Technology
Arts Sciences
TH Köln

Monte-Carlo-Lokalisation

AI-Projekt Sommersemster 2016

Arno v. Borries,
Jan Phillip Kretzschmar,
Andreas Walscheid

Betreuer:
Prof. Dr. Heiner Klocke,
Dipl. Inform. Alex Maier

26. September 2016

Inhaltsverzeichnis

1 Aufgabenstellung

Ziel dieses AI-Projektes war die Umsetzung der sogenannten Monte-Carlo-Lokalisation für die Lego NXT-Roboter der TH Köln. Dabei sollten die folgenden Anforderungen erfüllt werden:

1. Konstruktion und Bauanleitung eines zur Lokalisierung sinnvollen Robotermodells.

2. Implementierung des LeJOS Steuerprogramms zum Wahrnehmen der Umgebung und Ausführen der Aktionen.

3. Einarbeitung in AIMA3e-Java Framework und Erweiterung um MCL.

4. Implementierung der Anwendung unter der Verwendung der verfügbaren Interfaces aus AIMA3e-Java/aima-core.

5. Das Agentenprogramm soll auf einem Rechner laufen und über Bluetooth mit dem Roboter kommunizieren.

6. Das Modell der Umgebung soll in geeigneter Form geladen und visualisiert werden können.

7. Evaluation der entwickelten Anwendung in mindestens zwei konstruierten Umgebungen.

Dieses Dokument soll als Einführung in verschiedene Aspekte der Umsetzung der Aufgabenstellung dienen. Dem zur Seite gestellt sind die Java-Implementierung, der Roboter samt Bauanleitung und ein Benutzerhandbuch für die graphische Benutzeroberfläche inklusive Beschreibung einer geeigneten Vorgehensweise für Experimente in den Testumgebungen der TH Köln.

Zunächst wird im zweiten Abschnitt ein Blick auf den umzusetzenden Monte-Carlo-Lokalisationsalgorithmus geworfen, gefolgt von einer Beschreibung der Klassenstruktur und -Interdependenzen der Implementierung innerhalb der Vorgaben des AIMA-Frameworks im dritten Abschnitt. Der vierte Abschnitt beschäftigt sich mit der Implementierung der graphischen Benutzeroberfläche. Schließlich wird im fünften Abschnitt auf Aspekte der Benutzung des LeJos-Frameworks und des Roboterdesigns eingegangen bevor im sechsten Abschnitt ein kurzer Blick über den Tellerrand dieses Projektes hinaus gewagt werden soll, um mögliche Richtungen für die weitere Entwicklung aufzuzeigen.

2 Der Monte-Carlo-Lokalisationsalgorithmus

2.1 Probabilistische Robotik

Eine der zentralen Entwicklungen auf dem Gebiet der Robotik seit den 1990er Jahren ist die vermehrte Verwendung wahrscheinlichkeitsbasierter (auch *probabilistischer*) Algorithmen. Die wahrscheinlichkeitsbasierte Robotik vermeidet sowohl die unrealistische Annahme perfekter Modelle wie auch die ebenso überoptimistische Annahme perfekter Wahrnehmung in den beiden in den 70er und 80er Jahren des vorigen Jahrhunderts weit verbreiteten modell- und wahrnehmungsbasierten Robotik-Paradigmen.[1] Der Kernpunkt der probabilistischen Robotik ist die Darstellung von Information durch Wahrscheinlichkeitsdichten. Die Klasse der probabilistischen Techniken in der Robotik wird unterteilt nach den beiden Kernbereichen der Entwicklung autonomer robotischer Agenten:[2]

Probabilistische Wahrnehmung Robotische Agenten können sich des tatsächlichen Zustands ihrer Umgebung nie völlig sicher sein. Die Unsicherheit erwächst aus den konstruktionsbedingten Einschränkungen der Sensoren, Rauschen und vor allem aus der Tatsache, dass viele Umgebungen, in denen solche Agenten produktiv eingesetzt werden, immer einen bestimmten inherenten Grad an Unvorhersehbarkeit enthalten. Nach dem wahrscheinichkeitsbasierten Robotik-Paradigma errechnet der Agent eine Wahrscheinlichkeitsverteilung über die möglichen Weltzustände, anstatt sich mit einer - wie auch immer errechneten - einzelnen 'besten Schätzung' zufrieden zu geben. Der Roboter kann - auf diese Weise in Kenntnis seiner eigenen Unkenntnis gesetzt - Messdaten mit einer deutlich erhöhten Fehlertoleranz erfolgreich verarbeiten. Er erfüllt dadurch eine der wesentlichen Forderungen an einen wahrhaft autonomen Agenten.

Probabilistische Steuerung Autonome Roboter müssen trotz ihrer eben beschriebenen Unsicherheit über den Zustand ihrer Umgebung zielgerichtet agieren können. Folglich bezieht der wahrnehmungsbasierte Ansatz diese Unsicherheit bei der Entscheidungsfindung des Agenten mit ein. Anstatt nur die wahrscheinlichsten Ausgangs- und Endzustände zu berücksichtigen, wird bei der Verwendung probabilistischer Steuerungsalgorithmen häufig das entscheidungstheoretische Optimum angestrebt, in dem Entscheidungen auf allen denkbaren Fällen beruhen.

2.2 Monte-Carlo-Algorithmen auf Partikelfiltern

Ein wichtige Algorithmenfamilie im Bereich der probabilistischen Wahrnehmung sind Monte-Carlo-Algorithmen auf Partikelfiltern. Die Markow-Ketten-Monte-Carlo-Methode (MCMC) wurde erstmals in den 40er Jahren von Stanislaw Ulam und Nicholas Metropolis beschrieben. Mit ihrer Hilfe können analytisch nicht oder nur

[1] [Thrun, S.511]
[2] [Adiprawita et al., S.166]

aufwendig lösbare Probleme mit Hilfe der Wahrscheinlichkeitstheorie numerisch gelöst werden.[3] Die grundlegende Vorgehensweise der Monte-Carlo-Methode ist wie folgt:

1. Wählen eines geeigneten Definitionsbereichs für die Eingabemenge.
2. Zufälliges Erzeugen einer Anzahl von Eingaben entsprechend einer Wahrscheinlichkeitsverteilung über dem in 1. gewählten Definitionsbereich.
3. Berechnung der Ausgabewerte der einzelnen Eingaben.
4. Aggregation der Ergebnisse (z.B. durch Durchschnittsbildung).

