Fachbereich 4: Informatik

Autonom-Explorierender Hexapod-Roboter

Bachelorarbeit
zur Erlangung des Grades
Bachelor of Science
im Studiengang Computervisualistik

vorgelegt von

Christian Schlöffel

Betreuer: Dipl.-Inform. Frank Neuhaus, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
Erstgutachter: Dipl.-Inform. Frank Neuhaus, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
Zweitgutachter: Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau

Koblenz, im Januar 2013

Kurzfassung

Das Ziel der Bachelor-Arbeit ist es, einen existierenden sechsbeinigen Kleinroboter zu programmieren, der dann in der Lage sein soll, seine Umgebung autonom zu explorieren und eine Karte selbiger zu erstellen. Zur Umgebungswahrnehmung soll ein Laserscanner integriert werden. Die Erstellung der Karte sowie die Selbstlokalisation des Roboters erfolgt durch Anbindung des Sensors an ein geeignetes SLAM (Simultaneous Localization and Mapping) Verfahren. Die Karte soll die Grundlage für die Pfadplanung und Hindernisvermeidung des Roboters bilden, die ebenfalls im Rahmen dieser Arbeit entwickelt werden sollen. Dazu werden sowohl GMapping als auch Hector Mapping verwendet und getestet. In der Arbeit wird zudem ein Explorationsalgorithmus beschrieben, mit welchem der Roboter seine Umgebung erkunden kann. Die Umsetzung auf dem Roboter erfolgt innerhalb des ROS (Robot Operating System) Frameworks auf einem „Raspberry Pi“ Miniatur-PC.

Abstract

Th goal of this Bachelor thesis was programming an existig six-legged robot, which should be able to explore any environment and create a map of it autonomous. A laser scanner is to be integrated for cognition of this environment. To build the map and locate the robot a suitable SLAM(Simultaneous Localization and Mapping) technique will be connected to the sensor data. The map is reported to be the robots base of path planning and obstancle avoiding, what will be developed in the scope of the bachelor thesis, too. Therefore both GMapping and Hector SLAM will be implemented and tested. An exploration algorithm is described in this bachelor thesis for exploring the robots environment. The implementation on the robot takes place in the space of ROS(Robot Operating System) framework on a "Raspberry Pi" miniature PC.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Vereinbarung der Arbeitsgruppe für Studien- und Abschlussarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	ja ☒	nein ☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	ja ☒	nein ☐

Koblenz, den 27. Februar 2013

Inhaltsverzeichnis

1 Einleitung **11**
1.1 Aufbau der Arbeit . . . 12
1.2 Motivation . . . 12

2 Grundlagen **13**
2.1 Verschiedene Filter . . . 13
2.1.1 Bayes Filter . . . 14
2.1.2 Kalman Filter . . . 16
2.1.3 Extended Kalman Filter . . . 20
2.1.4 Partikelfilter . . . 22
2.2 SLAM . . . 24
2.2.1 SLAM mit Kalman Filtern . . . 25
2.2.2 SLAM mit Partikelfiltern . . . 28
2.2.3 FastSLAM 1 . . . 28
2.2.4 FastSLAM 2.0 . . . 30
2.3 Exploration . . . 30
2.4 Ansätze zur Pfadfindung . . . 31
2.5 ROS . . . 32
2.5.1 Architektur von ROS . . . 32

3 Hardware **35**
3.1 Raspberry Pi . . . 35
3.2 Der Roboter . . . 36
3.2.1 Fortbewegung . . . 40

4 Praktischer Teil **43**
4.1 Nutzung von ROS im Rahmen der Arbeit . . . 43
4.2 Kommunikation mit dem Roboter . . . 44
4.3 Kartieren und Lokalisieren . . . 44
4.3.1 GMapping . . . 44
4.3.2 Hector SLAM . . . 47

4.4 Exploration 50
4.4.1 Distanztransformation 50
4.4.2 Pfadtransformation 51
4.4.3 Verwendeter Explorationsalgorithmus 52
4.4.4 Meijster Distanzkarte 54

5 Experimente **57**
5.1 Kartierung 57
5.2 Roboter 59
5.3 Exploration 59
5.4 Software + ROS 60
5.5 Kompletter Aufbau 63

6 Fazit und Ausblick **65**

A Schaltplan und Belegungsplan der Platine **67**

Abbildungsverzeichnis

2.1 Funktionsweise des Kalman Filter 17
2.2 grobe Funktionsweise der Noden in ROS [ros] 32

3.1 Raspberry Pi verbaut auf dem Roboter 36
3.2 Dritter Prototyp des Roboters 37
3.3 Erster Prototyp des Roboters 37
3.4 Zweiter Prototyp des Roboters 38
3.5 Aktueller Prototyp von der Seite. Die Beine sind aus Aluminium gefertigt und der Rest noch aus einer leichten Epoxidharzmischung. 38
3.6 Aktueller Prototyp von oben. Zu sehen ist die neue Beinstruktur, sowie die neuste Platine des Roboters. 39
3.7 Prototp der Platine zur Steuerung des Roboters 39
3.8 Laufschema des Roboters 40
3.9 Prototyp eines Beines des Roboters 40

4.1 Gewichte der Verteilungen des Zustandes und der Messungen bei Schätzung der Roboterpose in GMapping 45
4.2 Prinzip von Hector SLAM [KMSK11] 47
4.3 extrahierte Linien aus einer Messung zur Berechnung der EKF-Lokalisierung 49
4.4 Karte nach Hindernistransformation. Je heller der Wert, desto weiter ist ein Hindernis entfernt [Wir07]. 52
4.5 Explorationstransformation visualisiert 53
4.6 Distanztransformation nach Meijster 54

5.1 Falsche Karte erstellt von GMapping ohne Odometriedaten 58
5.2 Karte erstellt von GMapping mit Odometriedaten eines Scanmatchers 58
5.3 Karte erstellt von Hector SLAM 59
5.4 Abbildung zeigt gefundene Grenzen auf Testdaten. Grenzen sind rot markiert. Die längste Grenze ist grün markiert. 61

5.5 Extrahierte Grenzen in einem Binärbild visualisiert. D.ent zur Weiterverarbeitung der Daten. 61
5.6 GUI in Verbindung mit empfangenen Daten des SLAM Verfahrens . 62
5.7 GUI mit verschiedenen Visualisierungen 62
5.8 Karte der Wohnung, die als Testumgebung diente. Kleinere Objekte, wie Stühle oder Tischbeine wurden auch als Hindernisse erkannt und können somit umgangen werden. 63

A.1 Belegungsplan der Platine, die auf dem Roboter verwendet wurde. . 67
A.2 Schaltplan der Platine des Roboters. 68

Kapitel 1

Einleitung

Roboter sind in der heutigen Zeit kaum noch wegzudenken. In fast jedem Gebiet sind sie anzufinden und sie können viele Arbeiten erleichtern oder komplett übernehmen. Dabei werden die Anforderungen und auch die Aufgaben, die ein Roboter verrichten soll, immer komplexer. Das größte Ziel in diesem Gebiet ist ein Roboter, der vollkommen autonom und selbstständig handelt. Dabei soll er selbstständig Entscheidungen treffen und Probleme lösen. Da dieser Zustand zur Zeit noch nicht möglich ist, wird versucht, den Roboter Teilgebiete autonom bewältigen zu können, wie z. B. Roboter, die in einer Fabrik arbeiten. Sie haben eine feste Aufgabenstellung, welche sich nicht verändert und lösen diese dann. Ein weiteres Teilgebiet ist das selbstständige Erkunden und Kartieren der Umgebung. Solche Roboter können beispielsweise nach Umweltkatastrophen, wie z. B. Hochwasser, eingesetzt werden, um Verletzte zu finden. Der Vorteil ist, dass bei der Suche kein Menschenleben in Gefahr gerät und Roboter je nach Größe auch an schwer zugängliche Orte gelangen können. Jedoch stellt sich hier die Frage, wie dieses Szenario umgesetzt werden kann. Eines der größten Probleme ist dabei das SLAM(Simultaneous Localization and Mapping)Problem. Der Roboter muss gleichzeitig seine Umgebung kartieren aber auch wissen, wo auf dieser Karte er sich befindet. Das Problem besteht darin, dass der Roboter ohne bekannte Position keine Karte aufgrund von Messdaten bilden kann, da diese ja lokal gemessen werden. Wiederum kann die Position nur auf einer bekannten Karte erstellt werden. Das bedeutet, beides muss abhängig voneinander bestimmt werden. Ohne das Wissen der Position könnte er sich bei der Exploration(Erkunden der Umgebung), wie auch bei anderen Einsatzgebieten, nicht sinnvoll auf der Karte bewegen. Die Exploration auf der erstellten Karte bietet die Möglichkeit, dass der Roboter keine angefertigte Karte der Umgebung benötigt, oder eine Person, die den Roboter durch die Umgebung steuert. Er ist dabei vollkommen autonom. Ziel der Arbeit ist es, ein Verfahren zur Exploration zu entwickeln und dieses auf einem geeigneten Roboter zu implementieren. Dieser Roboter wurde eigens für dieses Thema entwickelt und wird in dem Kapitel

Hardware kurz vorgestellt. Der Roboter sollte in der Lage sein, seine Umgebung selbstständig zu Erkunden und dabei Hindernissen auszuweichen. Dazu wurden im Rahmen der Arbeit verschiedene Methoden getestet, um das SLAM-Problem zu lösen. Diese werden im Kapitel Praktischer Teil eingeführt und später in dem Kapitel Experimente evaluiert. Die Auswahl bestand hier zwischen GMapping und Hector SLAM. Es wurde sich anhand der einfachen Umsetzung und der guten Ergebnisse für Hector SLAM entschieden.

1.1 Aufbau der Arbeit

Die Arbeit wurde in vier grobe Teile gegliedert. In dem ersten Teil werden die theoretischen Grundlagen zu den verwendeten Verfahren und Techniken erklärt. Dazu gehören mathematische Grundlagen, wie stochastische Filter, aber auch eine kurze Einführung in die verschiedenen Fachtermini. In dem Teil Hardware wird kurz der verwendete Roboter und auch das verwendete System vorgestellt. Die Umsetzung der Theorie in die Praxis wird dann im Praktischen Teil vorgestellt und zuletzt in dem Kapitel Experimente getestet und evaluiert.

1.2 Motivation

Das Projekt hat als Hobbyprojekt angefangen. Das Ziel war es, einen Hexapod zu entwickeln. Dies ist ein 6-beiniger Roboter, wobei jedes Bein 3 Freiheitsgrade hat. Dadurch besitzt er eine hohe Beweglichkeit und kann sich auch auf schwierigem Gelände fortbewegen. Er wird dabei über WLAN gesteuert. An diesem Thema war besonders die Komplexität der Bewegung und die großen Einsatzmöglichkeiten interessant. Dies ermöglicht ein Einsetzen in Aufklärung oder auch in Rettungseinsätzen. Je nach Größe des Roboters kann sich dieser durch jedes Gebiet, wie z. B. Trümmer, bewegen.

Kapitel 2

Grundlagen

In diesem Kapitel werden alle Bestandteile, die in der Arbeit Relevanz haben, vorgestellt und die mathematischen Theorien hierzu erläutert. Es werden zuerst grundlegend verschiedene Filter vorgestellt,welche zum Lösen von SLAM verwendet werden können und in der Arbeit Relevanz hatten. Dazu zählen die verschiedenen Umsetzungen des Bayes Filters, welcher Annahmen über einen Zustand über eine Wahrscheinlichkeitsdichte darstellt. Danach wird das Prinzip der Exploration und die allgemeine Pfadfindung erläutert. Am Ende wird auf das verwendete Framework ROS und dessen Architektur verwiesen. Die praktische Umsetzung der vorgestellten Theorien und Konzepte wird dann in den folgenden Kapiteln vertieft.

2.1 Verschiedene Filter

In der Robotik wird unter Anderem versucht, den aktuellen Zustand eines Roboters zu schätzen. Dieser Zustand kann aus verschiedensten Attributen bestehen, wie z. B. der Position und Geschwindigkeit. Das Schätzen dieses Zustandes kann mit verschiedenen Methoden realisiert werden. Jede Methode hat dabei gemein, dass sie den aktuellen Zustand anhand verschiedener Kontrolldaten, wie z. B. der Bewegungsmessung des Roboters, schätzt. Dabei müssen Ungenauigkeiten der Messungen in die Berechnung eines Zustandes einfließen. Ungenauigkeiten, auch Unsicherheiten genannt, können durch ungenaue Messungen oder ungenaue Kontrolldaten entstehen. Sensoren selbst haben schon eine gewisse Varianz in der Genauigkeit ihrer Messungen, auch Rauschen genannt. Diese Ungenauigkeiten würden sich immer weiter aufaddieren und so falsche Ergebnisse liefern, die sich mit der Zeit nur weiter verschlechtern. Die beste Möglichkeit, eine gute Repräsentation des Zustandes zu gewährleisten, ist eine Schätzung des Zustandes. Es wird versucht, den Zustand unter Einbeziehen von Ungenauigkeiten, bestmöglich anzunähern. Für ein optimales Schätzen kommen oft stochastische Filter zum Einsatz, um die Daten,

die die jeweiligen Sensoren senden, zu verarbeiten und den wahrscheinlichsten Zustand eines Roboters zu finden. Der Bayesfilter, eines der grundlegendsten Modelle in diesem Gebiet, berechnet den Zustand unter Einbeziehen des alten Zustandes und verschiedener Kontrolldaten. Im Folgendem werden einige Filter, die auch bezüglich der Arbeit wichtig waren, vorgestellt und erläutert. Die vorgestellten Filter basieren alle auf dem Modell des Bayes Filter. Später wird dann auch die genaue Anwendung in Bezug auf das SLAM Verfahren gezeigt und inwiefern diese Verfahren ihren praktischen Nutzen haben.

2.1.1 Bayes Filter

Der Bayes Filter ist der wohl am häufigsten instanziierte Filter, wenn es darum geht, Schätzprobleme über den Zustande eines Roboters zu lösen. Das Besondere an dieser Art von Filter ist, dass er den vorherigen Zustand mit in die Schätzung einfließen lässt, also rekursiv den aktuellen Zustand bestimmt. Der Bayes Filter ist gut dazu geeignet, Schätzungen über den Zustand von Objekten in der Umgebung zu machen unter Berücksichtigung beliebig vieler Daten, wie z. B. Messdaten. So kann in einem Haus der Zustand einer Tür anhand des alten Zustandes der Tür und der Messung des aktuellen Zustandes geschätzt werden [TBF05]. Das Beispiel ist so jedoch nur umsetzbar, da dieser Fall einen endlichen Zustandsraum besitzt und das Problem diskret ist. Wieso das hier der Fall ist, wird später noch deutlicher. Die Zustandsschätzung im Bayes Filter sieht wie folgt aus

$$p(x_t|z_{1:t-1}) = \int p(x_t|x_{t-1})p(x_{t-1}|z_{1:t-1})\ \mathrm{d}x_{t-1} \tag{2.1}$$

Die Annahme hier ist, dass jeder Zustand durch alle vorherigen Messungen abgebildet werden kann. Diese Annahme kann rekursiv dargestellt werden, sodass ein Zustand auf dem vorherigen Zustand beruht. Die Vorhersage des neuen Zustandes bezieht sich dann zum einen auf den vorherigen Zustand $p(x_t|x_{t-1})$ und zum anderen auf die Wahrscheinlichkeit, dass der alte Zustand durch die Messungen bis zu diesem Zeitpunkt abgebildet wurde. Dabei wird über das Produkt der beiden Verteilungen integriert. Bei dem obigen Beispiel könnte hier das Integral durch eine Summe über die Wahrscheinlichkeiten aller möglicher Zustände ersetzt werden. Daraus folgt dann die Vorhersage des neuen Zustandes x_t. In der Robotik kommt hier noch die Bewegung u_t, die z. B. durch Odometrie bestimmt wird, hinzu. Damit sieht die Formel dann wie folgt aus

$$p(x_t|z_{1:t-1}, u_{1:t-1}) = \int p(x_t|x_{t-1}, u_t)p(x_{t-1}|z_{1:t-1}, u_{1:t-1})\mathrm{d}x_{t-1} \tag{2.2}$$

Zusätzliche Kontrolldaten können als zusätzliche Bedingung hinzugefügt werden. Dies kann mit beliebig vielen Parametern gemacht werden. Doch auch hier ist

schon zu erkennen, dass das Integral dadurch aufwändiger zu berechnen ist. Nach der Berechnung dieser Vorhersage wird eine Korrektur mittels

$$p(x_t|z_{1:t}) = \frac{p(z_t|x_t)p(x_t|z_{1:t-1})}{p(z_t|z_{1:t-1})} \tag{2.3}$$

vorgenommen, oder im konkreten Fall mit den Kontrolldaten u_t und u_{t-1} in den Bedingungen, wie oben. Diese Formel verwendet das Bayes Theorem, um die Wahrscheinlichkeiten aufzuteilen. Damit bekommt man einen Teil, der von dem neuen Zustand x_t unabhängig ist. Diese Formel teilt die Wahrscheinlichkeit, dass ein Zustand von allen bisherigen Messungen abgebildet wird, in zwei Teile. Der erste Teil beschreibt die Wahrscheinlichkeit, dass der geschätzte Zustand von der aktuellen Messung abgebildet wird. Diese Wahrscheinlichkeit beschreibt den Fehler, dass eine Messung etwas anderes misst, als den aktuellen Zustand. Die zweite Wahrscheinlichkeit im Zähler beschreibt die aktuelle Zustandsschätzung, wie sie oben bestimmt wurde. Der Term $\frac{1}{p(z_t|z_{1:t-1})}$ wird durch η ersetzt. Dies dient nur zur Normierung der Messungen und ist unabhängig von dem Zustand x_t. Mit dieser Korrektur werden die neuen Messungen der Schätzung des aktuellen Zustandes hinzugefügt. Somit nehmen die Messungen erst nach der übrigen Schätzung Einfluss auf die Zustandsschätzung.

Algorithm 1 Algorithmus des Bayesfilter

Input: $bel(x_{t-1})$ – Letzte Zustandsschätzung
Input: u_t – Kontrolldaten
Input: z_t – Messdaten
Output: $bel(x_t)$ – neue Zustandsschätzung

for all x_t
{
$\overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1})bel(x_{t-1})\mathrm{d}x_{t-1}$
$bel(x_t) = \eta p(z_t|x_t)\overline{bel}(x_t)$
}
return $bel(x_t)$

Der Algorithmus besteht aus zwei Teilen. Der erste Teil in Zeile 3 berechnet den geschätzten neuen Zustand unter Miteinbeziehung der Kontrolldaten u_t und der vorherigen Schätzung x_{t-1}. Dazu wird über das Produkt der zwei Verteilungen der Kontrolldaten und der vorherigen Schätzungen integriert [TBF05]. $p(x_t|u_t, x_{t-1})$ ist das sogenannte Bewegungsmodell. Intuitiv ist erkennbar, dass hier der neue Zustand aus dem alten Zustand addiert mit den Kontrolldaten entsteht. Diese

Kontrolldaten sind in der Praxis oft Bewegungsdaten des Roboters. $bel(x_{t-1})$ ist die Schätzung des alten Zustandes. Diese Zeile repräsentiert also die Schätzung des neuen Zustandes mithilfe des alten Zustandes und dem Bewegungsmodell.
Der zweite Teil in Zeile 4 erweitert diese Schätzung unter dem Gesichtspunkt der Messungen, die zu einem gewissen Zeitpunkt t durchgeführt wurden. Dazu wird das zuvor berechnete Ergebnis mit der Wahrscheinlichkeit, dass die Messungen den aktuellen Zustand abbilden, multipliziert. Dies wird dann für jeden möglichen nachfolgenden Zustand x_t gemacht. Da das Ergebnis jedoch keine Wahrscheinlichkeit darstellt, also Werte größer als 1 entstehen können, muss dieses noch durch die Konstante η normalisiert werden.
Der Bayes Filter ist für kleine Schätzprobleme wie z. B. den Zustand einer Tür praktisch gut einsetzbar. Dies ist möglich, da der Zustandsraum endlich ist und das Problem diskret beschrieben werden kann. Dadurch kann das Integral durch eine Summe repräsentiert werden. Der Bayes Filter ist in der praktischen Anwendung von kontinuierlichen Problemen so nicht anwendbar, da das Integral nich berechnet werden kann. Er stellt jedoch ein gutes Konzept dar, welches durch die nächsten Filter umgesetzt wird.

