Gymnasium Damme

Nordhofe 1
49401 Damme

Jugend Forscht 2015

Technik

Implementation, Optimierung und Erweiterung eines SLAM Algorithmus auf einem selbst entwickelten Rettungsroboter

Autor:
Jan Blumenkamp

Betreuung:
Herr Rohe

24. März 2015

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ARM Acorn Risc Machine

EKF Extended Kalman Filter

GUI Graphical User Interface

IMU Inertial Measurement Unit

IR Infrarot

LIDAR Light Detection and Ranging

LSB Least Significant Byte

MSB Most Significant Byte

SLAM Simultaneous Localization And Mapping

SPLAM Simultaneous Planning, Localization And Mapping

SRAM Static Random-Access Memory

UART Universal Asynchronous Receiver Transmitter

1 Kurzfassung

Rettungsroboter werden dazu eingesetzt, den Rettungskräften einen ersten Überblick über ein Gebäude oder ein Gelände zu verschaffen, bei dem bspw. nach einer Naturkatastrophe ein Betreten zu riskant für die Rettungskräfte ist. Als bekanntes Beispiel ist dabei der GAU von Fukushima (2011) zu nennen. Dabei ist es besonders wichtig, dass eine Karte der Umgebung erstellt wird, auf die die Rettungskräfte zugreifen können.

Es wird ein vollwertiger Simultaneous Localization And Mapping (SLAM) (Stachniss 2006, S. 9) Algorithmus, der die Informationen über die Umgebung über ein low-cost Lidar bezieht, auf einem zuvor erfolgreich im RoboCup Junior Rescue B Wettbewerb eingesetzten Rettungsroboter implementiert, auf dem der Algorithmus intensiv getestet und optimiert wird. Anschließend wird der Algorithmus um ein PC User Interface erweitert, zu dem die erstellte Karte drahtlos gesendet wird. Über das User Interface können Wegpunkte festgelegt werden, die vom Roboter der Reihe nach besucht werden sollen. Der Rettungsroboter ist dazu in der Lage, ein Gelände halbautonom (d.h. basierend auf den vom Benutzer festgelegten Wegpunkten) zu erkunden, eine Karte zu erstellen und drahtlos und in Echtzeit zu einem Computer zu senden und nach Fertigstellung der Karte zum Ausgangspunkt zurückzukehren. Alle Berechnungen und die Kartenerstellung finden dabei auf dem Roboter statt, sodass dieser auch ohne eine Verbindung zum Computer arbeiten kann.

Dieses halbautomatische Verfahren der Navigation wird auch bei ferngesteuerten Rovern in der Raumfahrt eingesetzt, wo aufgrund der Laufzeit des Signals und aus praktischen Gründen eine direkte Steuerung nicht möglich ist. (vgl. JPL 2013)

2 Einleitung

2.1 Der Roboter

Als Grundlage dient ein selbst entwickelter Roboter, der zuvor im RoboCup Junior Rescue B Wettbewerb eingesetzt wurde (siehe Abbildung 13a).

Beim RoboCup Junior Rescue B Wettbewerb soll ein autonomer Roboter in einem für Rettungskräfte unzugänglichen Gebäude nach verschütteten Opfern suchen. Dieses Gebäude wird in diesem Wettbewerb für Schüler vereinfacht dargestellt durch ein rechtwinkliges, modular aufgebautes Labyrinth mit Grundfliesen mit einer Größe von $30cm \cdot 30cm$. Die verschütteten Opfer werden durch auf Körpertemperatur aufgeheizte elektrische Heizplatten dargestellt. Das Labyrinth ist dem Roboter unbekannt, im besten Fall erstellt der Roboter daher eine Karte, mit deren Hilfe der Roboter systematisch nach Opfern suchen kann. Lokalisierten Opfern wird automatisch ein Rettungspaket abgeworfen. Wenn der Roboter das gesamte Labyrinth nach Opfern durchsucht hat, soll er zum Eingang des Labyrinths bzw. des Gebäudes zurückkehren. (vgl. Bonilla u. a. 2013, S. 2)

Vom im Rahmen des Wettbewerbs entwickelten Roboter werden die Hardware und Teile der Software verwendet, da sich diese Forschungsarbeit im Wesentlichen mit der Implementation des SLAM Algorithmus beschäftigen soll. Die Hardware des Roboters zeichnet sich durch ihre Robustheit sowie durch ihre im Wettbewerb mehrfach unter Beweis gestellte Zuverlässigkeit aus.