Eine *Partikelwolke* ist eine Multimenge von Vektoren, den sogenannten *Partikeln*, die jeweils einen Datenpunkt darstellen. Als *Partikelfilter* wird die Anwendung der Monte-Carlo-Methode auf eine Partikelwolke bezeichnet. Hierfür wird dem Partikelvektor jeweils ein weiterer Skalarwert angehängt, der eine Gewichtung des Partikels darstellt. Im Rahmen der hier betrachteten Algorithmen lässt sich diese Gewichtung in die Wahrscheinlichkeit übersetzen, mit der der durch den Datenvektor abgebildete Wahrnehmungsausschnitt mit der tatsächlichen Umgebung korrespondiert.

Partikelfilter werden verwendet, um annäherungsweise Inferenzen in teilweise beobachtbaren Markow-Ketten zu ziehen.[4] Eine nähere Diskussion von Markow-Ketten und ihrer Verwendung für Agenten in unsicheren Umgebungen würde den Rahmen dieser Arbeit deutlich sprengen. Eine gute Einführung in die Thematik findet sich z.B. bei Russel / Norwig (2004).[5] Es soll an dieser Stelle der Hinweis genügen, dass der oben mit 3. nummerierte Schritt der Monte-Carlo-Methode mit Hilfe einer eigenen Markow-Kette für jeden Partikel berechenbar ist. Für die probabilistische Wahrnehmung ist natürlich Vorraussetzung, dass die Umgebung des Roboters die Markow-Eigenschaft erfüllt, dass sich der aktuelle Umgebungszustand aus einer endlichen Anzahl Beobachtungen der Umgebung rekonstruieren lässt.

2.3 Monte-Carlo-Lokalisation

Lokalisation ist der vermutlich bekannteste Anwendungsfall für Partikelfilter im Bereich der Robotik. Diese Bekanntheit rührt daher, dass die Lokalisierung von Objekten in der Umgebung eine der Kernaufgaben der Roboterwahrnehmung ist, befähigt sie doch den Agenten erst zur weiteren Interaktion mit diesen Objekten. Um Missverständnissen vorzubeugen, sei an dieser Stelle betont, dass die Lokalisation von Objekten auch die Lokalisierung des Roboters selbst mit einschließt. Die *Monte-Carlo-Lokalisation* (MCL) löst gleich drei miteinander verwandte Probleme aus dem Bereich der Robotik, die bis zur Entwicklung performanter MCL-Algorithmen Ende der 1990er Jahre in tatsächlichen Umgebungen nicht gemeistert werden konnten:[6]

[3] [Ulam / Metropolis]
[4] [Thrun, S.511]
[5] [Russel / Norvig (2004), S.659ff]
[6] [Thrun, S.513]

Verfolgungsproblem Die Aufgabe besteht darin, die aktuelle Position eines Objekts zu bestimmen, dessen Ausgangsposition bereits bekannt ist. Wichtig ist hierbei die Annahme, dass die Position des Objekts nur vergleichsweise langsam veränderlich ist.

Globales Lokalisationsproblem Hier ist die anfängliche Position des zu lokalisierenden Objekts völlig unbekannt.

Kidnapping-Problem Der Agent muss ein Objekt lokalisieren, dessen bisherige Position ihm bekannt ist, das aber ohne sein Wissen auf eine völlig neue Position versetzt wurde.

Der MCL-Algorithmus ist theoretisch in Umgebungen mit beliebig vielen Dimensionen anwendbar. Er liefert eine Schätzung der *Pose* eines Roboters in einer bekannten Umgebung. In einer zweidimensionalen Umgebung wird eine solche Pose z.B. ein Vektor sein, der die Positionskoordinaten des Roboters sowie seine Orientierung enthält. Je nach Roboterkonstruktion und Umgebung können in der Pose aber auch noch andere Informationen enthalten sein. Der Algorithmus läuft wie folgt ab:[7]

1. Generiere eine Wolke aus zufälligen Posen-Partikeln $x_0^{(i)}, i = 1, ..., N$ und setze $t = 1$.

2. Propagiere die Agentenbewegung u_t auf alle Partikel $x_0^{(i)} N$ entsprechend dem Bewegungsmodell und addiere jeweils ein zufälliges Rauschen.

$$\tilde{x}_t^{(i)} = Bewegungsmodell(x_{t-1}^{(i)}, u_t)$$

3. Führe eine Entfernungsmessung ρ der Roboterumgebung mit Hilfe entsprechender Sensoren durch.

4. Führe 'virtuelle' Entfernungsmessungen $R^{(i)}$ für alle Partikel $\tilde{x}_0^{(i)}$ an der Repräsentation der Umgebung durch.

$$R_t^{(i)} = Sensormodell(\tilde{x}_t^{(i)})$$

5. Berechne die Gewichtungen $w_t^{(i)}$ für alle Partikel $\tilde{x}_0^{(i)}$. Dabei werden Partikel höher gewichtet, deren virtuelle Sensordaten $R^{(i)}$ eine geringere Diskrepanz zu der tatsächlichen Evidenz ρ aufweisen.

$$w_t^{(i)} = \frac{\beta}{\sum_{j=1}^{N} (\rho_j - R_j^{(i)})^2}$$

   Wobei j der Index des Messungsvektors ist und β eine Normalisierungskonstante, die hauptsächlich zur Umkehrung der Proportionalität dient und deren tatsächlicher Wert uninteressant ist, da im nächsten Schritt normalisiert wird.

[7] Vgl. [Sarkar et al., S.53]

6. Normalisiere die Gewichtungen.

$$w_t^{(i)} = \frac{\tilde{w}_t^{(i)}}{\sum_{k=1}^{N} \tilde{w}_t^{(k)}}$$

7. Berechne die geschätzte Pose.

$$\mu_t = \sum_{i=1}^{N} w_t^{(i)} \tilde{x}_t^{(i)}$$

8. Wähle und ersetze N Partikel $\left\{\tilde{x}_t^{(i)}\right\}_{i=1}^{N}$ aus der Multimenge $\left\{x_t^{(i)}\right\}_{i=1}^{N}$ auf der Grundlage ihrer Gewichtungen.