2.1.2 Kalman Filter

Der Kalman Filter gehört zu den Gauss'schen Filtern und war eine der ersten Implementationen des Bayes Filters im kontinuierlichen Raum [TBF05]. Der aktuelle Zustand wird dabei durch eine Gaussverteilung repräsentiert und es besteht die Annahme, dass auch die Messungen gaussverteilt sind [TBF05]. Dabei bewirkt der Mittelwert μ eine Verschiebung der Hochpunkte und eine unterschiedliche Varianz σ^2 eine Stauchung oder Streckung der Kurve. Jeder Filter, der in diese Gruppe gehört, hat die Annahme, dass alle möglichen Zustände durch multivariate Normalverteilungen repräsentiert werden können. Eine Verteilung ist genau dann eine multivariate Normalverteilung, wenn jede Linearkombination der einzelnen Zufallsgrößen jeweils eine univariate Normalverteilung ist. Jede univariat normal verteilte Zufallsgröße X besitzt eine Dichte und wird durch ihren Erwartungswert μ und die Varianz σ^2 charakterisiert.

$$f(x) = (2\pi\sigma)^{-\frac{1}{2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{2.4}$$

Die univariate Normalverteilung ist ein Spezialfall der multivariaten Normalverteilung im Eindimensionalen. Im mehrdimensionalen wird der Erwartungswert μ zu einem Erwartungswertvektor und die Varianz σ^2 wird durch P als Kovarianzmatrix ersetzt. Nun sieht die Dichtefunktion wie folgt aus

$$f(x) = \det(2\pi\mathbf{P})^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x-\mu)^{\mathrm{T}}\mathbf{P}^{-1}(x-\mu)\right) \quad (2.5)$$

Die Dichte repräsentiert somit die Eintrittswahrscheinlichkeit eines Ereignisses durch μ und $\mathbf{P}$, welche die Varianz σ^2 darstellt. Da diese Formel durch den Mittelwert als auch durch die Kovarianz parametrisiert wird, nennt man die Parametrisierung auch Parametrisierung durch Momente. Der Mittelwert und die Kovarianz stellen Momente dar.

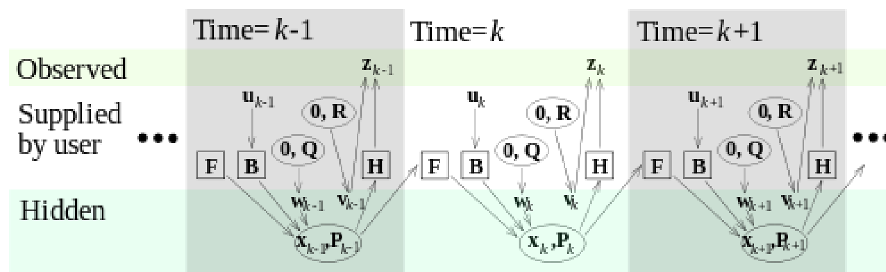

Abbildung 2.1: Funktionsweise des Kalman Filter

Der Kalman Filter arbeitet in zwei Schritten. Im ersten Schritt werden die Vorhersagen über den neuen Zustand mithilfe des Mittelwerts und der Kovarianz angestellt. Die zwei Schritte werden auch „Predict“und „Correct“genannt [WB95]. Dadurch wird die Gaussverteilung repräsentiert, welche die Dichte des geschätzten Zustandes beschreibt. Im zweiten Schritt werden dann die Messdaten, die Kovarianz und die Zustandsprognose aktualisiert. Dazu müssen 3 Bedingungen erfüllt sein.

1. Die Argumente des Zustandsübergangs $p(x_t|u_t, x_{t-1})$ müssen durch eine lineare Funktion zusammenhängen. Diese sieht wie folgt aus:

$$x_t = \mathbf{F}_t x_{t-1} + \mathbf{B}_t u_t + w_t \quad (2.6)$$

Das bedeutet, dass jeder Zustand x_t durch eine lineare Abbildung über die Vektoren u_t und x_{t-1} dargestellt werden kann. u_t ist hierbei der Kontrollvektor zu einer Zeit t und w_t ist zufälliges Gaussrauschen. Dieses Rauschen ist unbekannt und in seinem Mittelwert 0, da es auch wieder als gaussverteilt angenommen wird. $\mathbf{F}_t$ und $\mathbf{B}_t$ sind Matrizen, die mit den Vektoren x und u multipliziert werden.. $\mathbf{F}_t$ ist dabei eine Matrix, welche im konkreten Fall des Roboters das Bewegungsmodell darstellt. Dies bedeutet, dass durch diese Matrix die Änderungen des Zustandes x_t zu dem Zustand x_{t-1} beschrieben wird. Wenn z. B. der Zustandsvektor nur aus der Geschwindigkeit in x, als auch in y Richtung und der Position x und y besteht.

Dann würde bei der Annahme, dass die Geschwindigkeit immer gleich bleibt und sich die Position nur anhand der Geschwindigkeit ändert, da wir von einer linearen Zustandsänderung ausgehen, der Zustandsvektor und die zugehörige Matrix wie folgt aussehen:

$$x_t = \begin{pmatrix} v_x \\ v_y \\ x \\ y \end{pmatrix} \mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & v_x \cdot t & 0 \\ 0 & 0 & 0 & v_y \cdot t \end{pmatrix} \tag{2.7}$$

Es ist einfach zu sehen, dass hier bei Multiplikation nur die Position verändert wird und die Geschwindigkeit gleich bleibt. Dies ist einer der einfachsten Fälle. Meist sind es viel mehr Faktoren, von denen der aktuelle Zustand abhängt. Es könnte z. B. noch die Position im dreidimensionalen Raum und die Orientierung in Winkeln angegeben werden. Je nachdem wie die Bewegung des Roboters dann aussieht, wird die Matrix **A** weitaus komplexer.
Sobald man auf featurebasierten Zuständen arbeiten will, ist der Zustand schnell mit allen möglichen und erkannten Features überladen. Ein Feature ist ein Merkmal, welches von einem Sensor erkannt wurde und zur Repräsentation des Zustandes dient. Des weiteren entsteht dabei das Problem, dass Features auch während der Laufzeit hinzugefügt werden können, und sich dadurch dann auch die Matrix während der Laufzeit ändern muss.
Dieselbe Strategie wird auch bei der Matrix **B** angewandt. Diese beschreibt, wie sich die Kontrolldaten, im konkreten Fall eines Roboters die Messdaten des Sensors, verhalten. Bei einem einfachen Laserscanner ist dies weniger ein Problem, da dort nur Winkel und Abstand bekannt sind, jedoch kommen bei kamerabasierten Messdaten Transformationen hinzu, welche die Kameradaten auf die Welt projizieren, usw. Hier erkennt man eine Schwäche des Kalman Filters durch zu viele Informationen, die berücksichtigt werden müssen.
Durch Einsetzen von (2.6) in (2.5) erhält man:

$$\begin{aligned} p(x_t|u_t, x_{t-1}) = & \det(2\pi\mathbf{R}_t)^{-\frac{1}{2}} \\ & \exp\left(-\frac{1}{2}(x_t - \mathbf{F}_t x_{t-1} - \mathbf{B}_t u_t)^{\mathrm{T}} \mathbf{R}_t^{-1} (x_t - \mathbf{F}_t x_{t-1} - \mathbf{B}_t u_t)\right) \end{aligned} \tag{2.8}$$

Hier ist der Mittelwert durch $\mathbf{F}_t x_{t-1} + \mathbf{B}_t u_t$ und die Kovarianz durch $\mathbf{R}_t$ dargestellt.
2. Die Eintrittswahrscheinlichkeit der Messungen $p(z_t|x_t)$ muss auch durch eine lineare Funktion, also durch eine Gaussverteilung repräsentierbar sein. Hier wird das Gaussrauschen

$$\mathbf{H}_t x_t + \sigma_t \tag{2.9}$$

hinzugefügt. $\mathbf{H}_t$ ist eine Matrix und σ ist das Rauschen der Messungen. Dabei müssen die Argumente auch wieder eine lineare Abhängigkeit haben. Das heißt, die Messung ist linear abhängig von einem Rauschen und der Matrix, welche bestimmt, wie sich die Messungen verhalten. Die Messungen sind auch wieder multivariat normal verteilt und die Eintrittswahrscheinlichkeit $p(z_t|x_t)$ sieht wie folgt aus

$$p(z_t|x_t) = \det(2\pi\mathbf{Q}_t)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(z_t - \mathbf{H}_t x_t)^{\mathrm{T}} \mathbf{Q}_t^{-1}(z_t - \mathbf{H}_t x_t)\right) \tag{2.10}$$

3. Die erste Prognose muss normal verteilt sein. Dazu werden der Mittelwert in μ_0 und die Kovarianz in P_0 umgenannt.

$$p(x_0) = \det(2\pi\mathbf{P}_0)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x_0 - \mu_0)^{\mathrm{T}} \mathbf{P}_0^{-1}(x_0 - \mu_0)\right) \tag{2.11}$$

Diese drei Bedingungen stellen sicher, dass die Annahmen zu jedem Zeitpunkt t in jedem Punkt auch wieder gaussverteilt sind. Der Beweis kann in [TBF05] nachgelesen werden.

Algorithm 2 Algorithmus des Kalmanfilter

Input: μ_{t-1} – Mittelwert der letzten Schätzung
Input: $\mathbf{P}_{t-1}$ – Kovarianz der letzten Schätzung
Input: u_t – Kontrolldaten
Input: z_t – Messdaten
Output: $\mu_t, \mathbf{P}_t$ – neue Zustandsschätzung

$\overline{\mu_\mathbf{t}} = \mathbf{F}_t \mu_{\mathbf{t-1}} + \mathbf{B}_t u_t$
$\overline{\mathbf{P_t}} = \mathbf{F}_t \mathbf{P_{t-1}} \mathbf{F}_t^{\mathrm{T}} + \mathbf{R}_t$
$K_t = \overline{\mathbf{P_t}} H_t^{\mathrm{T}} (\mathbf{H}_t \overline{\mathbf{P_t}} \mathbf{H}_t^{\mathrm{T}} + \mathbf{Q}_t)^{-1}$
$\mu_\mathbf{t} = \overline{\mu_\mathbf{t}} + K_t(z_t - \mathbf{H}_t \overline{\mu_\mathbf{t}})$
$\mathbf{P_t} = (I - K_t \mathbf{H}_t) \overline{\mathbf{P_t}}$
return $\mu_t, \mathbf{P}_t$

Der erste Teil des Algorithmus von Zeile 1 - 2 beschreibt die Vorhersage des Filters. Hier werden der neue Mittelwert, also der neue geschätzte Zustand und die zugehörige Kovarianz berechnet. Dies geschieht anhand der Kontrolldaten und des Zustandes des vorherigen Zeitpunktes, noch unabhängig von den Messdaten

z_t. Die neue Kovarianzmatrix kann wie in Zeile 2 berechnet werden, da davon ausgegangen wird, dass jeder Zustand auf einen vorherigen Zustand basiert.
In den darauffolgenden Zeilen werden dann die errechneten Werte abhängig von den Messdaten aktualisiert. In Zeile 3 wird der sogenannte Kalman Gain berechnet. Dieser bestimmt zu welchem Grad die Messungen die Zustandsänderung beeinflussen. Wie diese Formel zustande kommt kann in [TBF05] nachgelesen werden. In Zeile 4 wird dann der geschätzte Zustand mit den aktuellen Messungen erweitert. Dabei gibt der Kalman Gain den Grad der Änderung an. Dieser wird mit der Differenz zwischen der aktuellen Messung z_t und der erwarteten Messung $\mathbf{H}_t\overline{\mu_t}$ multipliziert. Durch die pure Schätzung des Zustandes ohne das Einbeziehen der Messungen würde die Unsicherheit immer weiter wachsen. Durch den Korrekturschritt ab Zeile 3 wird der Zustand in Richtung der Messungen „gezogen“, wodurch die Unsicherheit sinkt. Anschließend wird dann die neue Kovarianz abhängig von dem Kalman Gain berechnet.
Der Kalman Filter wurde entwickelt, um Prognosen über einen Zustand in einem linearen Gauss-System zu bestimmen. Dabei arbeitet er sehr effizient bei einem Aufwand von annähernd O($k^{2.4}$), wobei k die Dimension des Messdatenvektors ist [TBF05]. Er ist jedoch nicht für nichtlineare Probleme geeignet, das heißt, falls sich z. B. das Bewegungsmodell nichtlinear verhalten sollte. Auch zu große Zustände können den Filter schnell langsamer werden lassen.

2.1.3 Extended Kalman Filter

Der erweiterte Kalman Filter hingegen wurde für den Fall der Nichtlinearität konzipiert. Er ist so modifiziert, dass die Messungen und die Zustandsübergänge auch im nichtlinearen Fall normal verteilt sind. Hier fällt die Bedingung der Linearität weg, wodurch der Algorithmus ein größeres Einsatzgebiet erreicht. Dazu werden die im Kalman Filter verwendeten Matrizen A, B und H durch die nichtlinearen Funktionen

$$x_t = g(u_t, x_{t-1}) + w_t \tag{2.12}$$

$$z_t = h(x_t) + \delta_t \tag{2.13}$$

ersetzt. Hierbei werden die Matrix $\mathbf{A}$ und $\mathbf{B}$ durch g ersetzt und die Matrix $\mathbf{H}$ durch die Funktion h. Dadurch ist die Bedingung der Linearität aufgehoben, wodurch die Annahme nicht mehr gauss'sch sein muss. Der EKF(Extended Kalman Filter) approximiert die Funktionen durch Mittelwert und Kovarianz zu einem Zeitpunkt t, um wieder eine lineare Abbildung und somit eine geschätzte Gaussverteilung zu erreichen. Dadurch verlagert sich das Problem, eine exakte Lösung zu finden, dahin, eine gute Annäherung von Mittelwert und Kovarianz zu finden.
Es wird versucht, die nichtlinearen Funktionen g und h durch lineare Funktionen

anzunähern. In diesem Fall wird dazu die Taylor Entwicklung verwendet.
Der Vorteil liegt hier wieder in der Effizienz. Das Monte Carlo Verfahren [DGA00] könnte die Annäherung besser bestimmen, jedoch ist das Verfahren langsamer und nicht so adaptiv, wie der EKF. Der Vorteil der Linearisierung liegt darin, dass nach der Bestimmung der linearen Funktionen mit dem Kalman Verfahren fortgesetzt werden kann. Der EKF versucht also, die nichtlineare Darstellung eines Zustandes durch lineare Funktionen darzustellen, um wieder eine Gaussverteilung zu erhalten und dann den normalen Kalman Filter mit dieser Verteilung anzuwenden.
Durch die verwendetet Taylor Entwicklung entstehen die Matrizen $\mathbf{H}_t$ und $\mathbf{G}_t$, welche jeweils die Funktionen g und h zu einem Zeitpunkt t repräsentieren [TBF05].

Algorithm 3 Algorithmus des Extended Kalmanfilter

Input: μ_{t-1} – Mittelwert der letzten Schätzung
Input: $\mathbf{P}_{t-1}$ – Kovarianz der letzten Schätzung
Input: u_t – Kontrolldaten
Input: z_t – Messdaten
Output: $\mu_t, \mathbf{P}_t$ –neue Zustandsschätzung

$\overline{\mu_\mathbf{t}} = g(\mu_{\mathbf{t-1}}, u_t)$
$\overline{\mathbf{P_t}} = \mathbf{G}_t \mathbf{P_{t-1}} \mathbf{G}_t^{\mathrm{T}} + \mathbf{R}_t$
$K_t = \overline{\mathbf{P_t}} \mathbf{H}_t^{\mathrm{T}} (\mathbf{H}_t \overline{\mathbf{P_t}} \mathbf{H}_t^{\mathrm{T}} + \mathbf{Q}_t)^{-1}$
$\mu_\mathbf{t} = \overline{\mu_\mathbf{t}} + K_t(z_t - h(\overline{\mu_\mathbf{t}}))$
$\mathbf{P_t} = (I - K_t \mathbf{H}_t)\overline{\mathbf{P_t}}$
return $\mu_t, \mathbf{P}_t$

Hier ist zu sehen, dass sich der Algorithmus des EKF nicht viel von dem des KF unterscheidet. Der Unterschied besteht darin, dass die linearen Teile des Algorithmus durch die nichtlineare Generalisierungen des EKFs ersetzt werden.

Ein Vorteil des EKF ist die Einfachheit und schnelle Berechnungszeit. Sein Aufwand liegt in $O(k^{2.4} + n^2)$, wobei k die Dimension des Messvektors z_t und n die Dimension des Zustandsvektors x_t ist. Ein Nachteil des EKF jedoch ist es, dass die die Funktionen g und h lediglich durch eine Taylor Entwicklung approximiert werden, also nie die genaue Funktion abbilden können. Es entsteht eine ungenauere approximierte Gaussverteilung und somit ungenauere Schätzungen des Mittelwertes und der Kovarianz. Die Schätzung des Zustandes ist also ungenauer. Daher ist es wichtig, bei Benutzung des EKF den Fehler der Zustandsschätzung möglichst gering zu halten, da sonst die Unsicherheit des Zustandes immer weiter anwächst und die Zustandsänderung, sowie die Messungen nichtlinearer werden.
Zusammengefasst kann gesagt werden, dass mit dem Kalmanfilter die beste Lösung von linearen Schätzproblemen zu erwarten ist. Der EKF erweitert dies um nicht-

lineare Modelle, nähert sie aber nur durch die Taylorentwicklung an. Er ist dabei jedoch sehr schnell und einfach zu implementieren.

2.1.4 Partikelfilter

Der Partikelfilter ist auch unter dem Namen Sequentielle Monte-Carlo-Methode bekannt [DGA00, Mü07]. Er stellt eine andere Art der Umsetzung des Bayes Filters dar und gehört zur Obergruppe der nicht parametrisierten Filtern. Diese zeichnen sich dadurch aus, dass sie keine feste funktionale Form der Wahrscheinlichkeitsdichtefunktion besitzen. Stattdessen wird der nachfolgende Zustand mit einer endlichen Anzahl an Samples oder Regionen angenähert. Durch die Anzahl wird die Exaktheit des Filters bestimmt, aber auch der benötigte Aufwand erhöht. Jedes Sample repräsentiert dabei einen möglichen Zustand. Es wird dann der wahrscheinlichste nachfolgende Zustand gewählt und z. B. mit Kontrolldaten abgeglichen. Das beste Ergebnis wird weiter verwendet. Die so den nachfolgenden Zustand repräsentierenden Samples nennt man auch Partikel.
Der Vorteil des Partikelfilters ist, dass er mehr Verteilungen der Wahrscheinlichkeiten als nur Gaussverteilungen repräsentieren und auch nichtlineare Verteilungen modellieren kann. Konkret wird ein Zustand $\mathbf{X}_t$ wie folgt dargestellt

$$\mathbf{X}_t = x_t^{[1]}, x_t^{[2]}, x_t^{[3]}, .., x_t^{[M]} \tag{2.14}$$

Jedes $x_t^{[m]}$, mit $1 \leq m \leq M$ entspricht dabei einem Partikel, welches einen konkreten Zustand zur Zeit t darstellt. M ist dabei die Anzahl der verwendeten Partikel. Die ersten Samples entstehen dabei, indem unter dem Einfluss der Kontrolldaten u_t zufällig in die Umgebung gestreut wird. Danach wird eine Gewichtung $w_t^{[m]}$ zu jedem Partikel $x_t^{[m]}$ erzeugt, welche die Wahrscheinlichkeit des geschätzten Zustandes unter Berücksichtigung von Messdaten z_t bestimmt. Das Partikelset $\overline{X}_t$ um einen Partikel mit den Werten des neu geschätzten Zustandes $x_t^{[m]}$ und der dazugehörigen Gewichtung $w_t^{[m]}$ erweitert. Bei dem sogenannten Resampling oder Importance Sampling werden anhand der Gewichtung die Samples dem Partikelset $\mathbf{X}_t$ hinzugefügt und diese Partikel wieder über den neuen Zustand gelegt. Dadurch kommt es vor, dass viele Partikel doppelt in einem neuen Set erscheinen und Partikel mit geringer Gewichtung nicht mehr vorkommen. Es verlagert sich die Vermutung über den neuen Zustand zu den Partikeln mit der höchsten Trefferwahrscheinlichkeit. Wie auch bei dem Bayes Filter werden die neuen Zustände abhängig von den alten Zuständen rekursiv bestimmt. Im Folgenden wird eine mögliche Umsetzung des Algorithmus dargestellt. Dies ist der Grundalgorithmus und es gibt noch viele Optimierungsmöglichkeiten, welche später vorgestellt werden.