Der Roboter verfügt über vier Räder, die über je einen Getriebemotor aufweisen. Die Motoren auf einer Seite sind dabei hintereinander angeordnet und elektrisch parallel geschaltet. Es werden lediglich die Odometer, also Sensoren, die die gefahrene Distanz ausschließlich darüber messen, wie weit sich z.B. ein Rad[1] gedreht hat, der jeweils vorderen Motor einer Seite ausgewertet und die Motoren ggf. auf eine vorgegebene Geschwindigkeit geregelt, sodass der jeweils hintere Motor auf einer Seite als Verstärkung des vorderen Motors dient, was besonders bei möglichen auf dem Boden liegenden Hindernissen und auf Steigungen hilfreich ist. Die Geschwindigkeit und Drehrichtung beider Motoren auf der linken Seite kann so unabhängig von der Geschwindigkeit und Drehrichtung beider Motoren auf der rechten Seite geregelt werden. Drehen sich die Räder beider Seiten gleich schnell, fährt der Roboter vorwärts, dreht sich einer der beiden Motorenpaare langsamer, fährt der Roboter in Richtung des langsamer drehenden Motorpaars und drehen sich beide Motorpaare in entgegengesetze Richtungen, dreht der Roboter sich auf der Stelle.

Es ist möglich, durch die zusätzlich entwickelte Hardware (siehe Kapitel 2.4.1) die Roboterplattform zu steuern, also die Motorgeschwindigkeiten des sogenannten differentiellen Antriebs, dem gleichen Antriebskonzept von Kettenfahrzeugen, zu setzen, sowie auf die vorhandenen Sensoren zuzugreifen, wie z.B. Infrarot (IR) Entfernungssensoren oder die Odometer der Motoren, welche für die Implementation des SLAM Algorithmus elementar wichtig sind. (vgl. Oubbati 2012b)

2.2 Motivation

Beim RoboCup Junior Rescue B Wettbewerb spielt die Erstellung einer Karte eine wesentliche Rolle. Die Notwendigkeit der Erstellung ist zwar nicht in den Regeln vorgegeben, allerdings sind viele zu identifizierende Opfer nur über ein systematisches Absuchen das Labyrinths mithilfe einer Karte auffindbar. Der einfache Aufbau des Labyrinths (Rechwinkligkeit, $30cm$ große Einheiten) erlaubt es, eine sehr grobe Karte zu erstellen. Eine Karte mit hoher Auflösung ist nicht notwendig. Die Art und Weise der Fortbewegung des Roboters ist von der Karte abhängig - es muss sich immer in 30cm Einheiten und 90° Drehungen bewegt werden (kleinere Schritte sind aufgrund der niedrigen Auflösung der Karte nicht sinnvoll und nur schwierig umzusetzen). Wenn eine Fahranweisung beendet ist, wird die Karte aktualisiert. Dabei muss sich darauf verlassen werden können, dass der Roboter sich tatsächlich $30cm$ fortbewegt oder um 90° gedreht hat, sich also noch im Raster der Karte befindet. Über IR Entfernungssensoren ist dies auch recht zuverlässig möglich, ein Fehler ist aber trotzdem nicht auszuschließen. Geröll auf dem Boden, dargestellt durch Schaschlikspieße, oder andere Hindernisse, die in den Parcours eingebaut werden können und deren Größe und Form nicht genau definiert ist, können diese erreichte Zuverlässigkeit zerstören. Ein Hängenbleiben des Roboters an diesen Hindernissen muss zuverlässig erkannt werden, was ohne eine Vielzahl von Sensoren nicht (zuverlässig) möglich ist. Ein gängiger Ansatz, um diese Problematik zu lösen, ist SLAM.

