t$

$$\text{bel}(x_t) = p(x_t|Z_{1:t}, u_{1:t})$$

Eine Vorhersage liefert, da dort die aktuellen Sensordaten noch nicht berücksichtigt werden:

$$\overline{\text{bel}}(x_t) = p(x_t|z_{1:t-1}, u_{1:t})$$

**Algorithmus 4.1:** Bayes-Filter (diskrete Verteilung). Eingabe ist eine Aktion  $u$  sowie ein Perzept  $z$ . Die Aktion führt zu der a-priori-Schätzung  $\overline{Bel}(x_{t+1})$ , mit Hilfe der Messung  $z$  wird daraus dann die a-posteriori-Schätzung  $Bel(x_{t+1})$  aktualisiert.

**Eingabe :** Aktueller Belief  $Bel(x_t)$ , sowie ein Paar  $(u, z)$ .  
**Ausgabe:** Aktualisierter Belief  $Bel(x_{t+1})$

---

```

1: for alle Zustände  $x_t$  do                                // Update nach Aktion
2:    $\overline{Bel}(x_{t+1}) = \sum_{x_t} P(x_{t+1} | u, x_t) Bel(x_t)$ 
3: end for
4:  $\eta = 0$ 
5: for alle Zustände  $x_{t+1}$  do                      // Update nach Messung
6:    $Bel(x_{t+1}) = P(z | x_{t+1}) \overline{Bel}(x_{t+1})$ 
7:    $\eta = \eta + Bel(x_{t+1})$ 
8: end for
9: for alle Zustände  $x_{t+1}$  do                      // Normierung
10:   $Bel(x_{t+1}) = \eta^{-1} Bel(x_{t+1})$ 
11: end for
12: return  $Bel(x_{t+1})$ 

```

---

### 6.4.2 Beispiel Aufgabenstellung

Roboter schätzt den Zustand einer Tür mittels einer Kamera, er macht einen Beobachtung z.  
Annahmen:

- Türe kann nur offen oder geschlossen sein
- Nur der Roboter kann den Zustand der Tür ändern

Der Roboter kennt den Zustand der Tür nicht  $\Rightarrow$  Gleichverteilung

$$\begin{aligned} bel(X_0 = \text{open}) &= 0.5 \\ bel(x_0 = \text{closed}) &= 0.5 \end{aligned}$$

Verrauschen der Sensoren  $\Rightarrow$  bedingt Wahrscheinlichkeit

$$\begin{aligned} p(Z_t = \text{sense\_open} | X_t \text{ is\_open}) &= 0.6 \\ p(Z_t = \text{sense\_closed} | X_t \text{ is\_open}) &= 0.4 \\ p(Z_t = \text{sense\_open} | X_t \text{ is\_closed}) &= 0.2 \\ p(Z_t = \text{sense\_closed} | X_t \text{ is\_closed}) &= 0.8 \end{aligned}$$

Der Roboter kann die Tür öffnen

$$\begin{aligned} p(X_t = \text{is\_open} | U_t = \text{push}, X_{t-1} = \text{is\_open}) &= 1 \\ p(X_t = \text{is\_closed} | U_t = \text{push}, X_{t-1} = \text{is\_open}) &= 0 \\ p(X_t = \text{is\_open} | U_t = \text{push}, X_{t-1} = \text{is\_closed}) &= 0.8 \\ p(X_t = \text{is\_closed} | U_t = \text{push}, X_{t-1} = \text{is\_closed}) &= 0.2 \end{aligned}$$

Der Roboter tut nichts

$$\begin{aligned} p(X_t = \text{is\_open} | U_t = \text{nothing}, X_{t-1} = \text{is\_open}) &= 1 \\ p(X_t = \text{is\_closed} | U_t = \text{nothing}, X_{t-1} = \text{is\_open}) &= 0 \\ p(X_t = \text{is\_open} | U_t = \text{nothing}, X_{t-1} = \text{is\_closed}) &= 0 \\ p(X_t = \text{is\_closed} | U_t = \text{nothing}, X_{t-1} = \text{is\_closed}) &= 1 \end{aligned}$$

### 6.4.3 Beispiel Rechnung

Für  $t = 1$  mit  $u_1 = \text{nothing}$  und  $z_1 = \text{sense\_open}$ :