Algorithm 4 Algorithmus des Partikelfilter

Input: $\mathbf{X}_{t-1}$ – Menge aller Partikel
Input: u_{t-1} – Kontrolldaten
Input: z_t – Messdaten
Output: $\mathbf{X}_t$ – neue Zustandsschätzung

$\overline{\mathbf{X}}_t = \mathbf{X}_t = \emptyset$
for $(m = 1 \rightarrow M)$
{
  sample $x_t^{[m]}$ $p(x_t|u_t, \mathbf{X}_{t-1}^{[m]})$
  $w_t^{[m]} = p(z_t|x_t^{[m]}$
  $\overline{\mathbf{X}}_t = \overline{\mathbf{X}}_t + (x_t^{[m]}, w_t^{[m]})$
}
for $(m = 1 \rightarrow M)$
{
  draw $\overline{\mathbf{X}}_t^{[m]}$ with probability $\propto w_t^{[m]}$
  add $x_t^{[m]}$ to $\mathbf{X_t}$
}
return $\mathbf{X_t}$

Dieser Code entstammt Dem Buch Probablistic Robtics [TBF05] und wurde leicht von mir verändert, um die einzelnen Vorgehensweisen plausibler hervorzuheben.
Der Algorithmus besteht aus drei Teilen. Der erste Teil (Zeile 4) erzeugt die neuen Samples. der zweite Teil (Zeile 5) erzeugt die Gewichtung zu jedem Sample und der letzte Teil (Zeile 9 - 10) beschreibt das Resampling.
Im ersten Teil werden die Samples proportional zu der Wahrscheinlichkeit $p(x_t|u_t, x_{t-1}^{[m]})$ erzeugt. Das erzeugte Sample wird durch m indiziert, damit es eindeutig dem m-ten Partikel aus X_{t-1} zugeordnet wird.
Im zweiten Teil werden die Gewichte zu den jeweiligen Samples erzeugt. Diese werden auch Importance Faktor genannt und lassen die Messungen mit in die Samples einfließen. Dies ist charakteristisch für Umsetzungen des Bayes Filters. Wichtig ist, dass die Wahrscheinlichkeit $p(z_t|x_t^{[m]})$ genommen wird, also die Wahrscheinlichkeit, dass eine Messung den aktuellen Zustand abbildet. Danach werden die Partikel dem neuen Set $\overline{X}_t$ hinzugefügt. Die nach der Schleife erzeugten Samples repräsentieren $\overline{bel}(x_t)$ und das Set an Gewichten beschreibt $bel(x_t)$ aus dem Bayes Filter [TBF05].
Im dritten Teil wird dann das sogenannte Resampling oder Importance Sampling durchgeführt. Hier werden M Partikel anhand ihrer Gewichtung gewählt und diese

wieder auf den neuen Zustand gelegt. Dabei wird das neue Partikelset X_t mithilfe der Gewichte $w_t^{[m]}$ gebildet. Vor dem Resampling waren die Partikel wie in $\overline{bel}(x_t)$ verteilt. Nach dem Resampling ist die Verteilung $bel(x_t) = \eta p(z_t|x_t^{[m]})\overline{bel}(x_t)$. Dabei entstehen, wie oben beschrieben, viele Duplikate, da „gute"Partikel übernommen und in das neue Set X_t eingefügt werden. Dadurch werden die guten Samples bevorzugt und durch das Resampling werden nur Regionen untersucht, in denen es wahrscheinlich ist, den aktuellen Zustand vorzufinden.
Dabei kann es jedoch in Extremfällen vorkommen, dass nie der richtige Zustand getroffen wird. Dadurch entstehen Fehler in der Zustandsschätzung. Diese Fehler können verringert werden, indem nach der Resample-Phase zufällig neue Samples gestreut werden und mit diesen dann weiter gearbeitet wird. Das deckt jedoch nicht komplett den Fall ab, da der Partikelfilter bloß eine Annäherung an den aktuellen Zustand bildet. Doch in der Praxis kommt dies bei gut gewählter Anzahl an Partikeln so gut wie nie vor [TBF05]. Das bedeutet, dass bei ausreichend hoch gewählter Anzahl an Partikeln schon gute Ergebnisse erzielt werden und je höher die Partikelanzahl gewählt wird, desto höher ist die Genauigkeit der Approximation.
Einer der Vorteile von Partikelfiltern ist, dass er auch Zustände abbilden kann, die nicht gaussverteilt sind. Das Verfahren ist auch nicht so fehleranfällig wie der Kalmanfilter, da falsch angenommene Zustände bei ausreichend häufigem Resampling und ausreichend hoch gewählter Partikelanzahl die falschen Zustände herausgefiltert werden. Der Kalmanfilter kann im Prinzip wie ein Ein-Partikel-Partikelfilter angesehen werden, da er immer von dem erste angenommenen Zustand ausgeht und diesen weiter verfolgt. Jedoch ist der Rechenaufwand des Partikelfilters höher, da viele Hypothesen verfolgt werden, statt nur einer. Also kann gesagt werden, dass die Genauigkeit des Partikelfilters mit höherem Rechenaufwand steigt.

2.2 SLAM

Während sich ein Roboter in einem Gebiet bewegt, muss er immer seine aktuelle Position in diesem Gebiet kennen. Dies ist wichtig, um Aufgaben erfüllen zu können. Ist die Karte gegeben, erweist sich dieses Problem als einfach, da die Scans der Sensoren nur mit der Karte abgeglichen werden müssen. Ist die Karte jedoch nicht bekannt, also das Gebiet in dem der Roboter sich bewegt, sind komplexere Verfahren nötig. Dieser Fall tritt häufiger auf, da sich selten vorher die Mühe gemacht wird, eine Karte von jeder Umgebung zu erstellen, in der sich ein Roboter bewegen soll. Dieses gleichzeitige Erstellen einer Karte und auch die Selbstlokalisierung in dieser, bedeutet SLAM (Simultaneuous Localization and Mapping).

Das größte Problem an SLAM ist eben dieses, da beides voneinander abhängig ist. Dies ist einfach zu verstehen, wenn man sich über die Abhängigkeiten der bei-

den Komponenten klar wird. Ein Roboter kann nur dann wissen, wo er sich auf einer Karte befindet, wenn er diese Karte kennt. Eine Karte kann aber auch nur dann exakt erstellt werden, wenn bekannt ist, wo die Messungen gemacht wurden, die diese Karte bilden. Es gibt verschiedene Ansätze um dieses Problem zu lösen. GMapping verwendet dafür einen Rao-Blackwellized Partikelfilter [GSB05]. Allerdings benötigt explizit GMapping eine anfängliche Schätzung, was im Bereich der Robotik die Odometrie darstellt. Zu Testzwecken wurde hier ein geeigneter Scanmatcher zur Simulation der Odometrie verwendet. in Scanmatcher sucht die beste Transformation, um zwei verschiedene Scans so gut es geht aufeinander abzubilden. Hector SLAM [KMSK11] hingegen verwendet ein Gauss-Newton Verfahren für das Scanmatching und einen Erweiterter Kalman Filter zur Schätzung der Pose. Welche Vor- und Nachteile beides hat und wie genau die Verfahren funktionieren wird später im praktischen Teil erläutert.

Es gibt verschiedene Herangehensweisen bzw. Umsetzungsmöglichkeiten der theoretischen Filter in die Praxis, denn in der Theorie funktionieren die Filter im kontinuierlichen Raum, was aber in Berechnungen entweder sehr langsam oder auch gar nicht umsetzbar ist. Eines der ältesten und bewährtesten Verfahren dabei ist wohl FastSLAM. Dieses arbeitet in seiner Grundform auf einem featurebasierten System, sucht also nach Auffälligkeiten in seiner Umgebung und merkt sich diese, um sich anhand derer zu orientieren und die aktuelle Position zu bestimmen. Es gibt noch eine Weiterentwicklung des FastSLAM, welche unter FastSlam 2.0 bekannt ist. Diese Variante ist von den Features weg zu einem Ansatz mit Partikelfilter und EKF gekommen. Dazu gibt es noch SLAM-Verfahren, die nur auf dem EKF beruhen und auch welche, die nur mit Partikelfiltern arbeiten. Im folgenden werden diese Verfahren grob vorgestellt und Vor- bzw. Nachteile vorgestellt.

2.2.1 SLAM mit Kalman Filtern

SLAM mit dem EKF Filter zu lösen ist eine der ältesten Möglichkeiten der Problemlösung. Das Verfahren arbeitet feature-basiert auf Grundlage des EKF. Das bedeutet, es wählt aus der Umgebung, die von einem Sensor gemessen wurde, jedes Feature oder jede Landmarke aus, die erkannt wird und aktualisiert damit den prognostizierten Zustand des Roboters. Dazu wird dem schon bekannten Algorithmus des EKF ein Initialisationsschritt für jede Landmarke hinzugefügt, die neu erkannt wird. Dabei wird mithilfe der Pose und der Messdaten der aktuelle Zustand der Landmarke bestimmt. Der Zustandsvektor und die Kovarianzmatrix werden dann um diese Zustände erweitert. Die aktuelle Karte der Umgebung ist hierbei ein Zustandsvektor über den Roboterzustand R und alle erkannten Landmarken L.

$$x_t = \begin{bmatrix} R \\ M \end{bmatrix} = \begin{bmatrix} R \\ L_1 \\ \vdots \\ L_n \end{bmatrix} \tag{2.15}$$

Im EKF wird die Karte über eine Gaussverteilung dargestellt, welche über Mittelwert $\overline{\mu}$ und Kovarianz P des Zustandsvektors bestimmt ist. Dabei ist der Mittelwert $\overline{\mu}$ ein Vektor aus den Mittelwerten des Roboterzustandes und aller Landmarken. Der Mittelwert und die Kovarianzmatrix, welche den aktuell geschätzten Zustand repräsentieren, können wie folgt aussehen

$$\mu = \begin{bmatrix} \mathbf{R} \\ \mathbf{M} \end{bmatrix} \tag{2.16}$$

$$\mathbf{P} = \begin{bmatrix} \mathbf{P_{RR}} & \mathbf{P_{RM}} \\ \mathbf{P_{MR}} & \mathbf{P_{MM}} \end{bmatrix} \tag{2.17}$$

Dabei ist **M** die Menge aller Landmarken, **P** die Kovarianzmatrix und **R** der Roboterzustand.. Das Ziel ist es, die so repräsentierte Karte immer aktuell zu halten. Der Algorithmus sieht dann in etwa wie folgt aus:

1. Der neue Zustand vorhergesagt.
2. Die Messung wird vorhergesagt und damit auch die Landmarken
3. Die Messdaten werden mit dem Zustand assoziiert.
4. Update.

Dazu müssen einige Veränderungen in dem ursprünglichen Algorithmus gemacht werden. Die erste Änderung waren der Mittelwert und die Kovarianz unter Berücksichtigung aller Landmarken. Die Funktion g aus dem Algorithmus des EKF beeinflusst nur die Bewegung des Roboters und nicht die Landmarken, da diese als statische Objekte in der Welt angesehen werden. Damit ändert sich an der ersten Zeile des Algorithmus von der Theorie her nichts. Die Funktion g muss um eine Matrix erweitert werden, hier **F** genannt, welche nur den Zustand des Roboters verändert. Diese Matrix hat eine Größe, die abhängig von allen Roboterzuständen und den Landmarken ist. Die Kovarianzmatrix muss dann durch selbige Matrix F an den Zustandsvektor angepasst werden. Die genaue Herleitung als auch die konkrete Umsetzung können in [TBF05] nachgelesen werden. Nun muss der Kalman

Gain oder auch der Korrekturschritt verändert werden. Dazu muss beachtet werde, dass es zwei Fälle gibt. Der erste Fall, dass eine erkannte Landmarke schon im Zustandsvektor vorhanden ist und der andere, dass diese noch hinzugefügt werden muss. Muss die Landmarke noch hinzugefügt werden, sieht die Idee wie folgt aus:

$$\overline{\mu_j} = \overline{\mu_t} + \overline{z}_t^i \tag{2.18}$$

Die Position einer Landmarke mit dem Index j wird durch den Mittelwert μ_j beschrieben. $\overline{z}_t^i$ ist die i-te Messung unter der die Landmarke erkannt wurde. Diese beschreibt die relative Messung zur Position des Roboters. Die Position der Landmarke entsteht aus der geschätzten Position des Roboters und der relativen Messung des Sensors. Falls die Landmarke schon bekannt war, kann dieser Teil übersprungen werden und es wird direkt die erwartete Messung berechnet. Die erwartete Messung wird anhand der geschätzten Position des Roboters und der Landmarke berechnet. Dabei wird aus dem Vektor, der die Differenz zwischen den beiden Positionen beschreibt, und dem Winkel, unter dem die Landmarke erkannt wurde die geschätzte Messung bestimmt. Dann wird die Jacobi Matrix

$$\mathbf{H}_t^i = \frac{\partial h(\overline{\mu}_t)}{\partial \overline{\mu}_t} \tag{2.19}$$

gebildet, welche mithilfe einer Matrix $\mathbf{F}_j$ in den höher dimensionalen Raum transformiert und dann anstelle der aus dem EKF bekannten Matrix H verwendet wird. $\mathbf{F}_j$ ist dabei ähnlich zu der Matrix $\mathbf{F}$ aufgebaut. j entspricht hier dem Index der Messung. Jetzt muss noch das Rauschen $\mathbf{Q}_t$ durch eine Matrix mit allen Sensorparametern erstellt werden. Mithilfe dieser Änderungen können die weiteren Schritte des ursprünglichen EKF weiter verwendet werden. Der komplette Algorithmus und die genauen Beweise können in [TBF05] nachgelesen werden.
Mit bekannten Landmarken ist das Verfahren recht schnell, werden jedoch während der Laufzeit Landmarken hinzugefügt und damit eine Aktualisierung nötig, steigt der Aufwand quadratische zur Anzahl der Landmarken [TBF05]. Das Verfahren ist je nach Einsatzgebiet bei einem kompletten SLAM Problem nicht anwendbar, da mit jedem Zeitschritt der Zustandsvektor und auch die Kovarianzmatrix wächst und der Algorithmus damit immer langsamer wird. Jedes neue Feature was erkannt wird, wird in dem Zustand gespeichert. Bei kleineren Karten ist ein solches SLAM Verfahren noch denkbar, jedoch wird das Verfahren bei Karten unbekannter Größe leicht inkonsistent. Daher sind SLAM Verfahren nur mit dem EKF Filter nicht mehr aktuell, aber der EKF wird für Teilprobleme in einigen SLAM Lösungen verwendet, wie später noch zu sehen ist.

2.2.2 SLAM mit Partikelfiltern

Partikelfilter als SLAM Verfahren wurden z. B. in [FTBD01] implementiert. Es handelt sich um ein einfaches, aber doch effizientes Verfahren. Der Vorteil ist, dass der theoretische Algorithmus, welcher weiter oben beschrieben wurde, fast komplett übernommen werden kann. Falls anfangs keine Karte gegeben ist, können die ersten Sensordaten als Karte verwendet werden. Das Ergebnis des Partikelfilters ist dann die naheliegendste Pose des Roboters. Die zugehörigen Sensordaten werden dann anhand der Pose für das Aktualisieren der Karte benutzt. Der Standard Partikelfilter wird also nur um ein Aktualisieren der Karte erweitert. Jedoch gibt es einige Verbesserungsansätze, z. B. ICP [RL01] zur verbesserten Poseschätzung zu benutzen, mit welcher dann der Partikelfilter arbeitet. Dies wurde beim RoboCup 2009 - RoboCup Rescue vom Team resko@UniKoblenz umgesetzt. Dazu wird die Odometrie des Roboters verwendet und lokales Scanmatching durchgeführt. Weitere Möglichkeiten den partikelfilter für SLAM zu verwenden kommt in den folgenden Kapiteln mit FastSLAM 1 und FastSLAM 2.

2.2.3 FastSLAM 1

Eine der ersten Implementierungen eines effektiven SLAM-Verfahrens kann unter dem Namen FastSLAM gefunden werden [HBFT03]. FastSLAM verwendet sowohl den EKF als auch den Partikelfilter. Als erstes werden mit dem EKF Landmarken gespeichert und ihre genaue Position geschätzt. Damit wird die globale Karte repräsentiert. Mithilfe des Partikelfilters wird dann die Roboterpose geschätzt. Dazu stehen in jedem Partikel eine mögliche Karte und die Position des Roboters. Es können mehrere Karten verfolgt und die wahrscheinlichste später herausgefiltert werden. Dies nutzt den Vorteil beider Filter aus. Dabei arbeitet das Verfahren auf einem Occupancy Grid, welches in jeder Zelle die Wahrscheinlichkeiten einer Belegtheit hat. Der Vorteil von FastSLAM ist, dass nicht nur über verschiedene Positionen gestreut wird, sondern auch über verschiedene Karten. Dies beugt dem Fehler vor, der durch falsche Positionsschätzungen entsteht, vor. Bei K Landmarken und M Partikeln würde der Aufwand von FastSLAM $O(KM)$ entsprechen, aber durch einige Verbesserungen, wie eine kd-tree basierte Datenstruktur, lässt sich der Aufwand auf $O(M \log K)$ minimieren [MTKW02].
Der Algorithmus besteht dabei aus fünf Schritten. Zuerst wird eine Positionsvorhersage unter der Bedingung der Odometrie angestellt. Diese Schätzung wird noch mit einer Varianz verrauscht. Dies dient dazu, damit in dem so entstandenen Bereich Partikel gestreut werden können. Diese Streuung ist multivariat normal verteilt und besitzt den Mittelwert μ_t und die Kovarianz P_t. Dann folgt das Wiedererkennen der erkannten Landmarken, wobei sie der besten Messung z_i zugeordnet werden. Die Landmarken werden mit den Messungen über den Kalmanfilter aktua-

lisiert und neue Messungen werden als neue Landmarken hinzugefügt. Die Partikel werden aufgrund ihrer Gewichtungen neu verteilt, wie es im Partikelfilter üblich ist. Dabei erzeugen Partikel mit hohen Wahrscheinlichkeiten viel neue Partikel und Partikel mit geringeren Wahrscheinlichkeiten nur wenige bis keine Partikel. Jedes Partikel besitzt wieder eine neue Karte der Umgebung.
Die Positionsschätzung und das Rauschen werden über das Aufaddieren der Odometrie auf die Position jedes alten Partikels bestimmt. Das Rauschen kann aus Einfachheit als konstant angenommen werden, welches unabhängig von der Bewegung ist [TBF05]. Das Rauschen wird über die Kovarianzmatrix P_t repräsentiert. Eine Schwierigkeit besteht in dem Wiedererkennen der Landmarken. Oft kann die eindeutige Zuordnung nicht mehr stattfinden. Dazu kann es hilfreich sein, Landmarken mit großem Abstand zueinander zu wählen, aber dadurch kann die falsche Datenassoziation nicht verhindert werden. Auch Doppelassoziationen von Landmarken müssen beachtet werden. Dazu muss eine Minimalwahrscheinlichkeit festgelegt werden, bei der ein Feature als neues Feature erkannt wird. Sind alle Landmarken aus der aktuellen Messung bekannt, werden bereits bekannte Positionen der Landmarken durch den Kalmanfilter genauer berechnet. Dies wird bei jedem Erkennen einer Landmarke genauer. Dadurch wird die aktuelle Karte eines Partikels aktualisiert.
Im letztem Schritt, dem Resampling, wird die Positionsschätzung mittels Partikelfilter verfeinert. Dazu werden die Wahrscheinlichkeiten aller erkannten Landmarken miteinander addiert. Neue Landmarken bekommen dabei die Mindestwahrscheinlichkeit. Diese Gewichte werden dann zu jedem Partikel normalisiert, sodass die Summe der Gewichte aller Partikel 1 ergibt. Neue Partikel werden dann proportional zu den Gewichten neu erzeugt

$$p(k)\text{count}(\text{particles}) \tag{2.20}$$

Formel 2.21 ist dabei eine Möglichkeit der Verteilung. FastSLAM ist dabei eine sehr schnelle Implementierung. Dadurch, dass für jedes Partikel und jede Landmarke die Wahrscheinlichkeiten mit dem Kalmanfilter berechnet werden müssten, würde der Aufwand sehr hoch werden. Dies wird aber durch vorheriges Vorfiltern minimiert, indem die Wahrscheinlichkeit des nahesten Feature berechnet wird [MTKW02]. FastSLAM geht auch davon aus, dass Landmarken voneinander unabhängig sind, wenn die Roboter Pose gegeben ist. Dadurch minimiert sich der Aufwand, da keine große Kovarianzmatrix mehr verwendet wird, sondern für jede Landmarke eine einzelne.