[1] Abhängig von der Bauweise des Roboters; auch fliegende Roboter können via SLAM eine Karte der Umgebung anfertigen (vgl. Engel u. a. 2013, S. 1).

2.3 Lösungsansatz SLAM

Der RoboCup Wettbewerb existiert nicht nur für Schüler, auch Studenten entwickeln Roboter, die grundsätzlich die gleiche Aufgabenstellung wie die Schüler bewältigen müssen, insgesamt ist dort das dargestellte Gebäude allerdings wesentlich komplexer (größer, nicht nur rechtwinklig...). Hier ist die Erstellung einer Karte auch in den Regeln vorgeschrieben (vgl. robocup.org 2014). Aufgrund der Komplexität der Aufgabenstellung ist es für diese Roboter schon lange elementar wichtig, die Orientierung nicht zu verlieren. Eine gängige Methode zur Selbstlokalisierung innerhalb der selbst erstellten Karte ist SLAM. SLAM ist weniger ein Algorithmus als mehr eine Sammlung von Algorithmen und kann daher eher als Verfahren bezeichnet werden. (vgl. Stachniss 2006, S. 19 f.)

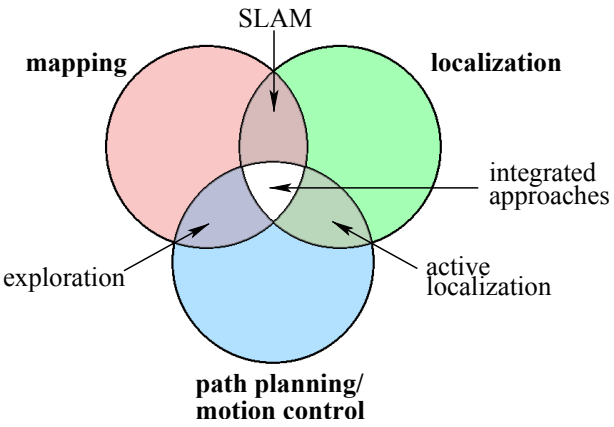


Abbildung 1: Differenzierung der Teilgebiete die nötig sind, um in einem Roboter ein genaues Abbild der Umgebung zu erzeugen. (Stachniss 2006, S. 20)

Damit ein Roboter autonom ein unbekanntes Gebiet erkunden kann, sind im Wesentlichen drei Schritte notwendig, und zwar die Erstellung einer Karte der Umgebung, die parallele Selbstlokalisierung in dieser Karte und die Planung eines Pfades, um gezielt an bestimmte Orte zu navigieren, im einfachsten Fall um die Karte zu vervollständigen. SLAM kombiniert dabei die Erstellung der Karte und die Lokalisierung des Roboters in dieser erstellten Karte. Die Planung des Pfades erfolgt unabhängig davon, aber basierend auf die erstellte Karte. Dieser Prozess wird „integrated approaches“ (ebd., S. 20) oder auch Simultaneous Planning, Localization And Mapping (SPLAM) genannt (siehe Abbildung 1). Da für eine akkurate Lokalisierung eine gute Karte und zum zuverlässigen Erstellen der Karte eine akkurate Position des Roboters benötigt wird, wird im Zusammenhang von SLAM auch auf das Henne-Ei-Problem verwiesen. (vgl. ebd., S. 20)

Da es sich bei SLAM um ein Verfahren handelt, gibt es verschiedene Ansätze. Grundsätzlich werden aber immer Filter benötigt, die die rohen Sensordaten interpretieren, um diese Sensordaten in die Karte zu integrieren, z.B. Extended Kalman Filter (EKF) oder, womit sich diese Forschungsarbeit im Weiteren beschäftigt, Partikelfilter. (vgl. Steux u. a. 2010, S. 1)