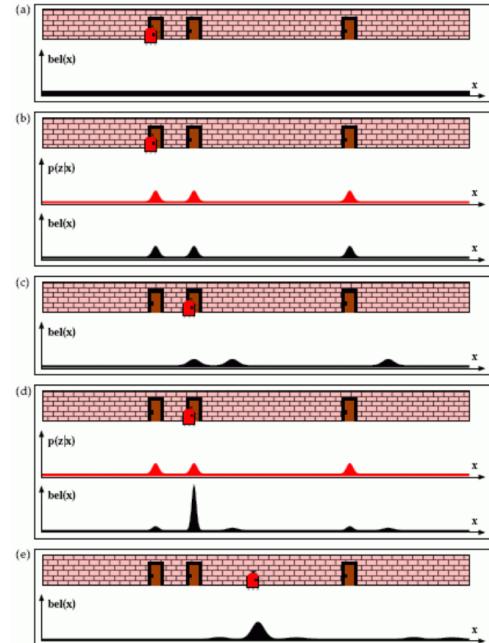
$$\begin{aligned}\overline{\text{bel}}(x_1 = \text{open}) &= \\ p(x_1 = \text{open}|u = \text{nothing}, x_0 = \text{open})\text{bel}(x_0 = \text{open}) + \\ p(x_1 = \text{open}|u = \text{nothing}, x_0 = \text{closed})\text{bel}(x_0 = \text{closed}) \\ &= 1 * 0.5 + 0 * 0.5 = 0.5 \\ \overline{\text{bel}}(x_1 = \text{closed}) &= \\ p(x_1 = \text{closed}|u = \text{nothing}, x_0 = \text{open})\text{bel}(x_0 = \text{open}) + \\ p(x_1 = \text{closed}|u = \text{nothing}, x_0 = \text{closed})\text{bel}(x_0 = \text{closed}) \\ &= 0 * 0.5 + 1 * 0.5 = 0.5 \\ \eta = 0 : \text{bel}(x_1 = \text{open}) &= \\ p(z = \text{sense}_{\text{open}}|x_1 = \text{open})\overline{\text{bel}}(x_1 = \text{open}) \\ &= 0.6 * 0.5 = 0.3 \\ \eta = 0.3 : \text{bel}(x_1 = \text{closed}) &= \\ p(z = \text{sense}_{\text{open}}|x_1 = \text{closed})\overline{\text{bel}}(x_1 = \text{closed}) \\ &= 0.2 * 0.5 = 0.1 \\ \eta = 0.4 : \text{bel}(x_1 = \text{open}) &= \text{bel}(x_1 = \text{open}) * \eta^{-1} = 0.3 * 0.4^{-1} = 0.75 \\ \text{bel}(x_1 = \text{closed}) &= \text{bel}(x_1 = \text{closed}) * \eta^{-1} = 0.1 * 0.4^{-1} = 0.25\end{aligned}$$

## 6.5 Markov Lokalisierung

Bei der Markov Lokalisierung geht von der Annahme aus, dass der nachfolgende Zustand nur vom aktuellen Zustand abhängt. Dafür bietet sich der Bayes-Filter an. Dabei wird der Bayes-Filter auf das Lokalisierungsproblem angewendet. Dafür ist als weitere Input die Karte notwendig. Für die Markov Lokalisierung ist eine **möglichst genaues Umgebungsmodell** notwendet.

Der Algorithmus ist wie beim Bayes-Filter es wird nur die Karte  $m$  als zusätzliches Parameter für die Wahrscheinlichkeit hinzugefügt.

$$\begin{aligned}\overline{\text{bel}}(x_t) &= \int p(x_t|u_t, x_{t-1}, m)\text{bel}(x_{t-1})dx \\ \text{bel}(x_t) &= \eta p(z_t|x_t, m)\overline{\text{bel}}(x_t)\end{aligned}$$