2.2.4 FastSLAM 2.0

FastSlam 2.0 ist eine verbesserte Variante des FastSLAM 1. Ein Unterschied liegt darin, schon vor dem Sampling eine Vorauswahl getroffen wird, wo der Partikelfilter seine Partikel streuen soll. Während FastSLAM 1 in einem groben Bereich um die Kovarianz der geschätzten Position streuen würde, streut FastSLAM 2.0 nur im Bereich der Kovarianz. Dies geschieht durch einen weiteren Kalmanfilter. Es kann also gesagt werden, dass die Positionsabschätzung durch Kalmanfilter und Partikelfilter zustande kommt [MTKW03]. Der Algorithmus wird dadurch leicht verändert.
Im ersten Schritt wird immer noch die Positionsabschätzung durch Odometriedaten und einem Rauschen ermittelt. Dann wird jedoch erst der Messung z_t die Landmarken zugeordnet, was dem dritten Schritt in FastSLAM entspricht. Dann erfolgt die Positionsschätzung des Roboters x_t mithilfe eines Kalmanfilters. Dabei wird auch ein mögliches Rauschen mitberechnet, welches durch die Kovarianzmatrix dargestellt wird. Jetzt werden um den errechneten Mittelpunkt $\hat{x}_t$ und der Kovarianz $\hat{\Sigma}_t$ die Partikel gestreut. Da es eine mögliche Veränderung der geschätzten Position gibt, müssen auch die Landmarken teilweise neu den Messungen zugeordnet werden. Es wird also der zweite Schritt wiederholt. Jetzt kann wieder die Systematik von FastSlam 1 angewendet werden und es findet das Aktualisieren der Karte und das Resampling statt, wie es in FastSLAM 1 beschrieben wurde [Men07].
FastSlam 2.0 hat also den Unterschied, dass die Messungen schon bereits vor der Streuung mit berücksichtigt werden. Dadurch ergibt sich eine bessere Schätzung und es müssen weniger Partikel gestreut werden. Es entstehen auch weniger falsch angenommene Karten und der Fehler dieser ist auch nicht so hoch wie in FastSlam 1 [Men07].

2.3 Exploration

Die Exploration beschreibt das autonome Erkunden einer Umgebung durch den Roboter. Dies ist besonders für autonome Roboter wichtig, die in unbekannten Gebieten arbeiten müssen. Roboter die z.B. in verschütteten Gebieten nach Lebenszeichen suchen müssen, von denen keine Karte existiert. Auch macht es Roboter unabhängiger von der Umgebung in denen sie arbeiten, da sie nicht mehr auf ihre bekannten gebiete reduziert sind. Es gibt schon viele Anwendungsgebiete und wird in Zukunft wohl auch noch mehr geben, in denen autonome Roboter ihren Einsatz finden. SLAM allein kann zwar berechnen, wo sich der Roboter befindet und wie die Karte aussieht, jedoch kann es nicht bestimmen, wo der Roboter als nächstes hingehen muss und wo er nicht entlang gehen kann. Diese Navigation übernimmt dann die Exploration. Dazu unterscheidet der verwendetet Algorithmus zwischen

schon erkundetem und noch nicht erkundetem Gebiet. Anhand einer erstellten Explorationskarte wird dann mit einem geeignetem Verfahren ein Ziel zur weiteren Exploration gesucht. Danach wird der Weg zu diesem Ziel berechnet. Der in dieser Arbeit verwendete Algorithmus [WP07] ist zudem noch in der Lage, mehrere autonome Roboter auf der selben Karte zu unterstützen. Dies wird in dem Kapitel Exploration im praktischen Teil genauer erläutert.

2.4 Ansätze zur Pfadfindung

Pfadfindung beschreibt im einfachsten Fall das Finden eines Weges zwischen zwei Punkten. Dabei soll auch möglichen Hindernissen ausgewichen werden. Es gibt verschiedene Verfahren der Problemlösung, die je nach gegebenen Daten besser oder schlechter sind. Einige arbeiten auf gitterbasierten Karten der Umgebung andere auf Graphen mit Knoten für die verschiedenen Ziele.
Beispielsweise arbeitet der A*-Algorithmus auf gitterbasierten Karten. Jedes Gitter in der Karte bekommt einen Belegtheitswert. Des weiteren besitzt A* eine Schätzheuristik für den absoluten Abstand zum Ziel und eine Kostenfunktion für den bisher berechneten Weg. Mit der Heuristik wird der Fortschritt in Richtung des Ziels forciert. Diese Heuristik kann beispielsweise die euklidische Distanz zwischen aktuellem Punkt und dem Endpunkt sein. Dadurch bekommen Punkte, die weiter vom Ziel weg sind höhere Werte, auch wenn der Weg dorthin einfacher war. Gäbe es die Heuristik nicht, würde sich die Suche in alle Richtungen gleich erstrecken, der Algorithmus viel länger brauchen. Mit der Kostenfunktion wird berechnet, wie teuer der Weg bis zu diesem Punkt schon ist. Im einfachsten Fall wird der Wert pro gegangenem Feld um 1 erhöht. Die Summe der beiden Funktionswerte ergibt den jeweiligen Wert für jedes Feld.

$$f(n) = g(n) + h(n) \tag{2.21}$$

Hier ist g die Kostenfunktion und h die Heuristikfunktion. Nachdem das Ziel erreicht wurde, wird von diesem Feld aus der günstigste Pfad zum Startpunkt abgegangen. Der A*-Algorithmus benötigt dabei eine genaue Vorstellung, wo der Start und das Ziel auf der Karte vorhanden sind. A* ist eine Erweiterung des Dijkstra-Algorithmus um die Heuristik. Damit wird der Algorithmus für Fälle, in denen der Ort des Ziels bekannt ist, schneller. A* ist also eine Generalisierung des Dijkstra [Joh73] mit reduzierten Kosten. Dijkstra hat dabei den Vorteil, dass er Wege, die am nahe am Start sind bevorzugt und somit nicht direkt auf ein Ziel angewiesen ist, da er solange sucht bis er es findet.
Zudem gibt es noch einen Ansatz, der den Abstand zu einem Hindernis mit einfließen lässt. Aus diesem Grund wird erst eine Distanzkarte zu allen Hindernissen erstellt und danach mit den verschiedenen Parametern soweit angepasst, dass der

effizienteste Pfad für ein Objekt, in diesem Fall den Roboter, berechnet wird. Dieses Verfahren wird im praktischen Teil noch genauer erläutert.

2.5 ROS

ROS(Robot Operating System [ros] ist ein Framework für Roboter, welches das ganze Verhalten eines Roboters steuern kann. Die Entwicklung begann 2007 am Stanford Artificial Intelligence Laboratory und wird nun von dem Robotikinstitut Willow Garage weiterentwickelt. Die Hauptaufgaben von ROS sind die Hardwareabstraktion, die Gerätetreiber, der Nachrichtenaustausch zwischen mehreren Programmen oder Programmteilen und das Bereitstellen der Möglichkeit, dass die Software auf mehreren PCs läuft.

2.5.1 Architektur von ROS

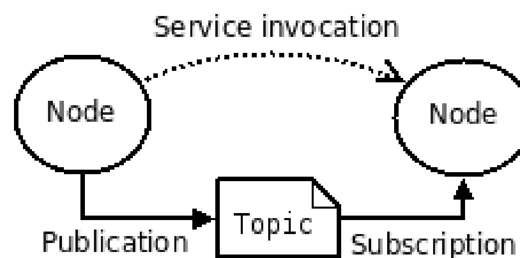

Abbildung 2.2: grobe Funktionsweise der Noden in ROS [ros]

Das Architektur von ROS ist servicebasiert(Bild 2.2). Dies bedeutet, jedes „Programm“bietet einen Service an, den andere Programme abonieren können. Wenn ein erwarteter Service nicht angeboten wird, wartet das Programm, bis es die erwarteten Daten bekommt. Jedes der Programme bekommt dabei einen eigenen Thread im System und ist somit unabhängig von den anderen. Sobald die erwarteten Daten nicht mehr synchron sondern asynchron sind, werden in ROS Topics verwendet. Ein Programm stellt ein Topic Bereit, welches abonniert werden kann. Dort können dann Daten in unregelmäßigen Abständen versendet werden. Der Kernprozess ist dabei der roscore, bei dem sich alle Programme anmelden müssen. Dieser wird auch Master genannt. Ohne diesen könnten sich die verschiedenen Nodes nicht finden. Nodes sind die laufenden Prozesse. Sie stellen Berechnungen an und senden und empfangen Daten. Die Kommunikation der verschiedenen Nodes, auch über Netzwerk, wird von ROS sehr einfach gestaltet. Die Programme arbeiten unter sich über eine Peer-to-Peer Verbindung. Die Kommunikation findet wie oben beschrieben über Topics und Services statt. Diese stellen Messages bereit. Dies ist

eine simple Datenstruktur, welche die zu empfangenden, sowie zu schickenden Daten beschreibt. Nur wenn der Typ der Message übereinstimmt, werden die Daten verwendet. Somit ist das System typsicher.
Die Topics arbeiten auf Basis von Sender(Publisher) und Empfänger(Subscriber). Ein einzelnes Node kann mehrere Topics senden als auch empfangen. Die Services dagegen bieten die Möglichkeit von Anfrage und Antwort. Dies bedeutet, dass der Service seine Informationen erst weitergibt, wenn er danach gefragt wird. Bei Treibern und deren Konfiguration oder ähnlichem ist dies recht sinnvoll. Eine Node kann sowohl Services als auch Topics gleichzeitig anbieten, was eine dynamische Einsetzbarkeit garantiert.
Der Programmaufbau, bzw. der Aufbau der Pakete, die als Prozess gestartet werden können ist auch intuitiv gestaltet. Ein sogenanntes Package ist im Grund ein Ordner, der alle Quelldateien sowie die ausführbaren Dateien und Verweise auf verschiedene genutzte Bibliotheken und Dateien bietet.

Kapitel 3

Hardware

In diesem Kapitel werden kurz die einzelnen Komponenten des Roboters und der Roboter selbst vorgestellt. Dabei ist der Schwerpunkt auf die Einsetzbarkeit der einzelnen Komponenten gelegt. Es wird auch ein kleiner Überblick über die Entstehung des Roboters und die einzelnen Prototypen gegeben. Das jetzige Modell ist auch noch ein Prototyp und immer noch in Entwicklung. Wichtig war es, dass die Hardware und die Software problemlos miteinander kommunizieren können. Das Raspberry Pi, welches später noch vorgestellt wird, bot sich dahingehend an, dass es GPIOs besitzt, die frei programmiert und angesteuert werden können. Dies in Verbindung mit einem ATMega 16 Mikrocontroller bildet die Steuerung des Roboters. Es wird nicht detailliert auf die Technik hinter den Komponenten eingegangen, da dies für die Arbeit eher unerheblich ist. Vielmehr wird das grobe Konzept des Roboters vorgestellt und eines seiner Bewegungsmuster.

3.1 Raspberry Pi

Das Raspberry Pi ist ein Miniaturcomputer auf Basis eines ARM 11. Er benötigt zum einen sehr wenig Strom(ca. 3.5 Watt) und durch seine geringe Größe und sein Gewicht war er die erste Wahl als Schnittstelle zwischen Roboter und PC. Mit seinem 700MHz Prozessor ist er in der Lage, die ausgewerteten Daten der Sensoren zu einem PC zu schicken und auch die Steuerung des Roboters zu übernehmen. Dies geschieht alles über WLAN und ist damit ortsunabhängig. Auf dem Raspberry Pi läuft ein Debian System, welches speziell für dieses Board entwickelt wurde. Das System wird von interenen Akkus des Roboters versorgt. In Bild 3.1 wird das verbaute Raspberry Pi in dem Roboter gezeigt. Durch seine geringe Größe kann auch der Roboter recht klein gehalten werden. Es sollte fähig sein, SLAM umzusetzen und die autonome Steuerung des Roboters zu übernehmen. Jedoch wurde schnell klar, dass das SLAM Verfahren und das Explorationsverfahren zu

viel Rechenleistung benötigen, als es das Raspberry Pi bieten könnte. Daher wurde das Raspberry Pi letztlich als Schnittstelle implementiert, welche die empfangenen Sensordaten an ein Topic über ROS sendet und Bewegungsbefehle von einem anderen Topic empfängt. Diese Befehle setzt es dann um, damit der Mikrocontroller den Roboter bewegen kann.

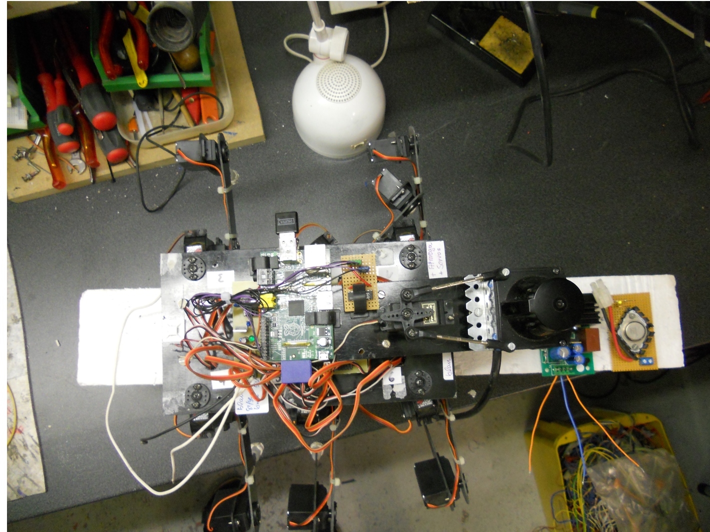

Abbildung 3.1: Raspberry Pi verbaut auf dem Roboter

3.2 Der Roboter

Die Idee war es, einen Roboter zu entwickeln, der sich in fast jeder Umgebung bewegen können sollte und dabei diese zu kartieren und sich zu lokalisieren. Zudem sollte er möglichst lange unterwegs sein können, also so leicht wie möglich sein und die Akkukapazität gut nutzen. Der entwickelte Roboter besteht aus einem Körper und 6 Beinen mit jeweils 3 Gelenken. Damit handelt es sich um einen Hexapod. Der Vorteil dieser Art von Roboter ist es, dass er sich auf nahezu jedem Gelände fortbewegen kann. Die ersten Prototypen bestanden nur aus der entwickelten Platine (Bild 3.7) und den Servomotoren. Der erste Prototyp in Bild 3.3 bestand aus preislichen Gründen noch aus Holz und hatte längere Beine als der bisherige.

Da das Material zu biegsam war und auch die langen Beine einen zu großen Hebel darstellten, wurde danach ein neuer Prototyp entwickelt. Es gab auch noch einige kleine Verbesserungen an der Platine, jedoch blieb die Grundfunktionalität gleich. In Bild 3.4 ist der zweite Prototyp zu sehen. Die Beine wurden verkürzt und stärkeres Material verwendet. Dieses besteht aus einer Art Epoxiharz und ist stabiler und nicht so flexibel wie das Holz des ersten Prototypen. Es wurden auch

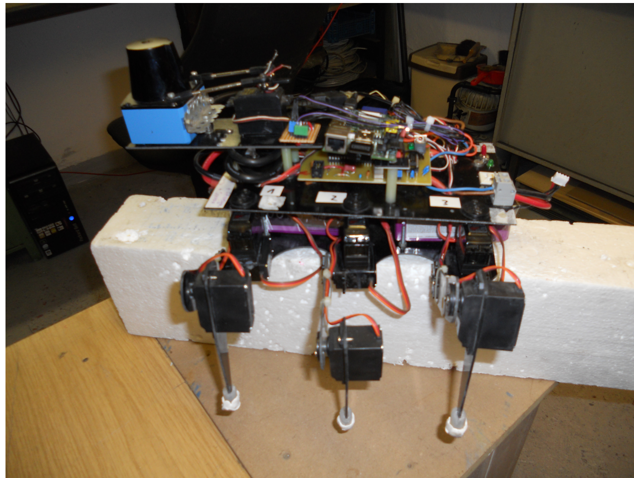

Abbildung 3.2: Dritter Prototyp des Roboters

neue Servomotoren verwendet, welche jedoch in späteren Tests das Gewicht des Roboters samt seiner Komponenten nicht mehr tragen konnte.