9. Setzte $t = t + 1$ und gehe zu Schritt 2.

Der Algorithmus wird terminiert, wenn die in Schritt 7 erzeugte Pose mit hinreichender Genauigkeit berechnet werden konnte, d.h. wenn der größte Abstand zwischen den Partikeln in der Wolke klein genug für den beabsichtigten Zweck der Lokalisation geworden ist. Die folgende Serie von Abbildungen soll das Prinzip verdeutlichen, sie zeigt die Lokalisierung eines Roboters in einer Straßenumgebung vom Anfangszustand der Partikelwolke bis zum gut lokalisierten Endzustand.

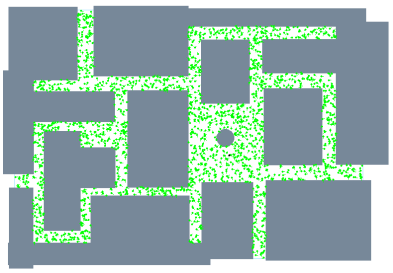

Abbildung 1: Anfangszustand der Lokalisation mit zufälligen Partikeln

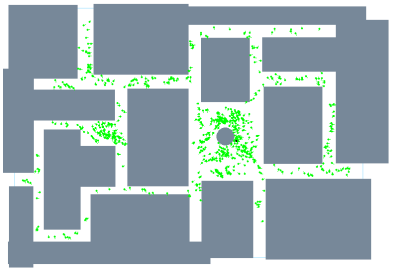

Abbildung 2: Unsicherheit der Lokalisation mit mehreren Ballungen

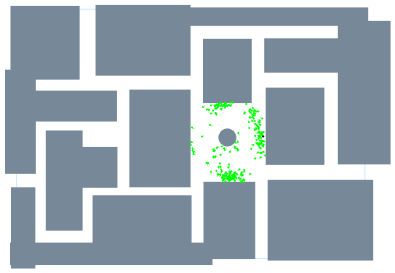

Abbildung 3: Bessere Lokalisation durch Konzentration der Partikel

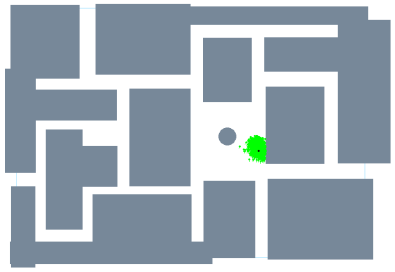

Abbildung 4: Erfolgreiche Lokalisation zur Position des Roboters

2.4 Grenzen und Weiterentwicklung

Der MCL-Algorithmus ist inzwischen eine der verbreitesten Techniken zur Roboter-Lokalisation. Er hat sich als robust und zweckmäßig herausgestellt, insbesondere weil er mit starkem Rauschen in den Messungen der Sensoren gut umgehen kann.[8] Außerdem können Partikelfilter je nach vorhandener Rechen- und Sensorleistung beliebig genaue (oder ungenaue) Ergebnisse liefern, was sie in Anwendungsgebieten attraktiv macht, in denen die Ressourcen begrenzt und benötigte Genauigkeit bekannt ist, wie z.B. in der Robotik.[9] Allerdings gibt es einige Probleme mit dem Algorithmus, wenn er unverfeinert und ohne genaue Anpassung an die jeweilige Umgebung verwendet wird.

- So konvergiert der Partikelfilter u.U. sehr schnell auf eine einzige Position auch in uneindeutigen Situationen, die im Hinblick auf die Gestalt tatsächlicher Umgebungen häufig vorkommen, z.B. aus architektonischen Gründen.[10]

- Die benötigte Populationsgröße des Partikelfilters wird für die Lösung den globalen Lokalisationsproblems mittels "naiver"Monte-Carlo-Lokalisation schon für recht kleine Umgebungen riesig.[11]

- Das Anwendunsgebiet von Partikelfiltern ist aus ähnlichen Gründen faktisch

[8] [Moreno et al., S.517]
[9] [Thrun, S.513]
[10] [Kootstra / de Beur, S.1107]
[11] [Moreno et al., S.517]

auch auf niedrig-dimensionale Umgebungen beschränkt, da die Anzahl der benötigten Partikel exponentiell mit der Anzahl der Dimensionen der Problemstellung wächst.[12]

- Die Verwendung der Markow-Ketten-Monte-Carlo-Methode bedeutet auch, dass bei jeder Messung eine vollständige Iteration aller nötigen Berechnungen durch das gesammte Sample gemacht werden muss, wodurch die Nachteile großer Partikelpopulationen besonders in den Vordergrund treten.[13]

Zur Verbesserung insbesondere der Skalierbarkeit des MCL-Algorithmus gibt es verschiedene Lösungsansätze, die an dieser Stelle allerdings nicht näher diskutiert werden sollen, da sie nicht nur den Umfang dieser Dokumentation sprengen würden, sondern auch in [Russel / Norvig (2012)] nicht betrachtet werden. Interessierte Leser werden z.B in [Moreno et al.] fündig.

Ähnliche Partikelfilter wie im MCL-Algorithmus können auch für den eng verwandten, aber deutlich mächtigeren Simultaneous-Localization-And-Mapping-Algorithmus (SLAM) und gleichzeitig zur Lokalisation von beweglichen Objekten in der Umgebung des Roboters verwendet werden.[14]

[12] [Thrun, S.514]
[13] [Moreno et al., S.517]
[14] [Russel / Norvig (2012), S.1134]

3 AIMA-Framework

Das AIMA-Framework stellt eine Grundlage von Algorithmen zum Thema Künstliche Intelligenz bereit. Die Algorithmen basieren dabei alle auf [Russel / Norvig (2009)]. Hierbei wird auf die dritte Auflage des Buches (AIMA3e) eingegangen. Es findet sich zu jedem Algorithmus Pseudocode, an welchem sich die Implementierung orientiert.