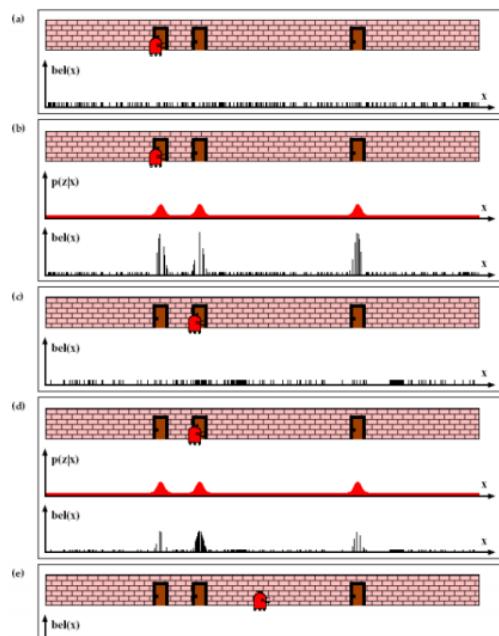
## 6.6 Monte Carlo Lokalisierung

Die Monte Carlo Lokalisierung ist ein Stichprobenbasiertes Approximationsverfahren für die Markov Lokalisierung. Der Bayes Filter wird auch hier benutzt, unterscheidet sich jedoch von gridbasierten Lokalisationsverfahren anhand der Betrachtung und Verarbeitung.

**Praxis:** Viele der Gitterzellen besitzen Wahrscheinlichkeit von 0. Diese Gitterzellen miteinzubeziehen ist ineffizient. Diese können vernachlässigt werden.

### 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 mit gewichteten Zufall Partikel entzogen  $\Rightarrow$  Integrierung des Steuerbefehls ( $u_k$ )
- 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