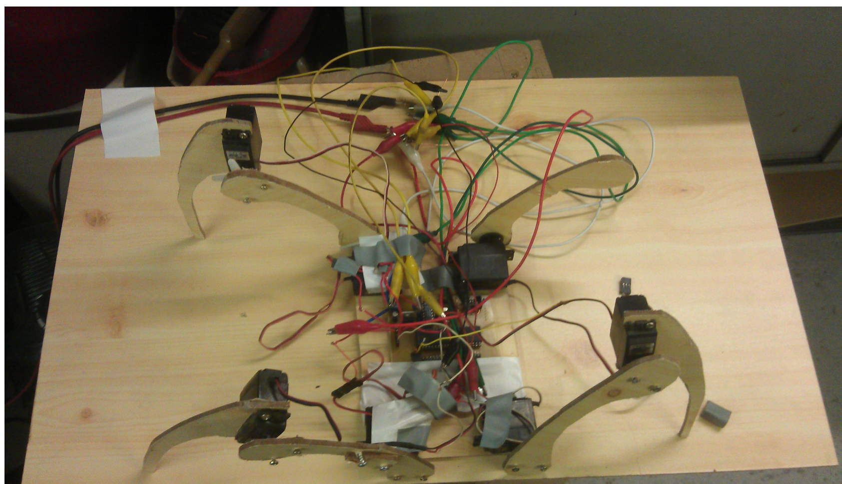

Abbildung 3.3: Erster Prototyp des Roboters

Der dritte Prototyp des Roboters ist in Bild 3.2 gezeigt. Es sind alle Komponenten verbaut und soweit wurde dieses System für einige Experimente benutzt. Er wurde mit einem SICK Laser Scanner, welcher die Daten für das Erstellen der Karte liefert und Akkus bestückt. Dazu kamen stärkere Servomotoren und leicht modifizierte Verkabelung, sowie Kühlung der einzelnen Komponenten. Ansonsten besteht er noch aus dem von mir entwickelten Boards mit einem ATMega16, welches für die Steuerung der Beine und des Kopfes zuständig ist. Die entwickelte Platine ist auf Bild 3.7 zu sehen. Dies war einer der ersten Prototypen der Platine,

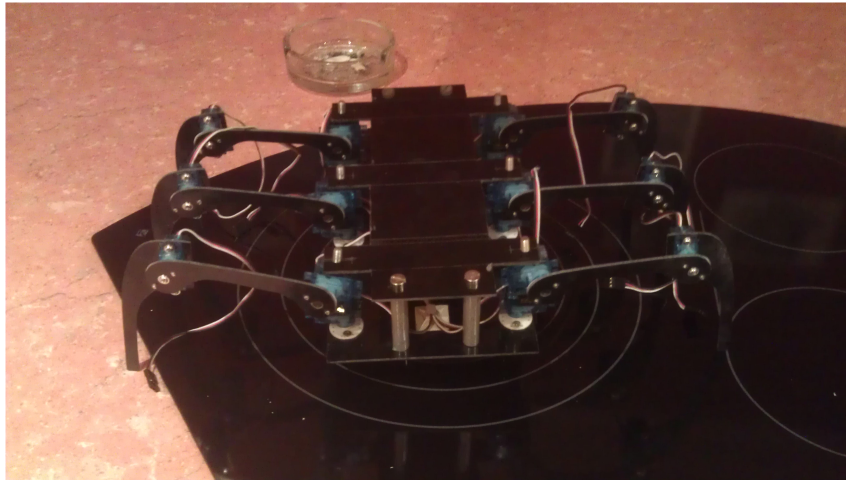

Abbildung 3.4: Zweiter Prototyp des Roboters

jedoch im Grundprinzip gleich. In Bild 3.1 und Bild 3.2 sind Teile der jetzigen Platine zu sehen.

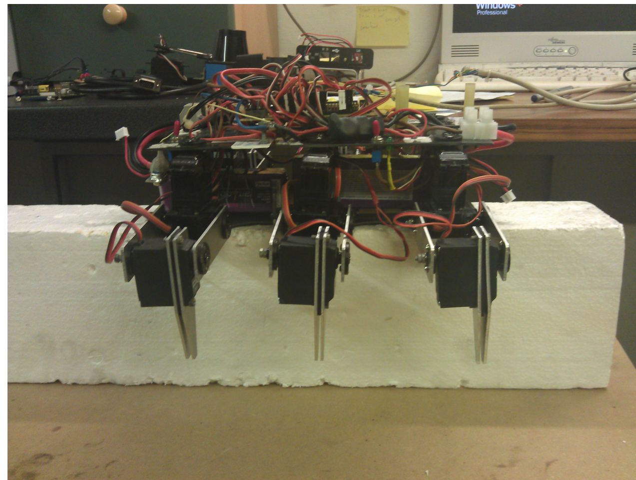

Abbildung 3.5: Aktueller Prototyp von der Seite. Die Beine sind aus Aluminium gefertigt und der Rest noch aus einer leichten Epoxidharzmischung.

Der Kopf lässt sich frei bewegen und somit kann die Neigung des Laser je nach Umgebung eingestellt werden. Im Laufe meiner Tests ist er auf Grundeinstellung geblieben. Zudem besitzt er noch jeweils Akkus für die Elektronik der Servos, der Schaltung und für den Laser. Der neuste Prototyp des Roboters wurde teilweise aus Aluminium gefertigt 3.5. Dies war nötig, da die Belastung an jedem Bein doch größer als erwartet war. Dazu wurden noch die Beinkonstruktion verstärkt, indem doppelte Streben verwendet wurden. Sie wurden auch wieder etwas verkürzt,

um die Kräfteverteilung auf die einzelnen Beine besser auszugleichen. Die neue Beinstruktur ist in Bild 3.6 zu sehen.

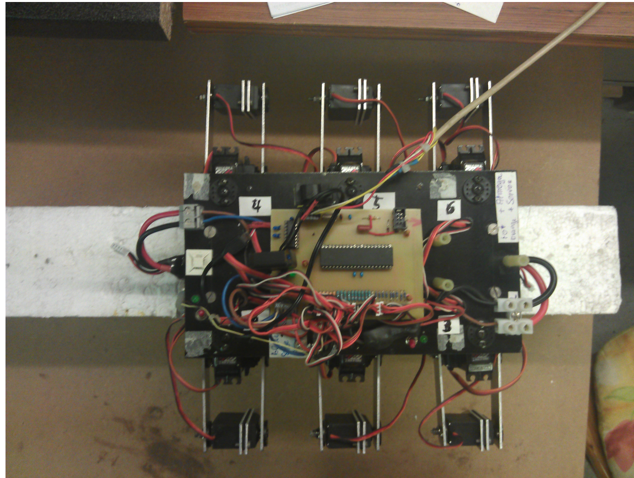

Abbildung 3.6: Aktueller Prototyp von oben. Zu sehen ist die neue Beinstruktur, sowie die neuste Platine des Roboters.

Der Rest des Roboters besteht aus einer Epoxidharzmischung. Die Beine sind jeweils kugelgelagert und die Füße wurden mit einer Gummibeschichtung zwecks besserer Reibung beschichtet.

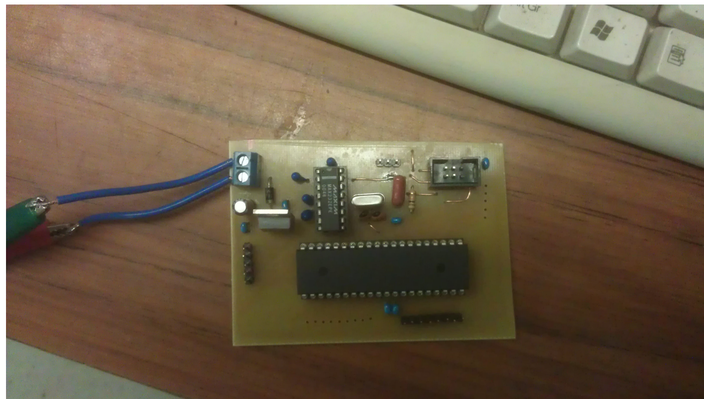

Abbildung 3.7: Prototp der Platine zur Steuerung des Roboters

Genauere Beschreibungen, sowie Einzelheiten der Programmierung werden im Anhang beigelegt. Dort werden auch die einzelnen Schaltpläne zu finden sein.

3.2.1 Fortbewegung

Zur Bewegung von sechs Beinen muss ein Verfahren gewählt werden, welches den Roboter in keine instabile Position bringt, jedoch auch effektiv und schnell sein sollte. Daher wurde im Rahmen der Arbeit ein Verfahren gewählt, welches jeweils drei Beine in der Luft hat und drei Beine, die den Roboter stützen. Dies ist eine der schnellsten Fortbewegungsmethoden, belastet den Roboter jedoch auch stark.

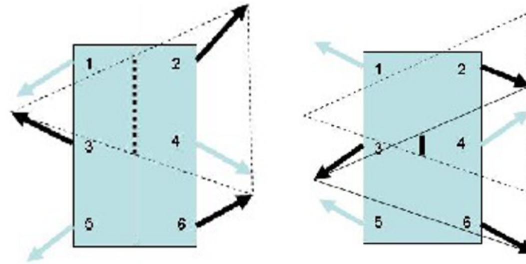

Abbildung 3.8: Laufschema des Roboters

Bild 3.8 zeigt die Art der Bewegung des Roboters. Dies zeigt nur das Vorwärts- bzw. Rückwärtsgehen des Roboters. Um Kurven gehen zu können müsste theoretisch eine Kreisbahn um einen Drehpunkt beschrieben werden, je nachdem wie steil die Kurve gegangen werden soll. In der Praxis wurde dies so umgesetzt, dass sich die äußeren Beine des Roboters in der Kurve schneller bewegen, als die inneren. Dies hat den Vorteil einer einfacheren Implementierung, aber auch eine schnellere Berechnung. Die Kreisbahn wird nur grob angenähert und es ist keine großartige Berechnung mit Gleitkommazahlen nötig. Dadurch können Bewegungsschritte fix gewählt werden, was sich in der Praxis kaum bemerkbar macht, jedoch den Mikrocontroller entlastet.
Ein Bein besitzt dabei 3 Freiheitsgrade, die Bewegung in alle 3 Achsen. Ein Prototyp ist in Bild 3.9 zu sehen. Das Prinzip des Beines ist geblieben, nur das Material und die Größe wurden angepasst.

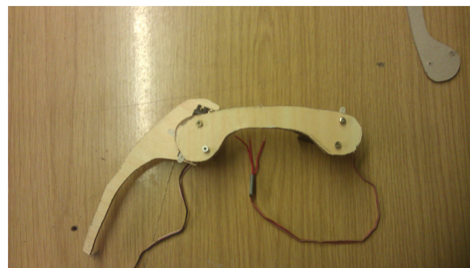

Abbildung 3.9: Prototyp eines Beines des Roboters

Es hat ein Kniegelenk für Auf- und Abwärtsbewegungen. das Oberschenkelgelenk kann sich zusätzlich noch zur Seite bewegen. Dies ermöglichst eine große Beweglichkeit und damit komplexe Bewegungsmuster. Das Material ist mittlerweile Aluminium, da es die Belastung der Hebelwirkung bisher am besten bewältigt.

Kapitel 4

Praktischer Teil

Hier werden die verschiedenen Verfahren, die weiter oben erklärt wurden in die Praxis umgesetzt. Dabei gibt es oft Abweichungen von der Theorie, da sie entweder so nicht umgesetzt werden kann oder es einen effizienteren Weg zu irgendeiner Berechnung gibt. Aufgeteilt ist dieses Kapitel zum einen in den Block SLAM und zum anderen in den Block Exploration. In Kartieren und Lokalisieren werden die beiden praktisch getesteten SLAM Verfahren Gmapping und Hector SLAM vorgestellt. In dem Teil der Exploration werden verschiedene Ansätze der Herangehensweise an dieses Problem vermittelt und die zugehörigen Teilmodule genauer erklärt. Des Weiteren wird noch auf die Kommunikation zwischen Roboter und PC und das Einwirken von ROS eingegangen.

4.1 Nutzung von ROS im Rahmen der Arbeit

In der Arbeit wurde vor allem das Topic Prinzip von ROS verwendet. Die Nodes kommunizieren so auch über Netzwerk miteinander. Dies war nötig, da das Raspberry Pi nicht leistungsstark genug war, alle Berechnungen selbst durchzuführen. Daher werden die Rohdaten über Netzwerk an einen stärkeren PC übermittelt und die ausgewertetetn Daten dann wieder an den Roboter. Ros bietet hier eine gute Möglichkeit der Kommunikation, da einfach die Master-URI, also die Adresse des Masters von „localhost“auf die entsprechende Adresse im Netzwerk geändert werden kann. Auch das Einbinden anderer Frameworks in ROS, wie z. B. QT oder auch ASIO für die Kommunikation über die serielle Schnittstelle mit dem Mikrocontroller stellte sich nach genügend Einlesen und Ausprobieren doch eher als einfach heraus.

4.2 Kommunikation mit dem Roboter

Der Roboter wurde so konzipiert, dass er über die UART(Universal Asynchronous Receiver Transmitter) kommuniziert. Die UART ist eine serielle Schnittstelle, die asynchrones Empfangen und Senden ermöglicht. Da das Raspberry Pi auch die Möglichkeit bietet, Signale an UART zu senden und zu empfangen, bot sich diese als beste Lösung an, den Roboter zu steuern. Das Raspberry Pi kann sich über WLAN ins Netzwerk verbinden und mit einer ROS-Node mit einem leistungsstärkeren PC kommunizieren, der aufgrund der empfangenen Daten alle Berechnungen ausführt und die Bewegungsbefehle für den Roboter wieder an das Raspberry schickt, welches dann die direkte Steuerung des Roboters übernimmt. Dies war die einzige Möglichkeit, SLAM in Echtzeit umzusetzen, da das Raspberry Pi in seiner Rechenleistung sehr beschränkt ist. Die Noden zur Kommunikation sind so aufgebaut, dass jeweils die Eingaben auf dem PC an ein Topic gepublished werden und die Node auf dem Raspberry Pi diese Befehle empfängt und auswertet. Zudem läuft auf dem Raspberry Pi noch eine Node, die die rohen Laserdaten über ein Topic verbreitet.

4.3 Kartieren und Lokalisieren

Hauptbestandteil der Arbeit war es, einen effizienten Algorithmus zur Kartierung der Umgebung zu integrieren. Dieser muss dabei auch die Selbstlokalisierung übernehmen, also SLAM lösen. Zu diesem Zweck wurden in der Arbeit zwei verschiedene Verfahren benutzt und getestet. Zum einen das GMapping Verfahren und zum anderen das Hector SLAM Verfahren. Die beiden Verfahren weisen grundlegende Unterschiede in der Umsetzung auf, weshalb diese beiden getestet wurden.

4.3.1 GMapping

GMapping basiert auf der Idee von FastSLAM 2.0. Es verwendet auch einen Rao-Blackwellized Partikelfilter [Mur99] und pro Partikel einen EKF, der die Pose des Roboters schätzt. Dabei enthält jedes Partikel eine Karte und einen EKF für jedes Feature, welches schon auf der Karte erkannt wurde. Die eigentlichen Verbesserungen gegenüber FastSLAM 2.0 [MTKW03] sind eine verbesserte geschätzte Verteilung und eine effizientere Möglichkeit, die Partikel zu verteilen, bzw. ein verbessertes Resampling. Ein Unterschied zwischen dem klassischen FastSLAM 2.0 und GMapping ist, dass nicht auf Feature-basierten Karten, sondern auf Gridmaps gearbeitet wird. Occupancy Gridmaps sind eine Art der Repräsentation von Karten. Sie bestehen aus einem Raster in dem für jede Zelle eine Belegtheitswahrscheinlichkeit existiert.

Das Rao-Blackwell Theorem beschreibt eine Möglichkeit, eine Schätzfunktion effizienter zu schätzen, indem die einzelnen Bedingungen aufgeteilt und separat berechnet werden. Dies bietet den Vorteil, dass die Zustandsschätzung

$$p(x_{1:t}, m|z_{1:t}, u_{0:t}) \tag{4.1}$$

in

$$p(m|x_{1:t}, z_{1:t})p(x_{1:t}|z_{1:t}, u_{0:t}) \tag{4.2}$$

unterteilt werden kann. Der erste Term $p(m|x_{1:t}, z_{1:t})$ kann dadurch effizient berechnet werden kann , wenn $x_{1:t}$ und $z_{1:t}$ bekannt sind [Mor88]. Der Teil $p(x_{1:t}|z_{1:t}, u_{0:t})$ wird durch den Partikelfilter approximiert. Dazu werden Samples von einer angenäherten Verteilung π genommen. Dies ist der Prediction Schritt des Partikelfilters. Nach [DGA00] ist die optimale Verteilung

$$p(x_t|m^{(i)}_{t-1}, x^{(i)}_{t-1}, z_t, u_t) \tag{4.3}$$

Nach Markov [FH96] sieht diese Verteilung dann wie folgt aus

$$\frac{p(z_t|m^{[i]}_{t-1}, x_t)p(x_t|x^{[i]}_{t-1}, u_t)}{\int p(z_t|m^{[i]}_{t-1}, x')p(x'|x^{[i]}_{t-1}, u_t)\mathrm{d}x'} \tag{4.4}$$

Diese Unterteilung ist sinnvoll, da nun bestimmte Eigenschaften der Verteilung auffallen und diese dadurch vereinfacht werden kann. Zum einen überwiegt die Wahrscheinlichkeit $p(z_t|m^{[i]}_{t-1}, x_t)$ gegenüber der Wahrscheinlichkeit $p(x_t|x^{[i]}_{t-1}, u_t)$ wie in Bild 4.1 gut zu sehen ist.

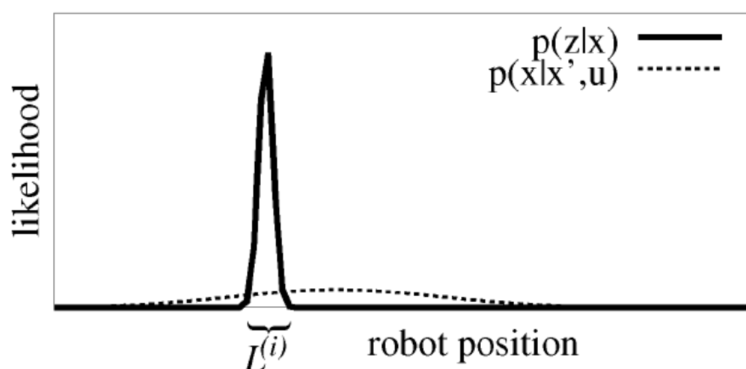

Abbildung 4.1: Gewichte der Verteilungen des Zustandes und der Messungen bei Schätzung der Roboterpose in GMapping

Dieser Teil wird in GMapping durch eine Konstante k in dem Intervall $L^{[i]}$ ersetzt. In diesem Intervall ist die Dichte am höchsten, den aktuellen Zustand zu erwarten. Dieser wird dann nur noch durch den Term $p(z_t|m_{t-1}^{[i]}, x_t)$ gebildet. Dort kann das Maximum der Verteilung über eine Gaussverteilung angenähert werden

$$p(x_t|m_{t-1}^{(i)}, x_{t-1}^{(i)}, z_t, u_t) \simeq N(\mu_t^{(i)}, \Sigma_t^{(i)}) \tag{4.5}$$

Anhand dieser neuen Verteilung werden auch die Gewichtungen des Partikelfilters $w_t^{[i]}$ neu berechnet. Die neuen Partikel werden dann um das lokale Maximum gestreut, welches der Scanmatcher gefunden hat. Die Partikel $\{x_j\}$ die über die Region gestreut werden, werden anhand der Unsicherheit der letzten Odometrie gestreut.

$$x_j \in \{x_t|p(x_t|x_{t-1}, u_t) > X\} \tag{4.6}$$

Das Bewegungsmodell hier kann durch einen EKF angenähert werden.
Durch diese neue Verteilung werden nur noch Partikel in direkter Umgebung der Maxima gestreut. Dies kann viele Partikel sparen und hat eine geringere Unsicherheit in der Streuung. Damit kann eine höhere Effizienz ermöglicht werden. Die angenäherte Verteilung wird also dahingehend verbessert, dass nur noch um lokale Maxima gesamplet wird. Anfänglich wird das lokale Maximum anhand eines Scanmatchers berechnet. Dann werden um dieses Maximum Samples mit einem Gewicht, abhängig von diesem Maximum gestreut. Die Gaussverteilung über diesen neuen Samples wird dann dazu verwendet, die neue Pose des Roboters durch weitere Samples zu ermitteln. Dann werden die Gewichte noch anhand der angenäherten Gaussverteilung korrigiert und der Algorithmus fährt wie bekannt mit dem Resampling fort.

Dieses Resampling wird in GMapping auch durch eine Verbesserung erweitert. Anstatt nach jedem Durchlauf des Algorithmus neu zu Resamplen, wird eine Variable N_{eff} eingeführt, die bestimmt, wann das Resampling nötig ist. Diese Variable beschreibt, wie gut das aktuelle Partikelset den wahren nachfolgenden Zustand [Liu96]. Die genaue Bedeutung von N_{eff} ist in [DGA00] zu finden. Je weiter die verschiedenen Gewichte auseinander liegen, also je höher die Varianz zwischen den Importance weights ist, desto geringer wird auch der Wert von N_{eff}. In [GSB05] wird als Schwellwert $\frac{N}{2}$ verwendet. ist N_{eff} unter diesem Wert, findet ein Resampling statt, ansonsten nicht. Diese reduziert nicht nur den Aufwand des Algorithmus, sondern reduziert auch die Anzahl an „guten“Partikeln, die durch das Resamplen gelöscht werden würden. Dieser Effekt entsteht, da ein Resampling nur noch stattfindet, wenn es nötig ist.
Damit ist GMapping eine Verbesserung des ursprünglichen FastSLAM 2.0 Algorithmus, sowohl hinsichtlich des Rechenaufwandes, als auch der Exaktheit. Dieses Verfahren hat sich mittlerweile in der Robotik etabliert und ist eines der am häufigsten verwendeten Verfahren [Abb].