3.1 Bestandteile der Implementierung

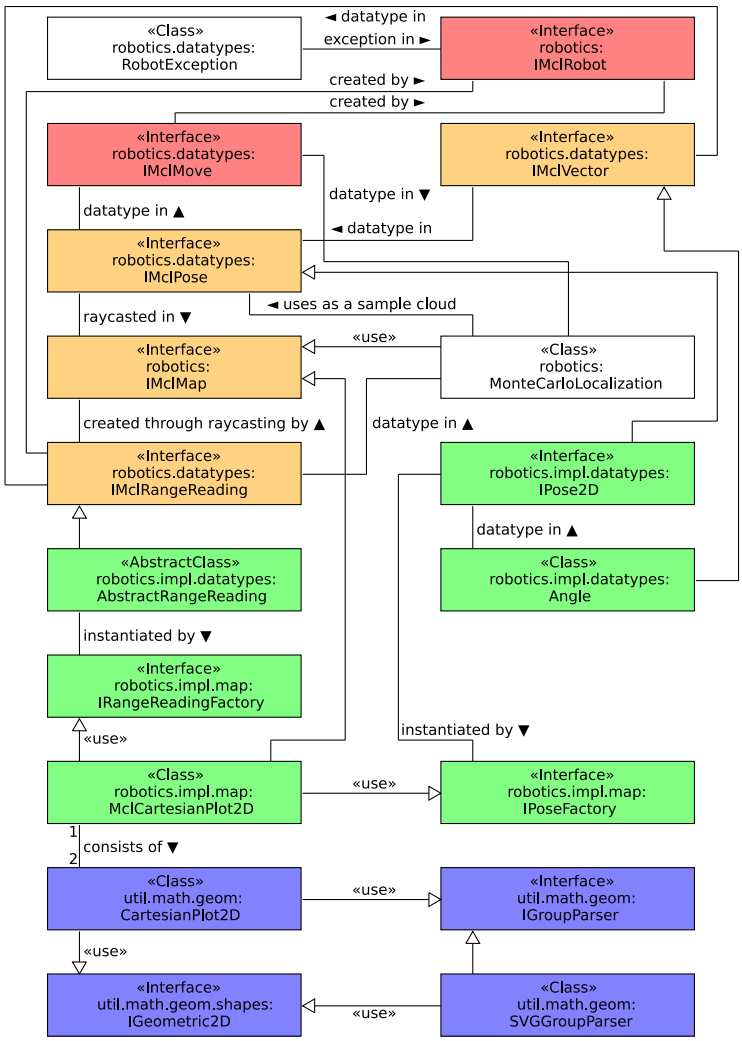

Abbildung 5: Zusammenhang zwischen den Klassen der MCL-Implementierung

Im Fall der Monte-Carlo-Lokalisation beschränkt sich dieser Pseudocode auf den Ablauf der einzelnen Schritte des Algorithmus. Vorausgesetzt wird, dass der Roboter dabei einen Sensor besitzt, über den er Entfernungsmessungen durchführen kann, sowie unspezifizierte Bewegungen, die jedoch zurückverfolgbar sein müssen, ausführen kann. Der MCL-Algorithmus ist Teil des Kapitels über Robotik, somit befinden sich alle Klassen der Monte-Carlo-Lokalisation im Paket *aima.core.robotics*.

Die Hauptklasse, welche den MCL-Algorithmus beinhaltet, *MonteCarloLocalization*, arbeitet generisch für eine n-dimensionale Umgebung mit den Interfaces *IMclPose*, *IMclRangeReading*, *IMclMove* und *IMclMap*. Eine Pose beschreibt dabei eine mögliche Position des Roboters in der Karte. Diese Posen werden ebenfalls als Partikelwolke verwendet. Eine Entfernungsmessung wird zum Einen von der Karte als Ergebnis eines Raycasts für eine Pose zurückgeliefert. Zum Anderen entsteht eine Entfernungsmessung durch den Roboter und wird dem Algorithmus übergeben. Eine Bewegung, die durch *IMclMove* repräsentiert wird, wird ebenfalls vom Roboter erzeugt und dem Algorithmus zugeführt.

Um die Monte-Carlo-Lokalisation generisch zu halten, sind alle Datentypen, die zur Kommunikation der einzelnen Komponenten verwendet werden, als Schnittstellen definiert. Dazu wird noch die Schnittstelle *IMcLVector* definiert, welche dazu dient, einen Winkel zwischen zwei Richtungen in der n-dimensionalen Umgebung zu beschreiben. Diese Schnittstelle wird daher als Datentyp zwischen *IMclRangeReading* und *IMclPose* verwendet.

Die Farben im Diagramm gruppieren die Klassen und Schnittstellen in mehrere Kategorien. Alle rot hinterlegten Schnittstellen sind in jedem Fall Teil der Problemumgebung und müssen daher bei Benutzung des Algortihmus spezifisch für den entsprechenden Roboter implementiert werden. Alle orange hinterlegten Schnittstellen müssen in n-dimensionalen Umgebungen implementiert werden. Für ein zweidimensionales Problem können alle Klassen und Schnittstellen verwendet werden, die grün hinterlegt sind. Dadurch müssen die orange hinterlegten Schnittstellen nicht mehr implementiert werden. Alle blau hinterlegten Klassen und Schnittstellen bilden die Implementierung der zweidimensionalen Karte. Sie befinden sich seperat im Paket *aima.core.util*.

Roboter

Der Roboter, definiert durch *IMclRobot*, dessen Position in der Problemumgebung bestimmt werden soll, muss Entfernungsmessungen liefern können. Weiterhin muss er eine beliebige Bewegung ausführen können und Diese zurückliefern.

Da die Kommunikation mit dem tatsächlichen Roboter störungsanfällig ist, kann eine *RobotException* geworfen werden, falls die aufgerufene Methode nicht erfolgreich ausgeführt wurde und eine weitere Ausführung der Monte-Carlo-Lokalisation unterbrochen werden sollte.

Karte der Umgebung

Die Monte-Carlo-Lokalisation benötigt eine Karte, definiert durch das Interface *IMclMap*, auf der die Partikel generiert werden und Partikel auf Gültigkeit ge-

prüft werden können. Die Karte muss zudem eine Methode zum Raycasting für gegebene Posen bieten. Eine Implementierung als zweidimensionaler kartesischer Graph findet sich in der Klasse *MclCartesianPlot2D*. Da es möglich ist, dass sich der Roboter nur in einem eingeschränkten Bereich bewegen kann, dessen Begrenzungen jedoch keine Hindernisse für den Entfernungssensor darstellen, ist es sinnvoll, die Karte in zwei Teile zu trennen. Der erste Teil, bezeichnet durch *MclCartesianPlot2D.OBSTACLE_ID*, enthält ausschließlich Hindernisse, die der entfernungsmessende Sensor wahrnehmen kann. Der zweite Teil, bezeichnet durch *MclCartesianPlot2D.AREA_ID*, enthält hingegen Flächen, auf denen der Roboter sich ausschließlich bewegen kann und nicht vom Entfernungssensor wahrgenommen werden können. Wichtig ist hierbei, dass die Hindernisse noch von den Bewegungsflächen abgezogen werden. Der Roboter kann sich also zu keinem Zeitpunkt in einem Hindernis befinden. Der kartesische Graph kann beliebig viele Hindernisse sowie Bewegungsflächen enthalten. Um das Raycasting zu Beschleunigen wird eine maximale Distanz, die der Sensor des Roboters messen kann, durch die Methode *setSensorRange* eingeführt, da alle Hindernisse auf der Karte, die weiter als diese maximale Distanz vom Ursprung des Strahls entfernt sind, beim Raycasting die gleiche Distanz liefern können.