#### 6.6.1 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  $\chi$ ;

```

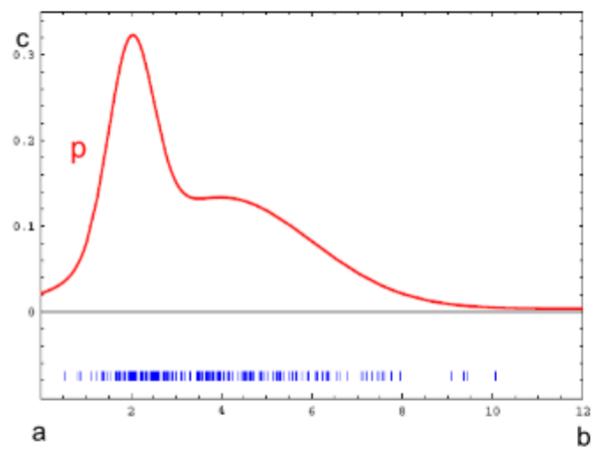


Figure 6.1: Partikelmengen

## 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  $\mu$  sowie die Kovarianz  $\sum_t$  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

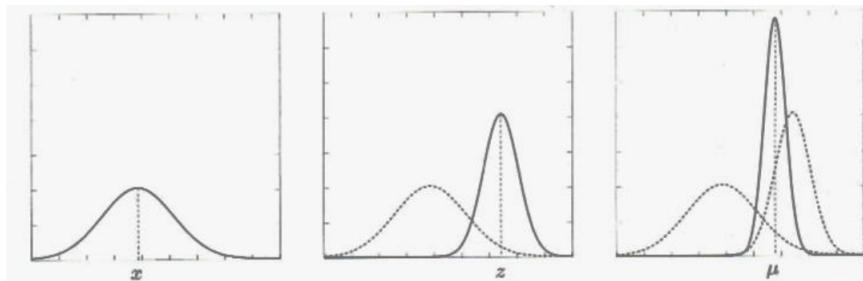


Figure 6.2: Kalman Filter

- **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 (SLAM)

### 6.8.1 Landmarkenbasiertes SLAM Problem

- **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

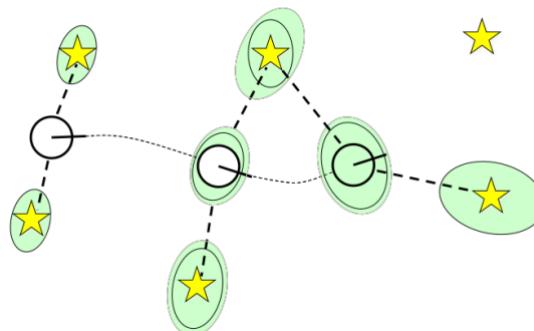


Figure 6.3

### 6.8.3 Funktionsweise

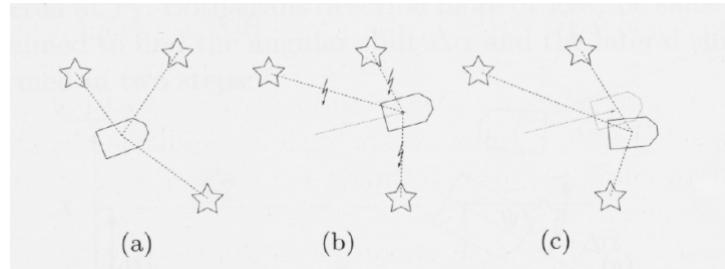


Figure 6.4: SLAM Darstellung

- (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

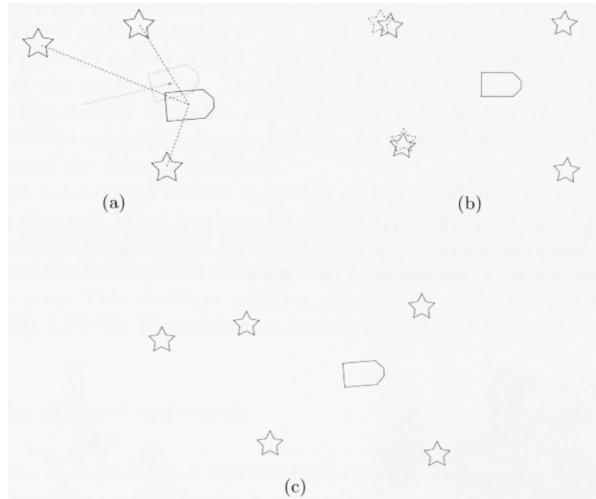


Figure 6.5

- (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

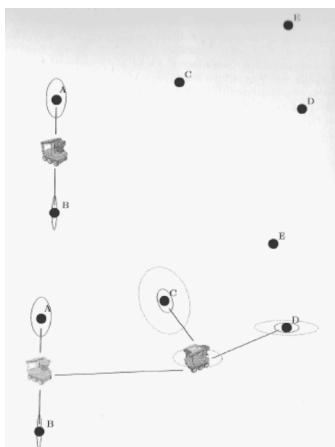


Figure 6.6

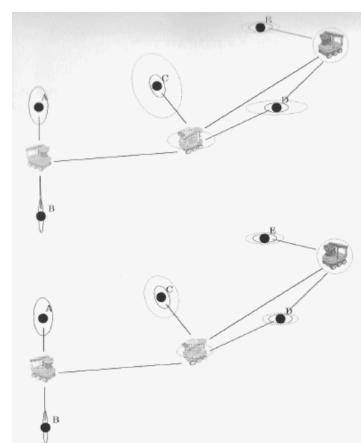


Figure 6.7

### 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

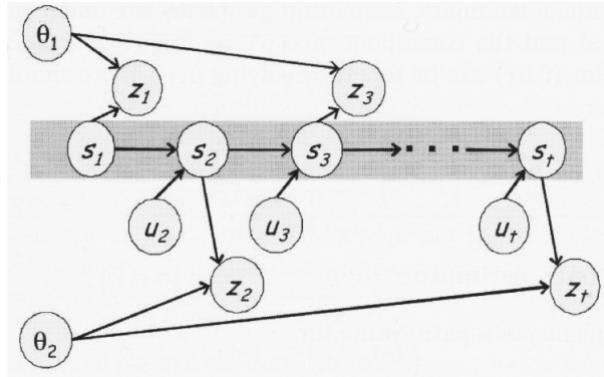


Figure 6.8: Bayes Netzwerk

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  $\theta_k$
  - 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  $\theta_1$  mittels  $z_1$  zum Zeitpunkt  $t = 2$  beobachtet er  $\theta_2$  via  $z_2$  und zum Zeitpunkt  $t = 3$  wieder  $\theta_1$
- Die Beobachtung  $z_t$  ist abhängig von der globalen Position der Landmarke  $\theta_k$  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

Bei der Schwarmrobotik werden simple Roboter eingesetzt. Diese haben einzeln nur ein sehr begrenzte Handlungs- und Wahrnehmungsmöglichkeiten. In der Gemeinschaft agieren sie als intelligentes System.

**Vorteil:** robust, skalierbar und flexibel

- robust, skalierbar und flexibel
- **Kosten:** die einzelnen Roboter sind jeweils billig und können leichter ersetzt werden, als ein einzelner Großer.
- **Verlässlichkeit:** Ausfall eines einzelnen hat keine großen Auswirkungen  $\Rightarrow$  Neustrukturierung und Koordination
- **Flexibilität:** Durch unterschiedliche Kooperation können die Roboter unterschiedlichste Aufgaben ausführen.

**Nachteil:** Schwarmverhalten nur schwer vorhersagbar

## 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 Nachbarn

**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.

	Swarm Robotics	Multi-Robot Systems	Sensor Networks	Multi-Agent Systems