4.3.2 Hector SLAM

Hector SLAM arbeitet anders als GMapping oder das zugrunde liegende Fast-SLAM 2.0. Es benutzt zur Erstellung der Karte ein Gauss-Newton Verfahren und einen EKF zur Bestimmung der Pose. Das Gauss-Newton Verfahren ist ein Scanmatcher, der versucht den Fehler zwischen zwei verschiedenen Scans zu minimieren, sodass diese mit möglichst kleiner Varianz übereinander gelegt werden können. Es ist also ein Minimierungsverfahren für Fehler. In der Arbeit benötigt der EKF zur Posebestimmung die geschätzte Position aus dem Scanmatcher, also dem Gauss-Newton Verfahren. Die aus dem Scanmatcher erzeugten Endpunkte werden dann dem Kalmanfilter übergeben. Der EKF verfährt dann wie in Grundlagen vorgestellt. Mithilfe des alten Zustandes, der Messung und der geschätzten Position aus dem Scanmatcher wird die aktuelle Pose korrigiert. Der Vorteil von Hector SLAM ist, dass dazu keine Odometrie nötig ist. Dies kam der Arbeit zugute, da es kein System auf dem Roboter zur Bewegungsmessung gibt. Odometrie kann jedoch trotzdem dem Modell hinzugefügt werden, damit die Schätzung genauer wird. Das Modell des Hector SLAM besteht aus 2 Teilen, die unabhängig voneinander arbeiten, jedoch die Daten des jeweils anderen brauchen. Dazu zählt das SLAM System, welches die Karte generiert und das Navigationssystem, welches die aktuelle Pose in der Karte schätzt(Bild 4.2).

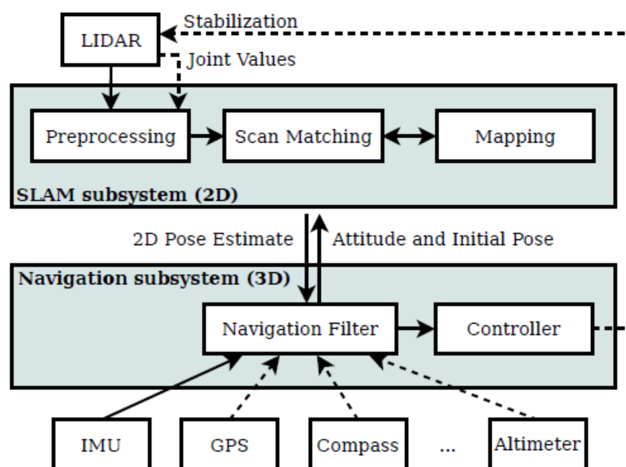

Abbildung 4.2: Prinzip von Hector SLAM [KMSK11]

Die Karte wird durch ein Occupancy Grid dargestellt. Da die Genauigkeit dieser Karte beschränkt ist, werden mehrere Grids mit verschiedenen Auflösungen benutzt, eine sogenannte multi-Level Karte. Das Scanmatching, also das Anpassen der jeweiligen Scans aneinander, wird über ein Gauss-Newton Verfahren geregelt.

Dieses Verfahren arbeitet ähnlich zu ICP und ist ein Minimierungsverfahren. Es wird also der minimale Fehler gesucht, unter dem 2 verschiedene Laserscans aneinander angepasst werden. Dadurch wird eine Rotation und Translation erzeugt, welche dann auf den zweiten Laserscan angewendet wird. Dadurch entsteht dann Schritt für Schritt eine Karte. Dieses Verfahren ist losgelöst von der Pose des Roboters oder anderen Einflüssen. Es arbeitet nur auf den eintreffenden Scans und der existierenden Karte. Diese Karte entsteht notfalls durch den ersten eintreffenden Scan auf den dann weiter Scanmatching betrieben wird. Die Formel des Gauss-Newton Verfahrens sieht wie folgt aus

$$\xi* = \operatorname{argmin} \sum_{i=1}^{n} [1 - M(S_i(\xi))]^2 \tag{4.7}$$

Dabei ist $\xi = (p_x, p_y, \psi)^{\mathrm{T}}$ die Transformation, die diese Funktion minimiert. $S_i(\xi)$ ist eine Funktion, die die Scanpunkte anhand der Transformation ξ in Weltkoordinaten transformiert. Die Funktion $M(S_i(\xi)$ gibt den Wert in der Karte an der Position $S_i(\xi)$ zurück. Falls also alle gescannten Punkte auch wirklich in der vorherigen Karte vorhanden sind kommt ein Fehler von 0 raus. Da dies aber selten der Fall ist, beginnt der Algorithmus mit einer geschätzten Starttransformation ξ. Diese kommt aus dem EKF und dessen verbesserten Pose des Roboters. Danach wird nach einem Gradientenverfahren das Minimum der Gleichung gesucht. Zu beachten ist hier, dass durch die Approximation nicht unbedingt ein Minimum erreicht wird. in der Praxis scheint dies dennoch gut zu funktionieren. Dies kann in [KMSK11] nachgelesen werden.

Das Navigationssystem verwendet den EKF zur Schätzung der Pose. Dazu wird er wie in Kapitel 2 vorgestellt verwendet. Er speichert nicht alle Features, die auf der Karte existieren, sondern arbeitet nur auf den aktuell gemessenen Endpunkten der Scans. Im Fall von Hector SLAM sieht das zugehörige Zustandsmodell wie folgt aus:

$$x = (\Omega^{\mathrm{T}}, p^{\mathrm{T}}, v^{\mathrm{T}})^{\mathrm{T}} \tag{4.8}$$

wobei $\Omega = (\phi, \theta, \psi)^{\mathrm{T}}$ die jeweiligen Eulerwinkel der Drehung im dreidimensionalen Raum beschreiben und $p = (p_x, p_y, p_z)^{\mathrm{T}}$ die aktuelle Position, sowie $v = (v_x, v_y, v_z)^{\mathrm{T}}$, sowie die Geschwindigkeit $v = (v_x, v_y v_z)^{\mathrm{T}}$. Das Bewegungsmodell wird dann noch mithilfe des gemessenen Inputvektors $u = (\omega^{\mathrm{T}}, a^{\mathrm{T}})$ erweitert. Hier beschreibt ω^{T} die Winkelgeschwindigkeit in alle Richtungen und a^{T} die Beschleunigung in alle Richtungen. Damit entsteht das Bewegungsmodell.

$$\dot{\Omega} = \mathbf{E}_{\Omega} \cdot \omega \tag{4.9}$$

$$\dot{p} = v \tag{4.10}$$

$$\dot{v} = \mathbf{R}_{\Omega} \cdot a + g \tag{4.11}$$

Die Matrizen $\mathbf{E}_{\Omega}$ und $\mathbf{R}_{\Omega}$ transformieren die Winkelgeschwindigkeiten und die Beschleunigungen in den Raum des Zustandes. g beschreibt die Gravitation, welche jedoch bei schlechteren Sensoren wegfällt, da diese dadurch nicht beeinflusst werden [KMSK11].

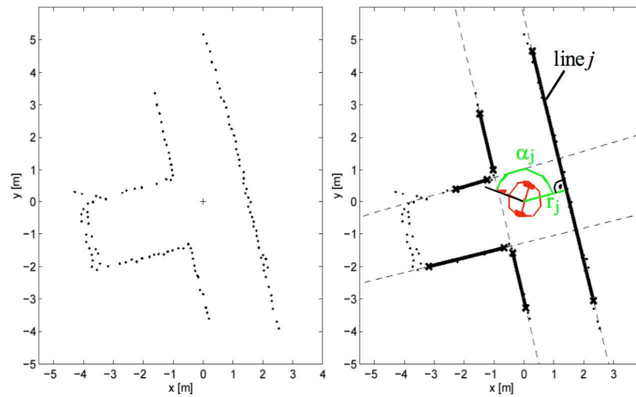

Abbildung 4.3: extrahierte Linien aus einer Messung zur Berechnung der EKF-Lokalisierung

In Bild 4.3 ist zu sehen, wie die Linien aus der Messung extrahiert werden. Dies ist ähnlich zu dem in Hector verwendeten Verfahren. Hier werden nur die Endpunkte verglichen statt Linien. Dann werden anhand der geschätzten Pose des Roboters und der aktuellen Karte die neuen Messungen vorhergesagt

$$\hat{z}_k = h(\hat{x}_k, m) \tag{4.12}$$

und ein Rauschen für die Unsicherheit der Position der erwarteten Endpunkte bestimmt.

$$\mathbf{H}\hat{C}_k\mathbf{H}^{\mathrm{T}} \tag{4.13}$$

Es wird versucht die gemessenen Daten mit den geschätzten Daten zu verbinden. Dieser Schritt wird auch Korrektur genannt und ist derselbe Schritt, wie im EKF Algorithmus beschrieben. Dort werden die Messungen auch mit dem aktuell geschätzten Zustand verknüpft. Der Vorteil hier ist, dass der EKF nicht alle Landmarken der Karte benutzt, sondern nur die, die erst in seinen geschätzten Messungen zu sehen glaubt. Dabei besitzt das Verfahren den Aufwand von $O(k^{2.376} + n^2)$ [WB10] wobei k die Dimension des Messungsvektors und n die Dimension des Zustandes ist. Das Verfahren kann jedoch divergieren, wenn die nichtlinearen Anteile in der Verteilung zu groß sind. Da das SLAM System immer ein

Feedback über die Position gibt, kann die Lokalisierung über den EKF auch ohne Odometrie stattfinden. Dazu wird einfach die Transformation, die durch den Scanmatcher errechnet wurde auf die Position des Roboters angewendet und eine wahrscheinliche neue Position des Roboters errechnet. Mit diesen IMU Daten arbeitet dann die EKF-Lokalisierung, also der Positionsschätzer. Mithilfe der dadurch errechneten neuen Pose des Roboters wird dann die initiale Transformation für den Scanmatcher berechnet. So greifen beide Systeme ineinander über. Dabei läuft die Positionsschätzung mit einer höheren Frequenz als das Erstellen der Karte und die Daten werden untereinander asynchron ausgetauscht [KMSK11].
In dieser Arbeit wurde Hector SLAM als Verfahren der Wahl verwendet, da es unabhängig von Odometrie arbeiten kann und dennoch sehr gute Ergebnisse liefert. GMapping hat in den Tests ohne Odometrie keine guten Ergebnisse geliefert. Erst nachdem durch einen separaten Scanmatcher eine simulierte Odometrie erzeugt wurde, lieferte auch dieses System gute Ergebnisse. Da Hector SLAM in diesem Fall einfacher zu implementieren war und auch gute Ergebnisse bei wenig Aufwand erzielte, wurde dieses genommen, was später im Kapitel Experimente noch detaillierter hervortritt.

4.4 Exploration

Ohne eine Exploration kann ein autonomer Roboter nicht auf fremdem Gebiet agieren. Der Roboter erkundet seine Umgebung Schritt für Schritt und sucht dabei nach verschiedenen Kriterien den optimalen Explorationspunkt. Dieser ist derjenige Ort auf der Karte, welcher sich für den Roboter am geeignetsten erweist, um an dieser Stelle weiter die Karte zu erkunden. Die Kriterien dafür sind zum einen die Entfernung des Punktes zum Roboter, da es sich als effizienter erweisen kann, immer den nächstgelegenen Punkt anzusteuern. Zum anderen ist auch die Größe des Ortes wichtig, die der Roboter als nächstes erkunden soll, da er selbst auch eine gewisse Größe hat und der Ort, welcher an sich nur eine Grenze zu einem unbekannten Gebiet beschreibt, mindestens so groß sein muss, wie der Roboter selbst. Dabei gibt es verschiedene Ansätze, welche Kriterien den nächsten Ort der Erkundung beschreiben. Der hier verwendete Ansatz bezieht sich auf Grenzen zwischen bekannten und unbekanntem Gebiet [YSA98, Wir07]. Anhand dieser Grenzen wird dann die „beste“ausgewählt, zu der sich der Roboter begeben soll. Es gibt auch noch andere Ansätze die auch eine „Safe Region“einführen [GBL02].

4.4.1 Distanztransformation

Während der Exploration muss zu beliebigen Startpunkt ein festes Ziel bei bekannter Karte gefunden werden. Eine Lösung dazu stellt [JB86] mit der Distanztrans-

formation vor. Dieses Verfahren arbeitet auf einem Occupancy Grid, welches in jeder Zelle den Abstand zu dem gesuchten Ziel speichert. Dieser Abstand kann auf verschiedene Weisen berechnet werden, z. B. durch die Schachbrettdistanz oder die euklidische Distanz. Das Ziel bekommt dann einen Wert von 0 zugewiesen und die anderen Zellen einen Wert von ∞. Dann wird ausgehend vom Ziel jede Nachbarzelle (in 8ter Nachbarschaft) gewählt, die frei ist und den kleinsten Abstandswert zum Ziel hat. Dann wird die Distanz zwischen den beiden Zellen anhand einer Distanzfunktion bestimmt und durch die Berechnung

$$if(w(z_j) + d < w(z_i))w(z_i) = w(z_j) + d \tag{4.14}$$

werden die jeweiligen Abstandswerte in den Zellen aktualisiert. Dabei ist d der Abstand zwischen zwei Zellen und w ist der Abstandswert in der jeweiligen Zelle. Durch dieses Verfahren entsteht eine Distanzkarte der Umgebung. Anhand der jeweiligen Position des Roboters kann durch Verfolgung des steilsten Gradienten der kürzeste Weg zum Ziel gefunden werden.
In der Arbeit wurde die Meijster Distanztransformation gewählt, da diese einfach zu implementieren war und einen linearen Aufwand besitzt. Dies wird im unteren Kapitel Distanzkarte detaillierter gezeigt.

4.4.2 Pfadtransformation

Das oben beschriebene Verfahren geht dabei aber nicht auf die Größe des Roboters ein. Dies bedeutet, dass es häufig vorkommt, dass der gefundene Pfad sehr nah an Hindernissen entlang führt. Da es jedoch auch ein wichtiges Kriterium für die Exploration ist, dass der Roboter den gefunden Weg auch befahren kann, hat [ZCS91] die Distanztransformation um eine Komponente der Sicherheit erweitert. Dazu wird eine Hinderniskarte erstellt, welche in jeder ihrer Zellen den Abstand zu der nächsten belegten Zelle speichert. Dieses Verfahren ist ähnlich zur Distanztransformation und nennt sich Hindernistransformation.

Ein Beispiel zu einer so erstellten Hinderniskarte ist in Bild 4.4 zu sehen. Die Pfadtransformation nach [ZCS91] ist dann definiert als

$$\Phi(z) = \min_{Z \in X}(l(Z) + \alpha \sum_{z_i \in Z} c_{\text{danger}}(z_i)) \tag{4.15}$$

Dabei ist X die Menge aller möglichen Pfade zwischen z und dem Ziel.
$l(Z)$ ist die Länge des Pfades Z.
$c_{\text{danger}}(z_i)$ ist eine Kostenfunktion für die Gefährlichkeit der Zelle.
α ist eine Gewichtung dieser Gefahr, also wie stark die Kostenfunktion Einfluss auf den Pfad nimmt.
Die Kostenfunktion c_{danger} wird mithilfe der Hinderniskarte bestimmt. Dazu kann

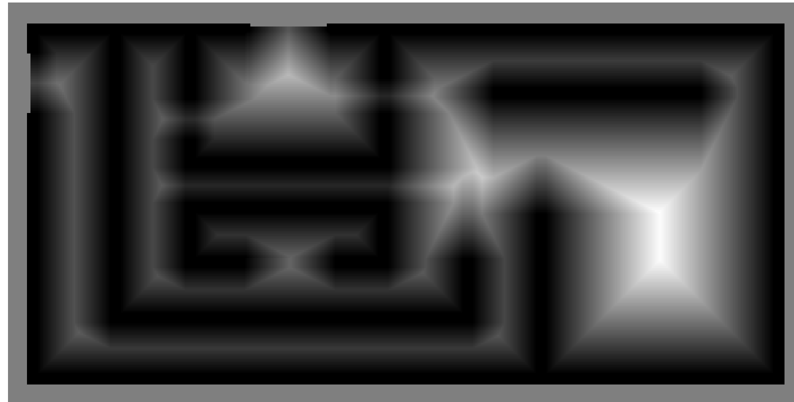

Abbildung 4.4: Karte nach Hindernistransformation. Je heller der Wert, desto weiter ist ein Hindernis entfernt [Wir07].

eine Funktion gewählt werden, die ab einer festen Distanz den Wert 0 annimmt und sonst, je geringer die Distanz wird, steigen [WP07]. Aber auch andere Kostenfunktionen sind denkbar. Die Pfadtransformation speichert dann die Kosten für den minimalen Weg in jeder Zelle. Damit kann wieder, wie in der Distanztransformation, dem steilsten Gradienten gefolgt werden, um das Ziel zu erreichen. Der Unterschied besteht hierbei in der Kostenfunktion und dem Parameter α, welcher den Grad angibt, ob ein sicherer Weg oder ein schnellerer Weg bevorzugt wird.

4.4.3 Verwendeter Explorationsalgorithmus

Der in dieser Arbeit verwendete Algorithmus entspricht im Groben der Idee von [WP07]. Die Grundidee ist das Kombinieren von grenzenbasierten Explorationsverfahren über eine Distanzkarte [YSA98] mit der beschriebenen Pfadtransformation. Dieses Verfahren nennt sich Explorationstransformation(Bild 4.5).