Zur Verwendung des kartesischen Graphen ist es nötig, zwei Fabriken zu implementieren. Die Fabrik spezifiziert in *IPoseFactory* liefert eine Pose für einen gegebenen Punkt. Weiterhin kann die Fabrik überprüfen, ob die Orientierung einer Pose gültig ist. Die Fabrik spezifiziert in *IRangeReadingFactory* erzeugt eine Entfernungsmessung für einen gegebenen Wert.

Dieser kartesische Graph greift in seiner Implementierung auf das Paket *util.math.geom* zurück. Dieses Paket bietet Lösungen für zwei Probleme der algebraischen Geometrie, den *Point-Inside-Test* und das *Ray-Casting*. Ein *CartesianPlot2D* lädt dabei eine Karte aus einem *InputStream* durch einen Parser, der als *IGroupParser* definiert ist. Aktuell existiert nur ein Parser, der sich in der Klasse *SVGGroupParser* befindet, für SVG-Dateien mit einem eingeschränkten Satz an unterstützten Elementen. Es ist zu beachten, dass der dem SVG-Parser zugrundeliegende *XMLStreamReader* sehr langsam arbeitet, sobald im Dokument ein $<!DOCTYPE>$ Element spezifiziert wird. Es ist zu empfehlen, diese Elemente aus den verwendeten Dateien zu entfernen. Die folgenden Formen können mit dem SVG-Parser verwendet werden:

- *rect*
- *line*
- *circle*
- *ellipse*
- *polyline*
- *polygon*

Weiterhin ist es möglich und nötig alle verwendeten Formen mit dem *g*-Element zu gruppieren. Um die Datei mit dem *MclCartesianPlot2D* verwenden zu können,

muss die Datei zwei Gruppen, die jeweils mit der entsprechende ID für Hindernisse und Bewegungsflächen gekennzeichnet sind, enthalten. Jedes Element kann dabei das *transform*-Attribut angeben. In diesem Attribut sind jedoch nur die folgenden Umformungen zulässig:

- *translate*
- *scale*
- *rotate*

Die *skew*-Operationen und damit auch die eigentliche Matrixtransformation sind nicht möglich, da sie dafür sorgen (können), dass die ursprünglich horizontale Achse insbesonderer runder Elemente nicht mehr vertikal zur ursprünglich vertikalen Achse eben dieser Elemente liegt. Da die Implementierungen von *Circle2D* und *Ellipse2D* jedoch auf der Annahme beruhen, dass die beiden Achsen nicht ihren Bezug zueinander verlieren, können diese Transformationen nicht verwendet werden.

Das untergeordnete Paket *shapes* enthält alle geometrischen Formen, die in einem Karthesischen Graphen in dieser Implementierung verwendet werden. Jede dieser Formen besitzt dabei die Schnittstelle *IGeometric2D*, somit kann auf jeder Form ein zufälliger Punkt generiert werden, sowie der *Point-Inside-Test* und *Ray-Casting* durchgeführt werden. Jede Form kann außerdem ihr umgebendes Rechteck angeben. Mit diesen Rechtecken wird der *Point-Inside-Test* und das *Ray-Casting* beschleunigt.

Modelle

Da Bewegungen durch die Schnittstelle *IMclMove* abgebildet werden, ist auch das Bewegungsmodell in dieser Schnittstelle enthalten. Das Rauschen des Modells wird dabei durch die Methode *generateNoise* auf die Bewegung angewendet. Ebenso ist das Rauschen des Sensormodells in der Schnittstelle *IMclRangeReading* vorgesehen. Es sollte somit auf statische Werte zur einfachen Anpassung der Parameter zur Laufzeit zurückgegriffen werden.

3.2 Einrichtung zur Benutzung

Das AIMA-Framework kann zunächst von GitHub[15] erhalten werden. Das Framework besteht dabei aus drei Teilen: *aima-core*, *aima-gui* und *aimax-osm*. Der dritte Teil dient dazu, Open Street Map in Verbindung mit dem AIMA-Framework nutzen zu können, was hier nicht weiter betrachtet wird. *aima-core* ist ein Eclipse-Projekt, welches sämtliche implementierten Algorithmen enthält. Eine Liste dieser Algorithmen findet sich in der *README.md* des Stammverzeichnisses des Frameworks. Das Eclipse-Projekt kann dann importiert werden. Andererseits kann auch per Ant-Build eine *.jar*-Bibliothek erzeugt werden. Genauso kann mit dem Eclipse-Projekt *aima-gui*, welches graphische Oberflächen und Implementierungen für Beispielumgebungen enthält, verfahren werden.

[15] https://github.com/aimacode/aima-java/tree/AIMA3e

Soll das Framework jedoch nur in Verbindung mit den Lego Mindstorms verwendet werden, muss dieses Framework nicht manuell bereitgestellt werden, da die nötigen *.jar*-Bibliotheken bereits enthalten sind. Diese Bibliotheken enthalten jedoch nur die kompilierten Klassen sowie das Javadoc zu diesen Klassen. Soll gleichzeitig noch der Sourcecode zur Verfügung stehen, müssen die beiden Projekte im Eclipse-*workspace* geöffnet sein.

4 Graphische Oberfläche

Die Graphische Benutzeroberfläche zur Verwendung der Monte-Carlo-Lokalisation dient zwar lediglich als Beispielimplementierung, sie ist jedoch generisch gehalten, sodass sie für jegliche zweidimensionale Umgebung verwendet werden kann. Sie befindet sich im Paket *aima.gui.swing.demo.robotics*.