Zu Beginn hat der Roboter keine Karte und somit auch keine Informationen über die Umgebung und beginnt in einem Koordinatensystem beim Zeichnen der Karte im Ursprung. Nun müssen Informationen über die Umgebung, in der sich der Roboter befindet, in das Koordinatensystem übertragen werden. Diese Informationen stammen i.d.R. bei komplexeren Robotern von einem LIDAR[2]. Der Roboter hat nun erste Informationen über die Umgebung gespeichert. Für den EKF würden als nächstes markante sogenannte „Landmarks“ (Riisgaard u. a. 2005, S. 16) aus dem Laserscan gefiltert werden. Landmarks sind einfach wiedererkannbare Eigenschaften der Umgebung (z.B. lange, ebene Wände, also Linien, Ecken, oder sonstige hervorstechende Merkmale). Vergleichbar ist die Orientierung eines Menschen in einer Stadt: Man erkennt markante Hochhäuser, Brücken o.Ä. eher wieder, als eine normale Hauswand. Diese Merkmale der Umgebung werden in einer Datenbank gespeichert. (vgl. ebd., S. 10 ff.)

Unabhängig vom verwendeten Filter fährt der Roboter in eine gewünschte Richtung, z.B. in der Informationen über die Umgebung fehlen (wenn das Ziel ist, die Karte zu vervollständigen). Die Navigation bzw. Erkundung ist nicht Teil von SLAM, hier greifen wieder die „integrated approaches“ (Stachniss 2006, S. 20). Während der Fahrt wird über Odometer die neue Position des Roboters in der Karte grob geschätzt und aktualisiert. Das ist recht ungenau, da Räder i.d.R. über Schlupf aufweisen. Dieser Fehler würde sich im Verlauf der Kartenerstellung summieren und die Abweichung des Roboters in der Karte von der Ist-Position des Roboters somit immer größer und die Karte immer ungenauer werden. An dieser Stelle werden die gespeicherten Merkmale der Umgebung oder einfach nur die bereits erstellte Karte für den Roboter interessant. Der Roboter führt erneut einen Scan der Umgebung durch und vergleicht diesen mit Hilfe des Filters mit den in der Datenbank in der Nähe des Roboters gespeicherten Merkmalen. Nun wird eine eventuelle Fehlposition des Roboters in der Karte (also eine Abweichung von der Ist-Position des Roboters in der realen Umgebung von der vermuteten Position des Roboters in der Karte) korrigiert. Der gesamte Prozess beginnt nun von vorne. Auf diese Art und Weise lässt sich sehr präzise eine Karte der Umgebung erstellen. (vgl. Riisgaard u. a. 2005, S. 10 ff.)

[2]Laserscanner (bei Verweis auf LIDAR ist i.d.R. nicht das Verfahren sondern der Scanner gemeint), dieser scant meistens mithilfe eines Lasers (vgl. Fujii 2005, S. 1 f.) oder über (3D-) Kameras (vgl. Engel u. a. 2013, S. 1) ein mehrdimensionales Bild der Umgebung.

Diese akkurate Karte kann nur mithilfe guter Sensoren erstellt werden. Die im Roboter verbauten IR Entfernungssensoren sind nicht gut genug für diese Anwendung. Zum einen ist die Qualität nicht hoch genug (Rauschen des analogen Ausgangssignals), zum anderen weist der Messbereich mit etwa 1m (SHARP 2006) über keine genügend große Spannweite und Reichweite auf. Weiterhin ist die Anzahl der Sensoren mit insgesamt zehn Stück zu klein. Optimal für SLAM ist ein LIDAR geeignet. Im Folgenden wird auf die Verwendung eines LIDARs aus einem handelsüblichen Staubsaugerroboter auf der Roboterplattform eingegangen.