Eine Grenze ist das Gebiet zwischen bekanntem und unbekanntem Gebiet, welches für den Roboter interessant zur Exploration ist. Dazu werden auf der ganzen bekannten Karte Punkte gesucht, die schon erkundet wurden, deren Nachbar jedoch unbekannt ist. Nachdem so jeder Grenzpunkt auf einer Karte erfasst wurde, werden dessen Nachbarn in eine Schlange hinzugefügt. Für diese Punkte muss der Wert der Explorationstransformation bestimmt werden. Dies ist der erste Schritt des Algorithmus. Er ist ähnlich zu dem Flood-Filling Algorithmus und dieser Schritt entspricht dem „Säe“. Von diesen Punkten aus, also den potentiellen Zielen, „wird eine Art Welle, die die Kosten entsprechend ihres Verlaufs inkrementiert“,erzeugt [WP07]. Dann wird für jede Zelle ein Explorationswert berechnet, welcher beschreibt, wie lohnenswert diese Zelle , unter gewissen Bedingungen, ist. Dazu zählen die Bewegungskosten von den Nachbarzellen zu dieser Zelle, eine

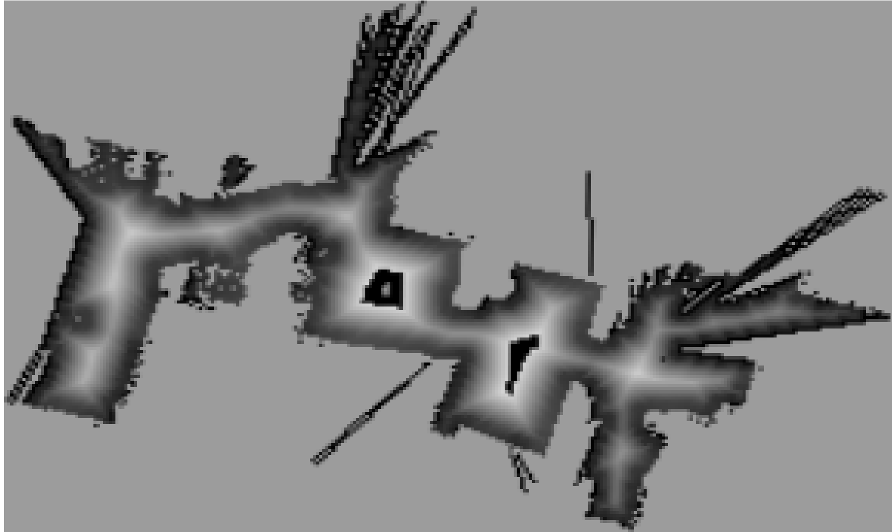

Abbildung 4.5: Explorationstransformation visualisiert

gewichtete Kostenfunktion und den kleinsten schon berechneten Wert der Nachbarzellen. Falls ein Wert verändert wird, werden dessen Nachbarn wieder in die Schlange eingefügt. Der Algorithmus terminiert, wenn die ganze Schlange abgearbeitet wurde. Somit wird jede Zelle der Karte mindestens einmal abgehandelt. Nach [WP07] ist der Algorithmus auch bei kleiner Rechenleistung sehr effektiv und war daher die erste Wahl für die Arbeit.
Die Explorationsransformation kann formal durch

$$\psi(c) = \min_{c_g \in F}(\min_{C \in X_c^{c_g}} (l(C) + \alpha \sum_{c_i \in C} c_{\text{danger}}(c_i))) \tag{4.16}$$

beschrieben werden. Dabei entspricht F der Menge aller Grenzpunkte , $X_c^{c_g}$ sind alle Pfade von c nach c_g, l(C) beschreibt die Länge eines Pfades C, $c_{\text{danger}}(c_i)$ ist die Kostenfunktion, welche für jeden Pfadpunkt c_i die Unkosten berechnet. α ist eine die Gewichtung der Kostenfunktion. Der Algorithmus ist ähnlich zu der Pfadtransformation nach [ZCS91], erweitert um das Suchen nach der nächsten Grenze. Die Explorationstransformation nimmt dabei an allen Gebieten nahe von Grenzen geringe Werte an. Die Kostenfunktion c_{danger} belegt damit Wegpunkte nahe an Hindernissen und auch Wegpunkte die zu weit von Hindernissen entfernt sind, vermieden werden. Das zweite dient dazu, dass Lokalisierung durch begrenzte Sensoren auch bei weit entfernten Hindernissen noch möglich ist [WP07].

$$c_{\text{danger}}(z) = \begin{cases} \infty, & \textit{wenn } d < d_{\min} \\ |d_{\text{opt}} - d|, & \textit{sonst} \end{cases} \tag{4.17}$$

d_{min} und d_{opt} sind Werte, die den minimalen Abstand zu Hindernissen und den optimalen Abstand beschreiben. Diese können frei, je nach Robotergröße gewählt werden. Durch den Betrag des Abstandes im zweiten Fall bekommen auch weite Entfernungen einen hohen Wert.
Zur Berechnung der Distanzkarte für den Abstand d wurde die Meijster Distanztransformation benutzt. Diese wurde in dem Paper [MRH02] vorgestellt und später noch etwas genauer erklärt. Nach der Berechnung der Explorationstransformation kann der kürzeste Weg zur nächsten Grenze , wieder durch Verfolgen des steilsten Gradienten, gefunden werden. Der Vorteil dieses Verfahrens ist, dass in jedem Punkt der bekannten Karte die Kosten zur nächsten Grenze stehen. Damit kann auf der ganzen Karte ohne neues Berechnen der kürzeste Pfad bestimmt werden. Auch mehrere Roboter können auf derselben Explorationstransformation arbeiten [WP07].

4.4.4 Meijster Distanzkarte

Die Distanzkarte, die durch die Meijster Distanztransformation entsteht, hat den Vorteil, dass sie mit linearem Aufwand in der Anzahl der Pixel generiert wird(Bild 4.6).

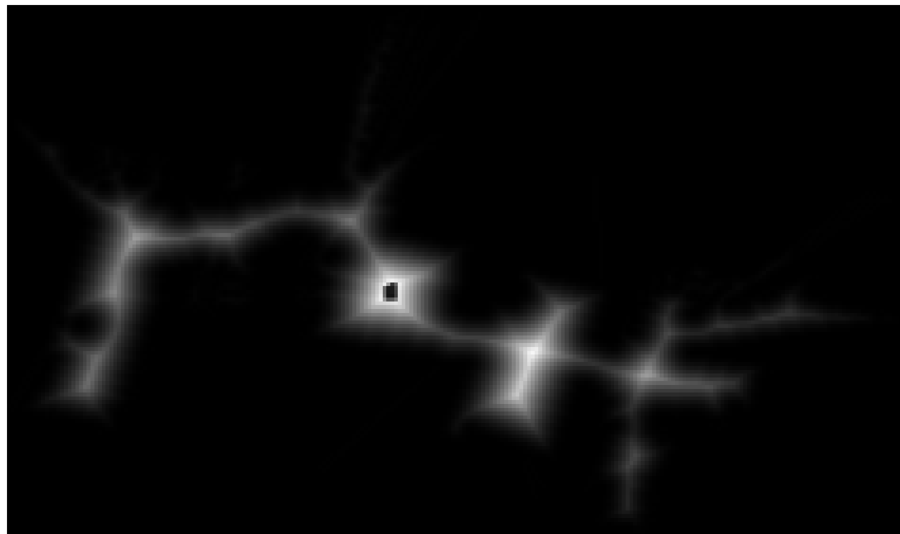

Abbildung 4.6: Distanztransformation nach Meijster

Dazu wird das Berechnen der Distanzen in zwei Teile aufgeteilt. Dazu wird das 2D-Problem, die kürzesten Distanzen in Vertikale und Horizontale zu bestimmen, in zwei 1D-Probleme geteilt. Dabei werden zuerst in der Vertikalen und danach in der Horizontalen alle Punkte abgearbeitet. Dabei geht der Algorithmus zuerst die y-Achse nach unten und danach wieder nach oben. Dabei werden die Distanzen zu

einem gesuchten Pixel in die y Richtung bestimmt. Die so entstandenen Werte werden als Menge von Funktionen angesehen, wobei jedes x pro Bildzeile eine Funktion repräsentiert. Danach werden jeweils Schnittpunkte der Funktionen gesucht und das Bild in verschiedene Regionen geteilt. Dieser Schritt geschieht von links nach rechts über das Bild. Anhand der Schnittpunkte können die Werte der jeweiligen Punkte bestimmt werden. Im letzten Schritt wird dann von rechts nach links die wirkliche Distanz bestimmt. Dabei können auch ungerade Werte herauskommen. Der Beweis zu diesem Verfahren ist sehr komplex und kann in [MRH02, Kra12] nachvollzogen werden.

Kapitel 5

Experimente

Dieses Kapitel lässt sich in mehrere Teile gliedern. Zuerst wurden die einzelnen Komponenten des Roboters einzeln getestet. Das bedeutet, dass sowohl die Einsatzmöglichkeiten des Laserscanners, sowie die Bewegung und Belastbarkeit des Roboters und auch die zugehörige Software getestet wurden. Dies wurde dann in das Framework ROS integriert und nochmals getestet. Am Ende wurde dann ein System aus allen Komponenten gefertigt und dieses in unbekanntem Gebiet ausgesetzt. Ziel war es, eine komplette Karte, sowie die aktuelle Position des Roboters zu erfassen. In den Tests wurde ein 1,6 GHz Netbook und das 700 MHz starke Raspberry Pi verwendet. Der Mikrocontroller auf welchem die Bewegung berechnet wird, hat eine Leistung von 16 MHz Im Folgenden werden die einzelnen Aufbauten und Experimente vorgestellt.

5.1 Kartierung

Die Kartierung wurde anfangs mit GMapping umgesetzt. Da es jedoch keine Odometriedaten des Roboters gibt und GMapping diese benötigt, musste eine andere Möglichkeit gefunden werden. In Bild 5.1 ist eine falsch erstellte Karte zu sehen. Hier wurde eine gefälschte Odometrie benutzt, die ein Stillstehen simulierte.

Eine Möglichkeit diesen Fehler zu umgehen ist es, vor dem eigentlichen GMapping einen Scanmatcher über die Scandaten laufen zu lassen, welcher die wahrscheinliche Odometrie ermittelt. An Bild 5.2 erkennt man die deutliche Verbesserung des Verfahrens.

Da es jedoch rein konzeptuell wenig Sinn ergibt zwei Scanmatcher zu verwenden und dabei die Odometrie zu simulieren, wurde in der Arbeit auf Hector SLAM zurückgegriffen. Dieses Verfahren benötigt nicht zwangsweise Odometriedaten, um gute Ergebnisse in Indoorbereichen zu erzielen. Es ist auch vom Grundgedanken her simpler als GMapping, wenn auch gedanklich nicht so ausgereift. Für die Ar-

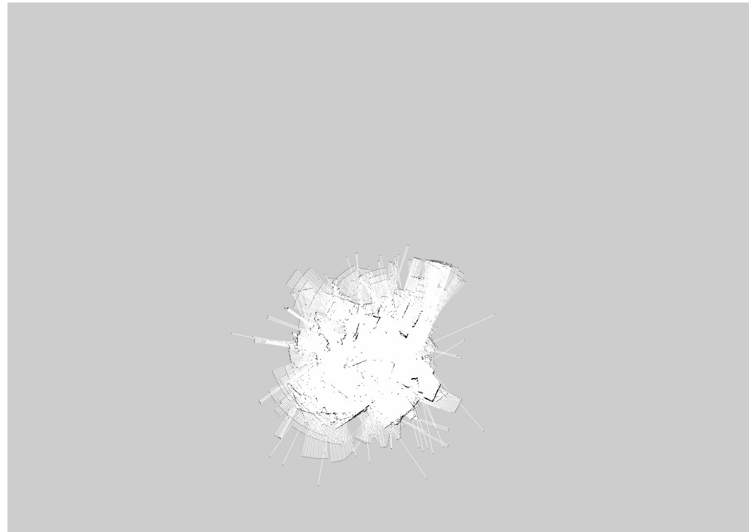

Abbildung 5.1: Falsche Karte erstellt von GMapping ohne Odometriedaten

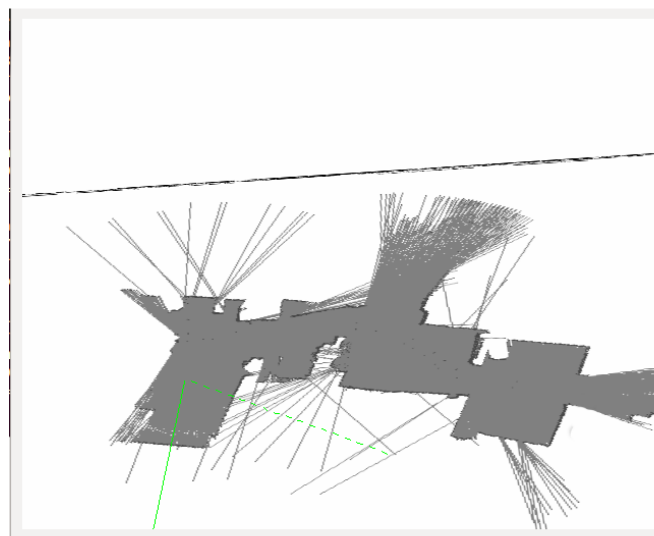

Abbildung 5.2: Karte erstellt von GMapping mit Odometriedaten eines Scanmatchers

beit war dieses Verfahren jedoch mehr als ausreichend. In Bild 5.3 ist eine durch Hector SLAM erstellte Karte und die erechnete Pose des Roboters zu sehen. Die Tests zeigten, dass sowohl die Bestimmung der Pose als auch das Erstellen der Karte sehr genau waren. Dabei wurde eine aufgenommene Datei des Roboters mitsamt aller Laserscans und Daten die Hector SLAM, als auch GMapping brauchte abgespielt und das jeweilige SLAM Verfahren gestartet. Beide Verfahren waren unter in

Echtzeit auch auf dem schwachen 1,6 GHz Rechner umsetzbar. Dies konnte durch die Software und deren GUI veranschaulicht werden.

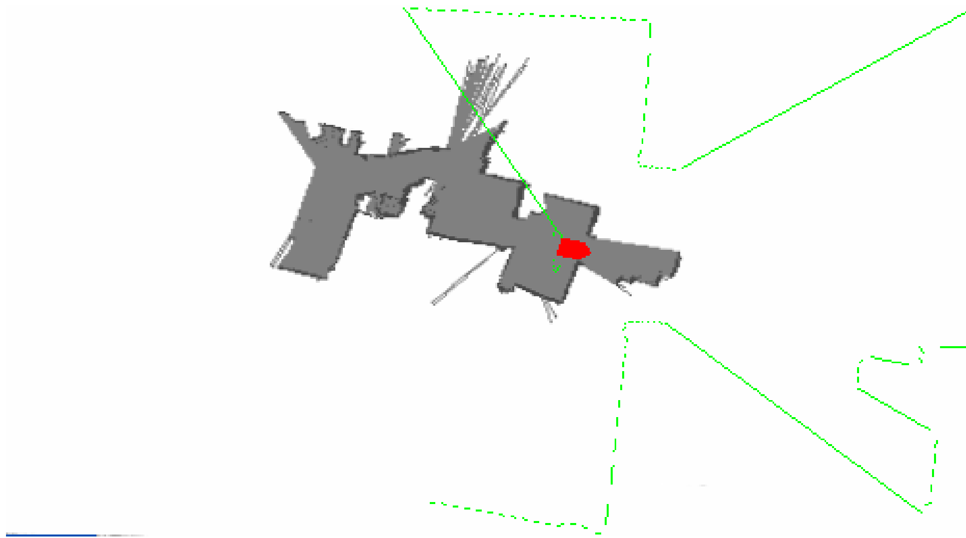

Abbildung 5.3: Karte erstellt von Hector SLAM

5.2 Roboter

Der erste Prototyp des Roboters bestand aus leichtem Holz und war noch mit schwachen Servomotoren ausgestattet. Daher waren die ersten Gehversuche des Roboters aufgrund der unterschätzten Belastung der einzelnen Beine recht schwierig. Nachdem sowohl die Akkus als auch der Laserscanner das erste Mal auf dem Roboter montiert waren, waren die 3 kg Gewicht in der Bewegung für 3 Beine nicht mehr zu tragen. Nach Verbesserung des Gehäuses und Verkürzung der Beine war die Belastung zwar geringer, jedoch wurde auch das neue Material, welches aus einem Gemisch aus Epoxidharz bestand, zu stark belastet. Die aktuelle Version verwendet nun zum größten Teil Aluminium und sollte damit die Belastungen aushalten. Auch die Servomotoren wurden durch leistungsstärkere Servomotoren ausgetauscht, welche auch noch Platz für mehr Gewicht ermöglichen.

5.3 Exploration

Da Testen der Exploration wurde auf der Karte, die Hector SLAM erstellt hat, durchgeführt. Dazu wurde während der Laufzeit des SLAM Verfahrens einige Male die Exploration gestartet. Es wurde der selbe PC verwendet wie zuvor. Die Tests

ergaben, dass der Explorationsalgorithmus nach [Wir07]mithilfe der Distanzkarte nach Meijster [MRH02] fast in Echtzeit implementiert werden konnte. Dies veranschaulicht die nachstehende Tabelle

Größe der Karte	Exploration	Distanzkarte	Grenzen finden	Pfad finden
200 x 200	0.18 s	0.43 s	-	0.3 s
1024 x 1024	1.2 s	0.6 s	-	0.3 s

Hier wurde die Zeit bei Eintreten in das Verfahren und die Zeit bei Austreten aus dem Verfahren gemessen. Die Differenz ergab dann die Zeitspanne, die für einen Durchlauf verbraucht wurde. Mithilfe der Meijster Distanztransformation konnte die Distanzkarte mit linearem Aufwand berechnet werden, was das System noch etwas beschleunigte. Das Finden der Grenzen war von dem von mir gewählten Messverfahren nicht mehr messbar. Das Finden des Pfades läuft rekursiv über die Minima der 8-er Nachbarschaft eines jedes Pixels. Dieses Verfahren ist nicht optimiert, soll aber noch verbessert werden.
Das Finden der Grenzen war nach Implementierung des geeigneten Verfahrens, hier war es ein Flood-Filling Algorithmus, sehr effizient, wie der Tabelle oben entnommen werden kann. In Bild 5.4 sieht man jede gefundene Grenze als rot und die längste als grün visualisiert. Jedoch ist auch zu sehen, dass die Exploration bei sehr großen Karten schnell langsam wird. Dies ist aber bei einer Größe von 1024 x 1024 nicht weiter verwunderlich und durch einige Optimierungen noch weiter zu verbessern. Die Pfadfindung bleibt jedoch nahezu konsistent, da die gefundenen Pfade bisher nie sehr weit von einem Roboter entfernt waren. Auch hier gilt es noch einige Verbesserungen zu finden.

Diese Grenzen wurden zur Weiterverarbeitung in ein Binärbild extrahiert(Bild 5.5). Anhand dieser Daten kann der Explorationsalgorithmus die Grenzen erkennen und die besten Pfade zu diesen berechnen. Im weiteren können auch andere Explorationsverfahren mithilfe dieser Linienfindung umgesetzt werden, da sie sowohl nächste als auch größte Grenze liefert.

5.4 Software + ROS

Anfangs wurde eine Software auf Basis von Qt entwickelt, die alle Berechnungen und Annahmen über den Roboter grafisch darstellt. Diese war zur Evaluierung der Geschwindigkeit der Algorithmen und der Funktionalität gut geeignet. Auch ROS lies sich gut integrieren und konnte die eintreffenden Werte schnell verarbeiten. Dabei besteht das Programm aus einer GUI von Qt und einem OpenGL Fenster. Durch die GUI lassen sich Einstellungen, wie z. B. das Kartierungsverfahren, ändern und durch OpenGL wurden die Position des Roboters, sowie empfangene Laserdaten und die erstellte Karte dargestellt(Bild 5.6). Zur besseren Überprüfung

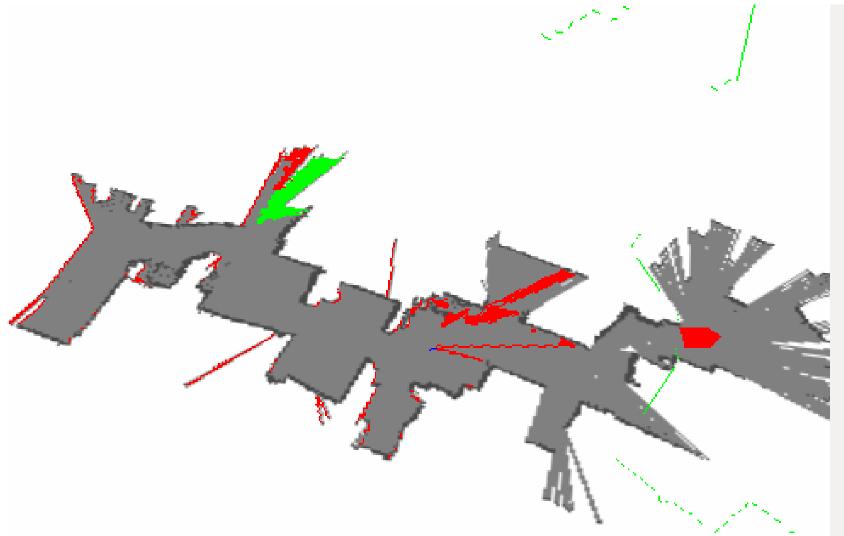

Abbildung 5.4: Abbildung zeigt gefundene Grenzen auf Testdaten. Grenzen sind rot markiert. Die längste Grenze ist grün markiert.

Abbildung 5.5: Extrahierte Grenzen in einem Binärbild visualisiert. D.ent zur Weiterverarbeitung der Daten.

der Berechnungen wurden dann alle Daten visuell dargestellt(Bild 5.7). Das Programm wurde auf einer Linuxplattform entwickelt und integriert und ist daher für Windowssysteme noch nicht geeignet.