In der Klasse *GenericMonteCarloLocalization2DApp* wird dazu das Hauptfenster erzeugt, in dem der Großteil aller relevanten Steuerungsmodule (Buttons) und Datenausgaben (Tabellen, Karte, usw.) angezeigt werden. Es werden drei Panels erzeugt:

1. Das Panel *leftPanel* befindet sich auf der linken Seite des Fensters und enthält die Buttons zur Nutzung des Programms:

   - *Roboterinitialisierung*
   - *Laden einer Karte*
   - *Ausführen eines einzelnen Schrittes der Lokalisierung*
   - *Vollautomatische Lokalisierung*
   - *Entfernen aller Datenausgaben*
   - *Aufrufen der Einstellungen*

2. Das Panel *rightPanel* wird auf der rechten Seite erzeugt. Es enthält zum einen die letzte Entfernungsmessung in einem Textfeld und alle ausgeführten Bewegungen in einer Tabelle.

3. Im Zentrum befindet sich das Panel *centerPanel* auf dem die Karte mit der Klasse *MapDrawer* angezeigt wird. Die Zeichnung der Karte erfolgt sobald eine Karte durch den Benutzer geladen wurde. Dies wird dabei durch die Klasse *MapLoader* übernommen.

   Über Methoden wie *drawParticles* werden anzuzeigende Daten an die Karte übergeben. Außerdem ist es mit der Methode *scaleMap* möglich die Karte in ihrer Größe anzupassen. Mit einem Slider kann man die Karte vergrößern. Wenn die Karte das Sichtfeld überschreitet werden Jscrollbars, für die horizontale und die vertikale, aktiv, damit der User jeden Bereich der Karte ansehen kann.

Über den letzten Button im ersten Panel wird ein weiteres Fenster, welches in der Klasse *Settings* erzeugt wird, geöffnet, in dem sich alle Parameter des Algorithmus und der Problemumgebung anpassen lassen. Dabei existiert für jede Einstellung ein eigenes Textfeld in dem entsprechende Werte eingetragen werden können. Weiterhin ist es möglich, spezielle Felder zu definieren, sodass komplexere Einstellungen ebenfalls möglich sind. Mit der Klasse *AnglePanel* wird solch ein Feld für die Eingabe von Winkeln definiert. In der Beispielimplementierung wird diese Klasse dazu genutzt, einzustellen, in welche Richtungen der Roboter Entfernungsmessungen durchführen soll.

Da alle Einstellungen zentral verwaltet werden, können sie mit Hilfe der Methode *saveSettings* extern in einer Datei gespeichert und werden so bei einem Programmneustart wieder geladen. Sollte der Benutzer Änderungen in den Einstellungen vornehmen, muss er diese zunächst durch den *Save* Button festschreiben. Durch den *Revert* Button können jedoch die vorherigen Einstellungen neu geladen werden. Beim Speichern der Einstellungen wird jedoch eine laufende Lokalisation unterbrochen und mit einer neuen Partikelwolke wieder begonnen.

5 Monte-Carlo-Lokalisation auf Lego Mindstorms NXT

Um die im AIMA-Framework implementierte Monte-Carlo-Lokalisation auf den Lego Mindstorms NXT benutzen zu können, müssen alle Komponenten, die auf Seite der Problemumgebung fallen, implementiert werden. Dazu gehören der Roboter und die Bewegungen, sowie mögliche Posen des Roboters. Als Karte kann der bereits implementierte kartesische Graph des AIMA-Frameworks genutzt werden. Weiterhin ist eine grafische Oberfläche nötig, um eine Verbindung mit dem NXT aufbauen zu können. Auf dem mit leJos NXJ bestückten NXT muss schließlich noch ein Programm vorhanden sein, welches Befehle vom Hauptprogramm entgegen nimmt und auf Diese antwortet. Die zwei Projekte für die GUI und den NXT sollten erst nach der Einrichtung des leJos NXJ Frameworks in Eclipse importiert werden.

5.1 LeJos NXJ Framework

Das leJos NXJ Framework[16] ermöglicht die Programmierung der Lego Mindstorms mit Java. Es besteht aus einer Tiny Java Virtual Machine, die eine Teilmenge von Java unterstützt, sowie eine Bibliothek zur Einbindung in andere Java-Anwendungen zur Steuerung von und Komminkation mit den NXTs.

5.1.1 Einrichtung zur Benutzung

Um leJos erfolgreich in Eclipse verwenden zu können, empfiehlt es sich, das leJos NXJ Eclipse Plug-In zu installieren. Dazu wird im Eclipse Marketplace, erreichbar im Punkt *Help > Install New Software...*, das Repository[17] hinzugefügt. Daraufhin wird das Item *leJos NXJ plugin* eingeblendet, welches nun ausgewählt wird und im nächsten Schritt installiert wird. Zusätzlich müssen noch die Programme und Bibliotheken von leJos NXJ bereitgestellt werden. Dazu wird das Archiv zur manuellen Installation[18] heruntergeladen und entpackt. Abschließend muss dem Eclipse Plug-In noch mitgeteilt werden, an welcher Stelle sich das entpackte Archiv befindet. Dazu wird in den Einstellungen von Eclipse der Reiter *leJos NXJ* ausgewählt. In diesem Reiter muss dann der Pfad für *NXJ_HOME* festgelegt werden. Weiterhin sollte der Haken bei *Run program after upload* für den *run mode* sowie den *debug mode* entfernt werden.

Nachdem leJos in Eclipse eingebunden ist, kann entweder ein neues leJos Projekt angelegt werden oder bestehende Projekte in leJos Projekte umgewandelt werden. Dabei ist zwischen zwei Arten von Projekt zu unterscheiden. Es gibt zum einen ein PC-Projekt, welches die PC-Bibliothek von leJos verwendet und immernoch mit dem normalen Java Compiler erstellt wird und in der normalen Java Laufzeitumgebung arbeitet. Die zweite Art von Projekt ist ein NXT-Projekt, welches mit dem

[16] http://www.lejos.org/nxj.php
[17] http://www.lejos.org/tools/eclipse/plugin/nxj/
[18] http://www.lejos.org/nxj-downloads.php

leJos Compiler erstellt wird und auf den NXT-Bausteinen in der leJos NXT Laufzeitumgebung arbeitet. Daraus folgt, dass diesen NXT-Projekten nur der begrenzte Umfang der leJos NXT Laufzeitumgebung zur Verfügung steht.