Population Size	Varies in great range	Small	Fixed	In a small range
Control	Decentralized and Autonomous	Centralized or Remote	Centralized or Remote	Centralized or Hierarchical or Network
Homogeneity	Homogeneous	Usually Heterogeneous	Homogeneous	Homogeneous or Heterogeneous
Flexibility	High	Low	Low	Medium
Scalability	High	Low	Medium	Medium
Environment	Unknown	Known or Unknown	Known	Known
Motion	Yes	Yes	No	Rare
Typical Applications	Post-disaster relief Military applications Dangerous applications	Transportation Sensing Robot soccer	Surveillance Medical care Environmental protection	Net resources management Distributed control

Figure 7.1: Vergleich der Unterschiedlichen Systeme

## 7.2 Mechanismen zur Schwarmorganisation

**Positive Rückkopplung:** Ein Agent motiviert ein bestimmtes Verhalten eines anderen

**Negative Rückkopplung:** Ein Agent hemmt ein bestimmtes Verhalten eines anderen

**Zufällige Abweichung:** Ein Agent weicht zufällig vom normalen Verhalten ab um neue Lösung zu finden.

**Vielfachwechselwirkung:** Agenten können gleichzeitig mit mehreren Agenten eine Wechselwirkung aufbauen

**Stigmmerie:** Kommunikation in einem dezentral organisierten System durch Modifikation der Umgebung ⇒ Ein Agent verändert die Umwelt und löst damit ein bestimmtes Verhalten bei anderen Agenten aus

## 7.3 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**

## 7.4 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 **beworzt**, 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

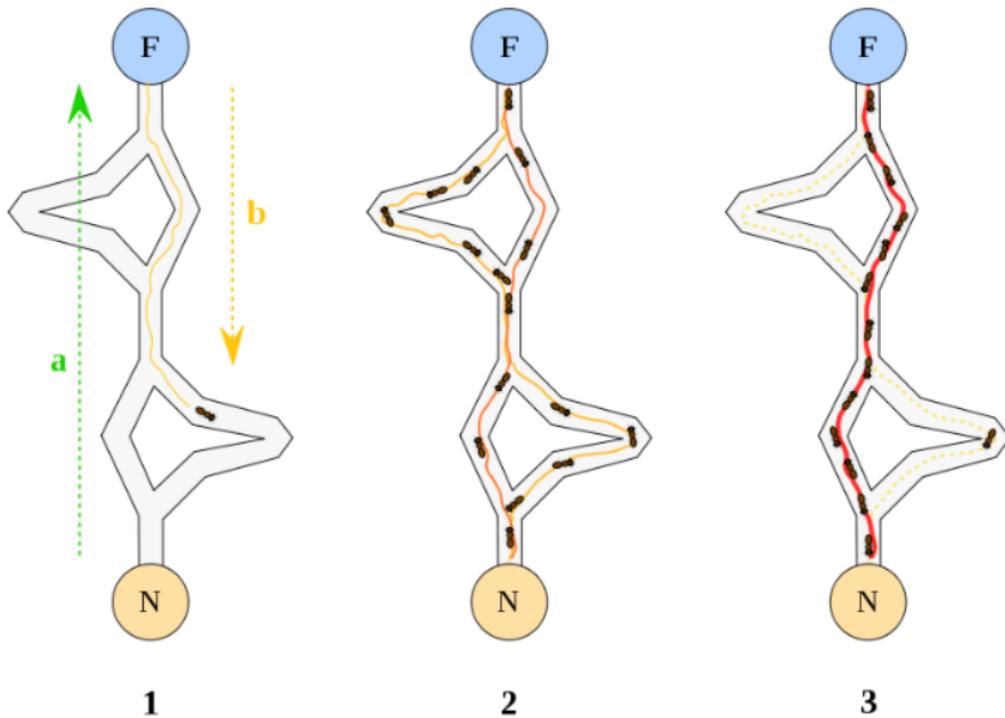


Figure 7.2: Darstellung des Ameisenalgorithmus Prinzips

## 7.4.2 Ant Colony Optimization Algorithm (ACO)

### 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 vorhandener Pheromone
4. Operationen, die globales Wissen voraussetzen 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 7.3: Ameisenalgorithmus - 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:

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

- $\alpha$  und  $\beta$  steuern das Verhältnis zwischen Anteil der Pheromone und lokaler Information
- für  $\alpha$  hat man den klassischen Greedy Ansatz

# 8 Locomotion

Die Fortbewegungsarten sich oft durch die Natur inspiriert. Dabei haben Roboter mit Beinen in der Regel mehr Freiheitsgrade und sind mechanisch komplexer als Roboter auf Rädern.

## 8.1 Laufroboter

### Vorteile von Laufrobotern

- Können auf unregelmäßigen Untergrund laufen