In Bild 5.7 werden sowohl die Distanzkarte, alle extrahierten Linien und die gesamte Explorationstransformation dargestellt. Das Programm bietet zudem eine Möglichkeit, den Roboter manuell zu steuern. Dazu werden alle Tastendrücke

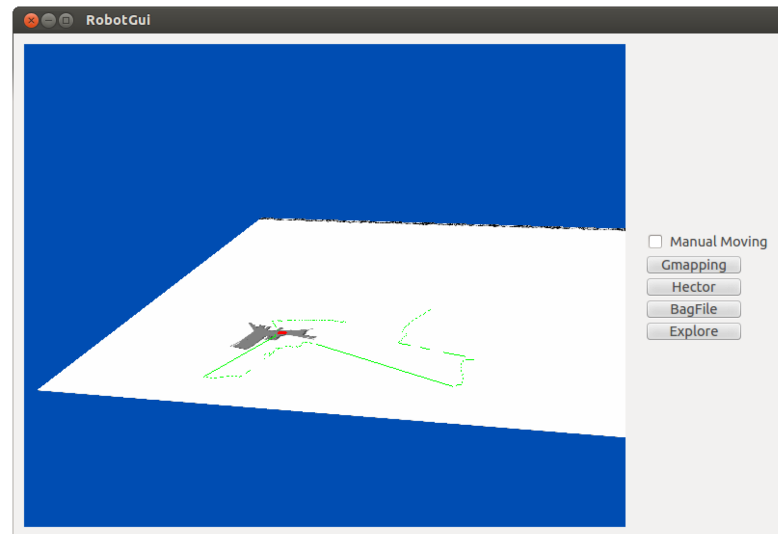

Abbildung 5.6: GUI in Verbindung mit empfangenen Daten des SLAM Verfahrens

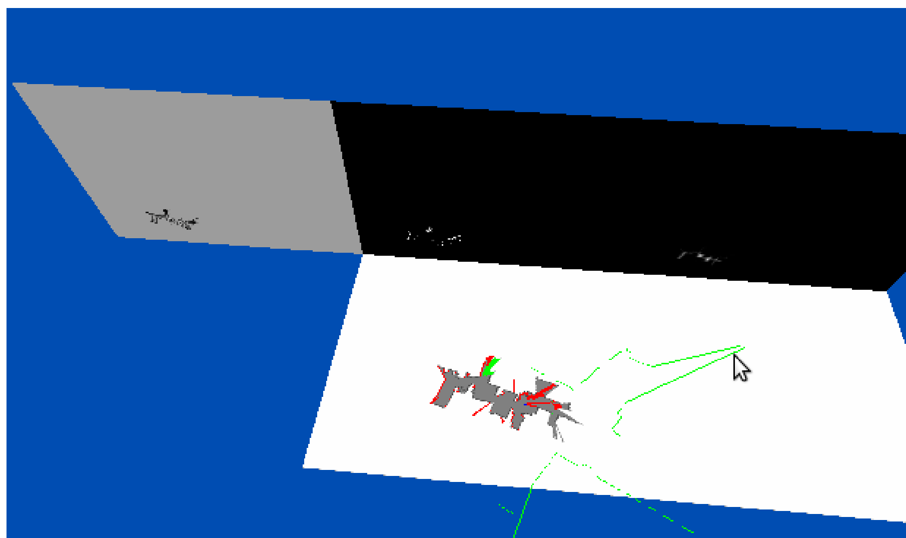

Abbildung 5.7: GUI mit verschiedenen Visualisierungen

abgefangen und an ein Topic in ROS weitergeleitet. Dieses kann dann von dem Roboter abonniert werden. Dabei war die Verwendung des Frameworks ROS sehr intuitiv gestaltet und lies sich gut mit Qt verbinden. Sowohl die Hardware- als auch die Softwareschnittstellen der verschiedenen Komponenten konnten gut miteinander kommunizieren. Dies war vor allem hilfreich bei der Verwendung des Raspberry Pi in Verbindung mit einem herkömmlichen PC.

5.5 Kompletter Aufbau

Der komplette Aufbau hat vorgesehen, dass der Roboter mit allen Komponenten bestückt durch eine Umgebung gesteuert wird. Dies geschieht über WLAN von einem leistungsstärkeren PC mit einem 1,6 GHz Prozessor. Auf diesem PC laufen die Software zur Visualisierung, die Pfadplanung und die Exploration. Auf dem Roboter läuft der Treiber für den Laser, welcher dauerhaft die Laserdaten an ein Topic in ROS sendet. Zudem läuft noch ein Keylistener, der eigens entwickelt wurde, um die Signale, die an den Roboter geschickt werden in Befehle für die Steuerung umzuwandeln.
Die gewählte Umgebung war eine Wohnung mit verschiedenen Räumen, leichten Höhenunterschieden und unterschiedlichen Bodenbeschaffenheiten. Die Höhenunterschiede waren bis zu 3 cm gut von dem Roboter begehbar. Auch die verschiedenen Bodenbeschaffenheiten, wie z. B. glatter Laminatboden oder Teppich waren gut begehbar. Dies beeinflusste in keiner Weise das Ergebnis (Bild 5.8).

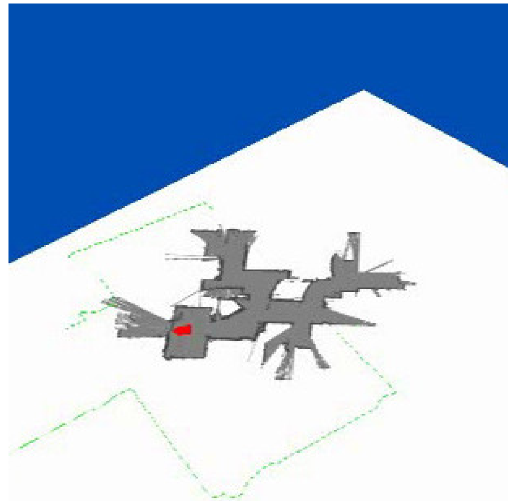

Abbildung 5.8: Karte der Wohnung, die als Testumgebung diente. Kleinere Objekte, wie Stühle oder Tischbeine wurden auch als Hindernisse erkannt und können somit umgangen werden.

Die Test ergaben, dass auch ruckartige oder schnelle Bewegungen des Roboters wenig Einfluss auf die Qualität der Kartenerstellung haben. Dabei war auch die ermittelte Pose des Roboters durch Hector SLAM sehr genau, sodass auch in engen Umgebungen eine autonome Bewegung möglich wäre.

Kapitel 6

Fazit und Ausblick

In dieser Arbeit wurden zwei effektive Lösungen des SLAM Problems vorgestellt. GMapping ist dabei ein Verfahren, welches auf der Idee von FastSLAM 2 basiert. Es ist mathematisch durchdacht und wird oft in der Praxis verwendet. Jedoch benötigt dieses Verfahren Kontrolldaten, welche z. B. durch Odometrie bereitgestellt werden. Hector SLAM hingegen basiert auf einer einfachen Idee und trennt das Verfahren der Posebestimmung und des Kartenerstellens voneinander. Herauskristallisiert hat sich dabei, dass Hector SLAM für den verwendeten Roboter das beste Verfahren darstellte, da es keine Odometrie benötigt und dennoch gute Ergebnisse liefert. Es wurde gezeigt, dass dieses SLAM Verfahren in Echtzeit umsetzbar ist. Zudem wurde ein Algorithmus zur Exploration und Pfadplanung nach Wirth in das System integriert. Dieses Verfahren bietet den Vorteil, dass es für eine Exploration mehrerer Roboter geeignet ist. Dabei wird nur die globale Explorationskarte verwendet, auf die jeder Roboter Zugriff hat. In Hinblick auf spätere Anwendungen des Roboters wurde dieses Verfahren der Exploration gewählt.

Später sollen noch Verbesserungen in der Pfadplanung, sowie eine Optimierung der Exploration stattfinden. Es wird ein schnelleres Verfahren zur letztlichen Pfadsuche und eine geschicktere Implementierung der Distanzkarte gewählt. Dabei soll die Pfadsuche auf mehr-leveligen Karten arbeiten können. Zudem sollen auf dem Roboter weitere Sensoren, wie z. B. Ultraschallsensoren und ein Beschleunigungssensor integriert werden. Dadurch sollen genauere Ergebnisse erzielt und eine horizontale Lage der Sensoren garantiert werden. Es sollten auch noch weitere Tests folgen, in denen sich der Roboter autonom durch seine Umgebung bewegt. jetzige Versuche sind immer mit einer manuelle Steuerung durchgeführt wurden. Zu diesem Zweck wird die Pfadplanung noch dahingehend erweitert, dass sie die Möglichkeit besitzt, den Roboter selbst zu steuern. Dies wurde in dieser Arbeit nicht mehr vorgestellt, da noch einige Praxistests fehlen und eine sichere Exploration noch nicht gewährleistet werden konnte.

Die Arbeit hat jedoch gezeigt, dass der Roboter die Möglichkeit besitzt, schwieriges Gelände zu bewältigen und dabei eine Karte der Umgebung zu erstellen. Zudem war er in der Lage naheliegendes unbekanntes Gebiet zu erkennen und auch den optimalen Weg zu errechnen, um dieses zu erkunden. Einige Nachteile gab es in der Rechenleistung des Raspberry Pi bzw. in der Komplexität der Software, da nicht das komplette SLAM Verfahren auf diesem umgesetzt werden konnte. Die größeren Teilmodule mussten ausgelagert werden und das Raspberry Pi war als Schnittstelle zur Datenlieferung und Steuerung des Roboters zuständig.

Anhang A

Schaltplan und Belegungsplan der Platine

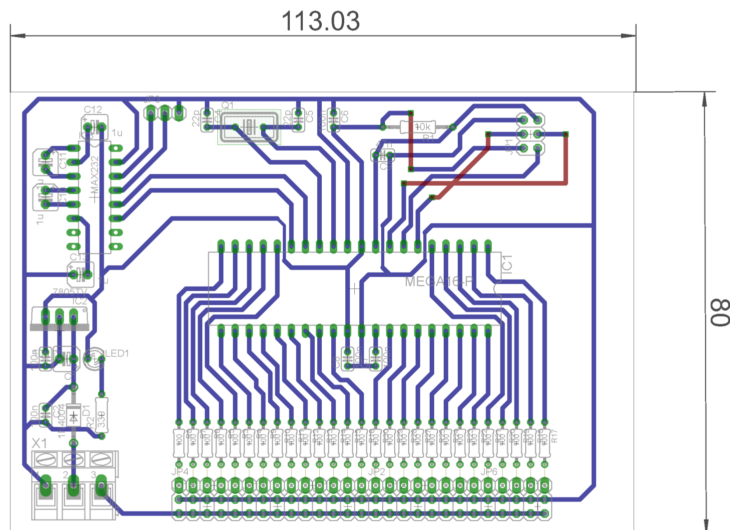

Abbildung A.1: Belegungsplan der Platine, die auf dem Roboter verwendet wurde.

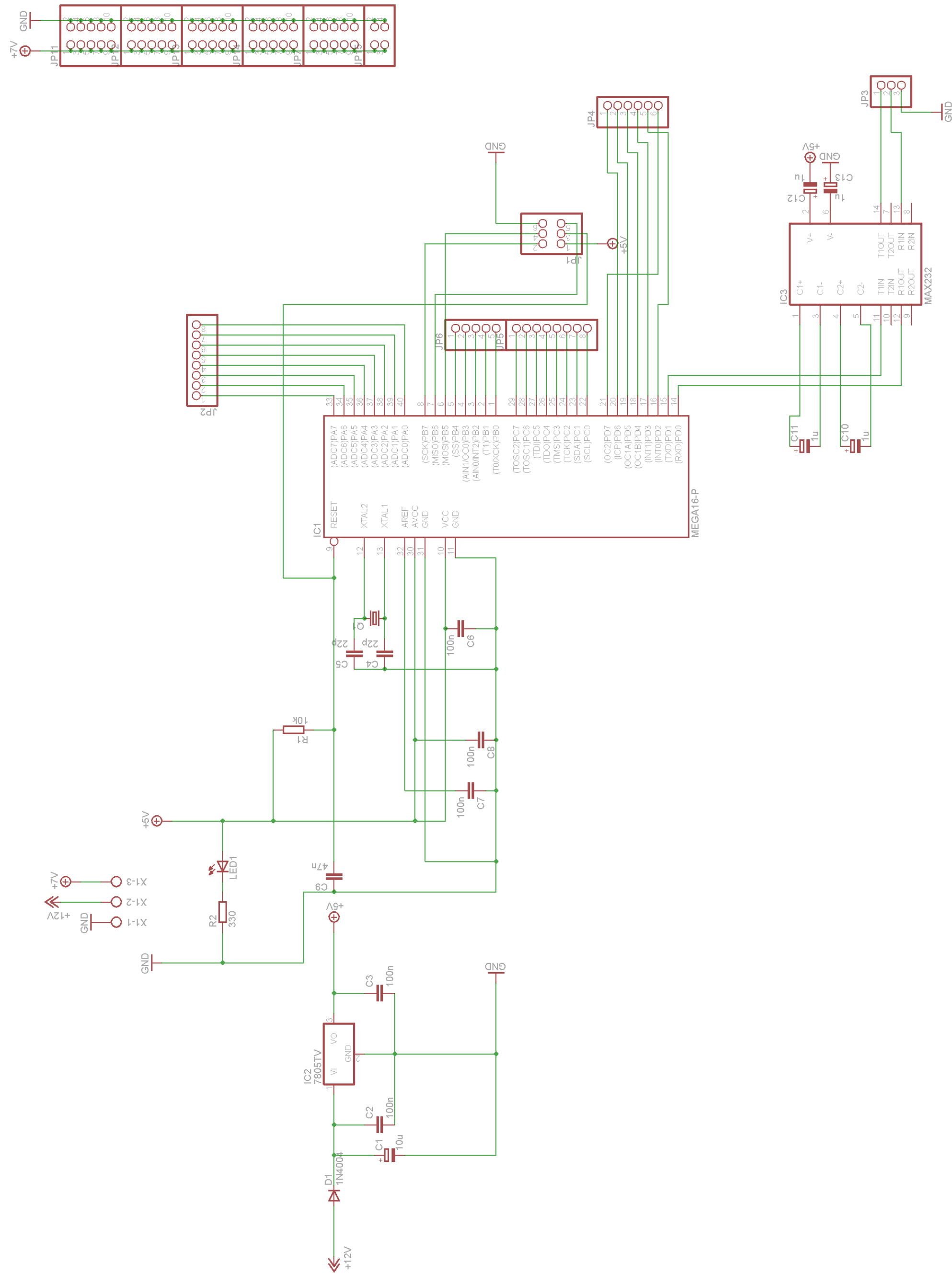

Abbildung A.2: Schaltplan der Platine des Roboters.

Literaturverzeichnis

[Abb] Abbeel, Pieter: *gMapping*. Website. http://www.cs.berkeley.edu/~pabbeel/cs287-fa12/slides/gMapping.pdf

[DGA00] Doucet, Arnaud ; Godsill, Simon ; Andrieu, Christophe: On Sequential Monte Carlo Sampling Methods for Bayesian Filtering. In: *STATISTICS AND COMPUTING* (2000)

[FH96] Friedman, Nir ; Halpern, Joseph Y.: A qualitative Markov assumption and its implications for belief change. In: *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*, 1996

[FTBD01] Fox, Dieter ; Thrun, Sebastian ; Burgard, Wolfram ; Dellaert, Frank: *Particle Filters for Mobile Robot Localization.* 2001

[GBL02] Gonzalez-Banos, Hector H. ; Latombe, Jean-Claude: Navigation Strategies for Exploring Indoor Environments. In: *The International Journal of Robotics Research* (2002)

[GSB05] Grisetti, G. ; Stachniss, C. ; Burgard, W.: Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling, 2005

[HBFT03] Hähnel, Dirk ; Burgard, Wolfram ; Fox, Dieter ; Thrun, Sebastian: A highly efficient FastSLAM algorithm for generating cyclic maps of large-scale environments from raw laser range measurements. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2003

[JB86] Jarvis, RA ; Byrne, JC: Robot navigation: Touching, seeing and knowing. In: *Proceedings of the 1st Australian Conference on Artificial Intelligence*, 1986

[Joh73] Johnson, Donald B.: A Note on Dijkstra's Shortest Path Algorithm. In: *J. ACM* (1973)

[KMSK11] Kohlbrecher, S. ; Meyer, J. ; Stryk, O. von ; Klingauf, U.: A Flexible and Scalable SLAM System with Full 3D Motion Estimation. In: *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2011

[Kra12] Krayer, Bastian: *Meijster Distanztransformation*, Universität Koblenz, Diplomarbeit, 2012

[Liu96] Liu, JunS.: Metropolized independent sampling with comparisons to rejection sampling and importance sampling. In: *Statistics and Computing* (1996)

[Mü07] Müller, Simon: *Monte Carlo-Methoden - Angewandt in der Roboterlokalisierung*, Universität Stuttgart, Diplomarbeit, 2007

[Men07] Mengelkoch, Marco: *Implementieren des FastSLAM Algorithmus zur Kartenerstellung in Echtzeit*, Universität Koblenz, Diplomarbeit, 2007

[Mor88] Moravec, H. P.: Sensor fusion in certainty grids for mobile robots. In: *AI Magazine* (1988)

[MRH02] Meijster, Arnold ; Roerdink, Jos B. ; Hesselink, Wim H.: A general algorithm for computing distance transforms in linear time. In: *Mathematical Morphology and its applications to image and signal processing* (2002)

[MTKW02] Montemerlo, Michael ; Thrun, Sebastian ; Koller, Daphne ; Wegbreit, Ben: FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In: *AAAI/IAAI*, 2002

[MTKW03] Montemerlo, M. ; Thrun, S. ; Koller, D. ; Wegbreit, B.: FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping that Provably Converges. In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 2003

[Mur99] Murphy, Kevin: Bayesian Map Learning in Dynamic Environments. In: *In Neural Info. Proc. Systems (NIPS*, 1999

[RL01] RUSINKIEWICZ, Szymon ; LEVOY, Marc: Efficient Variants of the ICP Algorithm. In: *3rd International Conference on 3D Digital Imaging and Modeling (3DIM).* Quebec City, Canada : IEEE Computer Society, 2001, S. 145–152

[ros] `http://www.ros.org`

[TBF05] THRUN, Sebastian ; BURGARD, Wolfram ; FOX, Dieter: *Probabilistic Robotics.* MIT Press, 2005

[WB95] WELCH, Greg ; BISHOP, Gary: An Introduction to the Kalman Filter. University of North Carolina at Chapel Hill, 1995. – Forschungsbericht

[WB10] WOLFRAM BURGARD, Maren Bennewitz Giorgio Grisetti Kai A. Cyrill Stachniss S. Cyrill Stachniss: *EKF Localization.* Website. `http://ais.informatik.uni-freiburg.de/teaching/ss10/robotics/slides/13-ekf-localization.pdf`. Version: 2010

[Wir07] WIRTH, Stephan: *Autonome gründliche Exploration unbekannter Innenräume mit dem mobilen Roboter „Robbie"*, Universität Koblenz, Diplomarbeit, 2007

[WP07] WIRTH, Stephan ; PELLENZ, Johannes: Exploration Transform: A stable exploring algorithm for robots in rescue environments. In: *Workshop on Safety, Security, and Rescue Robotics, http://sied.dis.uniroma1.it/ssrr07/* (2007), S. 1–5

[YSA98] YAMAUCHI, Brian ; SCHULTZ, Alan ; ADAMS, William: Mobile robot exploration and map-building with continuous localization. In: *In Proceedings of the 1998 IEEE/RSJ International Conference on Robotics and Automation*, 1998

[ZCS91] ZELINSKY, A. ; COMPUTER SCIENCE, University of Wollongong. Dept. o.: *Environment Exploration and Path Planning Algorithms for Mobile Robot Navigation Using Sonar.* University of Wollongong, 1991