Weiterhin weißt leJos einige Besonderheiten auf, die zu beachten sind. Zunächst muss für den in der leJos PC-Bibliothek verwendeten Bluetooth-Stack ein 32-Bit-Java installiert sein und das Programm mit dieser Java Version gestartet werden. Folglich sollte Eclipse direkt mit dieser Java Version gestartet werden. Es empfiehlt sich außerdem, leJos unter Windows zu benutzen, da unter Linux weitere Probleme auftreten können.

Da für die USB Verbindung mit dem NXT ein spezieller Treiber nötig ist, ist es leichter, Programme per Bluetooth auf den NXT aufzuspielen. Eine Bluetooth Verbindung wird für die Steuerung des NXT ohnehin in der Implementierung aufgebaut.

5.1.2 Debugging unter leJos

Exceptions in der leJos Laufzeitumgebung führen dazu, dass auf dem Bildschirm der Stack in einer verkürzten Form dargestellt wird. Beispielsweise wird ausgegeben:

$$Exception : 28$$
$$at\ 65 : 35$$
$$at\ 55 : 8$$

Hierbei gibt die erste Zeile die Klassennummer der Exception an. Jede weitere Zeile steht dabei für ein Frame des Stacks. Die erste Zahl gibt die Methodennummer an, die zweite ist der *program counter*. Diese Zahlen lassen sich durch das *NXJDebugTool* auflösen. Dieses Kommandozeilenhilfsprogramm ist im *bin*-Verzeichnis der lejos Paketes enthalten. Um dieses Programm verwenden zu können, ist die *.nxd*-Datei nötig, welche sich normalerweise mit der *.nxj*-Datei im Stammverzeichnis des entsprechenden Projektes befindet. Der Pfad zu ihr wird auf den Parameter *-di* folgenden spezifiziert. Weiterhin wird mit *-c* die erste folgende Zahl als Exceptionklasse untersucht und mit *-m* die letzten beiden Zahlen als Stack-Frame. Eine vollständige Kommandozeileneingabe würde für die Debug-Datei *MclDaemon.nxd*, falls sich diese im selben Verzeichnis wie das Hilfsprogramm befindet, wie folgt aussehen:

$$nxjdebugtool - di\ MclDaemon.nxd - c - m\ 28\ 65\ 35$$

Diese Eingabe liefert, dass eine *ArrayIndexOutOfBoundsException* in der Methode *MclDaemon.run()* aufgetreten ist. Der PC liefert dabei, dass die Exception in der Datei *MclDaemon.java* in der Zeile 276 aufgetreten ist. Es ist aber auch ein Remote-Debugger[19] im leJos Framework enthalten.

[19] http://www.lejos.org/nxt/nxj/tutorial/ErrorHandlingAndDebugging/ErrorHandling_and_debugging.htm

5.2 Roboterdesign

Das Design des für die Verwendung mit den in der TH Köln vorhandenen Testumgebungen entworfenen Roboters baut auf den mit dem Prototypen aus dem vorherigen Semester gemachten Erfahrungen auf. Ziele der Verbesserungen waren erstens das Einsparen von Bauteilen und die Vereinfachung der Konstruktion, um das Nachbauen zu erleichtern, was auch gelungen ist, sowie zweitens eine Verringerung der Höhe des Fahrzeugs.

5.2.1 Sensoren

Da der Sensormast der am höchsten stehende Teil des Systems ist, und dieser den Autoren beim Prototypen gefährlich hoch für die Testumgebung erschien, gleichzeitig aber weiterhin hoch über dem Rest des Fahrzeugs montiert sein sollte (beides um Fehlmessungen zu vermeiden) waren erhebliche Designanstrengungen nötig, um die Höhe des Roboters abzüglich der Höhe des Sensormastes zu verringern. Der um 360 Grad drehbare Sensormast ist mit dem Ultraschallsensor bestückt und steht mittig, relativ weit hinten auf dem Roboter. Dies hat einerseits den Vorteil, dass das Bedienfeld gut zugänglich ist, erfordert aber andererseits einen höheren Mast, um Fehlmessungen am eigenen Fahrzeug zu verhindern.

Wie bereits der Prototyp vom letzten Semester ist auch der aktuelle Entwurf sowohl mit einem Farb- als auch mit einem Helligkeitssensor ausgestattet. Dies hat zu besseren Ergebnissen bei einer der Testumgebungen geführt. Es konnte durch geeignete Testreihen ermittelt werden, dass beide Sensoren auf keinen Fall näher am Boden montiert werden dürfen, als sie es im aktuellen Entwurf sind. Andernfalls sind keine für die Testumgebung nutzbaren Messwerte zu erwarten.

5.2.2 Aktuatoren

Vom Prototypen übernommen wurde insbesondere der Raupenantrieb, da diese Form der Lokomotion dem Roboter erlaubt, Drehungen auf der Stelle zu vollführen. Zwar könnte diese Eigenschaft auch mit einem dreirädrigen Entwurf erreicht werden, aber gegenüber diesem hat der Raupenvortrieb den Vorteil, dass es kein einzeln stehendes Rad gibt, das eventuell blockieren kann. Der Raupenantrieb des aktuellen Entwurfs hat im Unterschied zum Prototypen ein Laufrad mehr pro Seite, was bessere Traktion verspricht. Die Raupenketten erstrecken sich nicht über die ganze Länge des Fahrzeugs, was zweifellos wünschenswert gewesen wäre, aber auf der Grundlage der zur Verfügung stehenden Bauteile nicht zu realisieren war. Der Roboter hängt am vorderen Ende weiter über die Lauffläche der Raupen hinaus als am hinteren, da sich alle drei Elektromotoren und damit der Hauptteil der Masse des Systems hinten befinden. Auf diese Weise wird eine bessere Balance des Fahrzeugs erreicht. Die Raupenketten sind durch stoßstangenartige Abweiser vor Kollisionen mit der Umgebung geschützt, um zu verhindern, dass sich der Roboter an Hindernissen 'festfrisst'.