- Natürliche Fortbewegung
- Keine Umweltveränderung erforderlich
- Kompensation von Unebenheiten durch Anpassung der Schrittlänge und Schritthöhe

### 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, ...

### 8.1.1 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

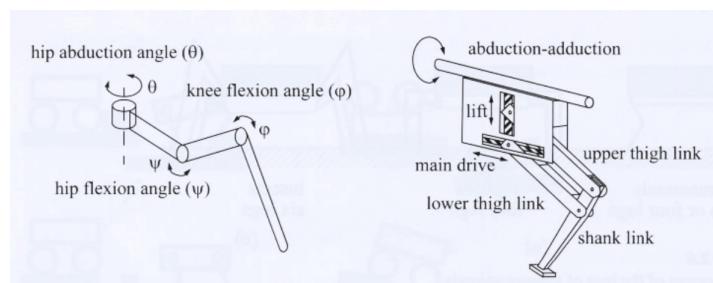


Figure 8.1: Freiheitsgrade für Roboterbeine

## Freiheitsgrade für Zweibeiner

- Zweibeiner haben meist 6 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

### 8.1.2 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.

### 8.1.3 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 Stemphase und in der Schwingphase
- Verbindet man die drei Beine in der Stemphase ergibt sich ein Dreieck

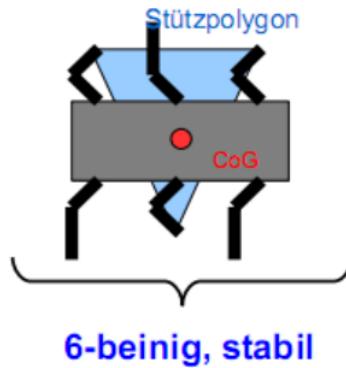


Figure 8.2

### 8.1.4 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

### 8.1.5 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.

# 8.2 Radroboter

## 8.2.1 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**

# 8.3 Kinematik mobiler Radroboter

## 8.3.1 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

### 8.3.2 Kinematik und Positionsveränderung für Zweiradantrieb

- **Voraussetzung:** kreisförmiger Roboter mit Zweiradantrieb, Bewegung ist nur in der Ebene möglich
- Position: Koordinaten  $x, y$ , Orientierung  $\alpha$
- Roboter misst zu diskreten Zeitpunkten  $t$  den mit dem linken bzw. mit dem rechten Rad zurückgelegten Weg
- $\Delta L$  und  $\Delta R$  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  $\Delta L = \Delta R$   

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

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

### 8.3.3 Fortbewegung bei Omnidirektionalem Antrieb

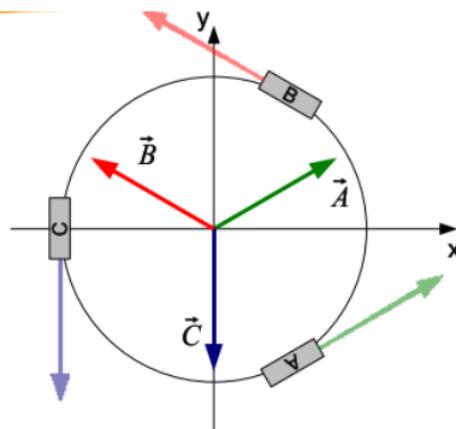


Figure 8.3: Einheitsvektoren

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

# 9 Lego Mindstorms Code

## 9.1 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();  
    }  
}
```

---

## 9.2 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(); }
}

```

---

## 9.3 Wishes

```

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

```

---

## 9.4 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);
        }
    }
}

```

---

## 9.5 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;
    }
}

}

```

---

## 9.6 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();
    }
}

}
```

---