Die Verringerung der Höhe des Roboters wurde unter anderem durch den umgekehrten Einbau der Antriebsmotoren erreicht. Hierdurch wird eine geringfügige

Verbreiterung des Gesamtfahrzeugs bedingt, die nicht weiter bedenklich erscheint. Ungünstiger ist die ebenfalls nötig gewordene Aufhängung der Laufräder auf langen Achsen, was sowohl die Wahrscheinlichkeit des Verlierens einer Raupenkette während einer Drehbewegung des Roboters als auch die eines Achsbruchs vergrößert. Da das Fahrzeug allerdings nicht für die Geländefahrt vorgesehen ist, schienen den Autoren diese Risiken vertretbar. Aus dem gleichen Grund wurde auf eine Federung der Laufradaufhängung verzichtet und eine geringe Bodenfreiheit des Roboters hingenommen.

Der neue Sensormast wird über eine Zahnradübersetzung gedreht. Auf diese Weise war es möglich, den zuständigen Motor niedriger zu montieren, ohne eine allzu große einseitige Gewichtsverteilung zu erzeugen.

5.2.3 Sekundäre Designmerkmale

Es ist gelungen, den NXT-Computer gegenüber dem Prototypen invertiert einzubauen, was den entscheidenden Vorteil hat, dass das Bedien- und Anzeigefeld sich nun auf der Oberseite des Fahrzeugs befindet. Dies erlaubt die Bedienung und Fehlerdiagnose des Roboters ohne ihn zu bewegen.

Die Kabelführung des Roboters orientiert sich an der Vorgabe, dass die Kabel nicht die 'Sichtbahn' der Sensoren kreuzen dürfen. Daher befinden sich die Kabelführungen am Fahrzeugboden und eng an den Vortriebsmotoren anliegend.

Als nachteilhaft muss beim aktuellen Entwurf der schlecht erreichbare Zugang zum Ladekabel-Anschluss betrachtet werden, ein Problem, das sich gegenüber dem Prototypen sogar noch verschärft hat. Um den Akku aufzuladen, ist es am einfachsten, den Sendemast sammt Motor zu demontieren. Dieses Vorgehen erlaubt es, das Ladekabel schnell und unkompliziert anzuschließen. Der Akkumulator selbst ist überhaupt nicht mehr leicht zugänglich, ein Austausch würde die Demontage praktisch des gesamten Systems bedeuten. Dieser Nachteil erkauft die bessere Zugänglichkeit des Bedienfeldes. Er erscheint den Autoren in sofern als verschmerzbar, als der Akku mit Hilfe des Ladekabels geladen werden kann, während er im Fahrzeug verbleibt.

6 Möglichkeiten zur Fortführung der Entwicklung

Die vorliegende Implementierung orientiert sich an der in [Russel / Norvig (2012)] vorgestellten "naiven"Monte-Carlo-Lokalisation. Sie könnte ohne größeren Aufwand erweitert werden, um die im 2. Abschnitt angerissenen Probleme mit der Skalierbarkeit des Algorithmus anzugehen. Darüber hinaus würden sich allerdings auch einige weitergehende Modifikationen anbieten, die das Forschungsgebiet der Robotik weiter erschließen.

- Da der Partikelfilter des MCL eine Markow-Kette abbildet, liegt es nahe, zur Verbesserung der Genauigkeit die Stufe des zugrundeliegenden Markow-Prozessmodels zu erhöhen.[20] In der vorliegenden Implementierung ist dies ein Modell erster Stufe, d.h. es werden nur die Daten der aktuellen Iteration zur Vorhersage der nächsten herangezogen und diese daher als allein abhängig von der letzten angesehen. Obwohl diese Betrachtungsweise nicht falsch ist, kann man u.U. zu besseren Ergebnissen (weniger falsch-positive Lokalisierungen) gelangen, wenn man mehrere konsekutive Messung-und-Bewegungs-Zyklen in die Berechnungen mit einbezieht. Erst hierdurch würden die Vorteile der Verwendung eines Markow-Modells wirklich zum tragen kommen.

- Ein weiteres Feld wäre die Erweiterung des Systems zur vollen Simultaneous-Lokalization-and-Mapping Funktionalität, bei der dem Roboter keine Karte der Umgebung zur Verfügung steht und er sie sich durch Messungen an den Hindernissen erst selbst erstellen muss.

- Daneben gäbe es natürlich auch die Möglichkeit, das Design des physischen Roboters weiter zu verbessern oder mit dem Einbau weiteren Sensoren zu experimentieren, z.B. mit einem Kompass. Dies hätte ebenfalls das Potential, die Lokalisation zu beschleunigen und die Genauigkeit zu verbessern.[21]

[20] [Russel / Norvig (2004), S.663]
[21] [Russel / Norvig (2004), S.665]

Literatur

[Adiprawita et al.] Adiprawita, Widyawardana et al.: 'A Novel Resampling Method for Particle Filter for Mobile Robot Localization', *International Journal on Electrical Engineering and Informatics* Volume 3, Number 2 (2011) S.165-177.

[Kootstra / de Beur] Kootstra, Gert / de Boer, Bart: 'Tackling the premature convergence problem in Monte-Carlo localization' *Robotics and Autonomous Systems*, **57** (2009) S.11071118.

[Moreno et al.] Moreno, Luis et al: 'Differential Evolution Markov Chain Filter for Global Localization', *Journal of intelligent Robot Systems* **82** (2016) S. 513-536.

[Russel / Norvig (2004)] Russel,Stuart / Norvig, Peter (Hg.): *Künstliche Intelligenz - Ein moderner Ansatz*, 2. Auflage, Pearson, u.A. München (2004).

[Russel / Norvig (2009)] Russel,Stuart / Norvig, Peter (Hg.): *Artificial Intelligence - A modern Approach*, 3. Auflage, Pearson, u.A. München (2009).

[Russel / Norvig (2012)] Russel,Stuart / Norvig, Peter (Hg.): *Künstliche Intelligenz - Ein moderner Ansatz*, 3. Auflage, Pearson, u.A. München (2012).

[Sarkar et al.] Sarkar, Biswajit et al.: 'A novel method for computation of importance Weights in Monte Carlo localization in line segment-based maps', *Robotics and Autonomous Systems* **74** (2015) S. 5165.

[Thrun] Thrun, Sebastian: *Particle Filters in Robotics*, in: Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence (UAI) (2002) S.511-518.

[Ulam / Metropolis] Ulam, Stanislaw / Metropolis, Nicholas: 'The Monte Carlo Method', *Journal of the American Statistical Association* Vol. 44, No. 247 (September 1949) S. 335-341.