2.4 Erweiterung des Grundgerüsts

2.4.1 Mikrocontroller

Dem Roboter liegt ein auf $16MHz$ getakteter Atmel 8bit AVR ATmega2560 Mikrocontroller mit $256kB$ programmierbaren Flashspeicher und $8kB$ Static Random-Access Memory (SRAM) (vgl. Atmel 2011) zu Grunde. Für die Aufgabenstellung des RoboCups ist dieser Controller ausreichend, dennoch stößt er schon hier an seine Grenzen, sowohl in Sachen Geschwindigkeit, als auch in Sachen Arbeitsspeicher. Eine Karte mit einer Größe von $1m \cdot 1m$ und einer Auflösung von $1cm$ würde schon einen Speicher von $100 \cdot 100B = 10kB$ in Anspruch nehmen. Deshalb muss ein leistungsfähigerer Controller verwendet werden. Die Entscheidung fiel auf ein STM32F4 Discovery Entwicklungsboard der Firma STMicroelectronics. Auf dem preisgünstigen Entwicklungsboard befindet sich ein leistungsstarker 32bit Acorn Risc Machine (ARM) Controller des Typs STM32F417. In dieser Anwendung wurde der Mikrocontroller auf $168MHz$, die höchstmögliche Taktfrequenz, getaktet. Es stehen $192kB$ SRAM zur Verfügung, was für ausgedehnte Tests mit relativ großen Karten ausreichend ist. (vgl. ST 2014b)

2.4.2 LIDAR

Als wichtigstes Bauteil ist das LIDAR zu nennen. In Kapitel 2.3 wurde auf die Einsatzmöglichkeiten von SLAM in Bereichen der Forschung und in Rettungsszenarien eingegangen. Es gibt allerdings auch noch praxisnähere Einsatzgebiete. Staubsauerroboter sind seit einiger Zeit aufgrund ihrer sinkenden Kosten einer immer größeren Masse an Menschen zugänglich. Günstigere Modelle fahren nach dem Zufallsprinzip einen Raum ab, im Gegensatz zu hochwertigeren Modellen, die eine Karte des Raumes erstellen, dort ihre Ladestation vermerken, den Raum (oder mehrere Räume) systematisch abfahren, bei Bedarf zur Ladestation zurückkehren und dann an der Stelle mit der Reinigung fortfahren, an der sie unterbrochen wurde. Für die Orientierung im Raum wird ein LIDAR verwendet. (vgl. neatorobotics.com 2014)

Das im Roboter verbaute LIDAR kann sehr preiswert als Ersatzteil erworben werden. Offizielle Datenblätter stehen nicht zur Verfügung, das Protokoll des Sensors wurde jedoch schon vor einiger Zeit von einer Community entschlüsselt und die Daten im Internet zur Verfügung gestellt. Das LIDAR weist über sechs Anschlüsse auf: Die Versorgungsspannung (GND und $+3{,}3V$) für das Lidar, die Versorgungsspannung für den Motor und die Datenleitungen TX und RX der seriellen Universal Asynchronous Receiver Transmitter (UART) Schnittstelle. Die Kommunikationsgeschwindgkeit liegt bei $115200b/s$. Die Maximalreichweite beträgt ca. $6m$, die Minimalreichweite $15cm$. Der Sensorkopf mit dem Laser dreht sich mit einer Sollfrequenz von $5Hz$, also 300 mal pro Minute, um sich selbst, wodurch der Laser einen Bereich von 360° mit einer Auflösung von 1° abdeckt. Die Daten werden dabei aus dem LIDAR gestreamt, sobald sich der Sensorkopf dreht. Das Protokoll bzw. ein Paket daraus sieht dabei wie folgt aus:

<S><I>[SP][D0][D1][D2][D3][CHK]

- S: Start: 1 Byte: 0xFA
- I: Index: 1 Byte
- SP: Speed: 2 Bytes (Least Significant Byte (LSB)/Most Significant Byte (MSB))
- Dx: 1 Entfernungsmessung: 4 Bytes
  - <distance LSB>
  - <{invalid data flag}{strength warning flag}{distance MSB}>
  - <signal strength LSB>
  - <signal strength MSB>

- CHK: 2 Bytes (LSB/MSB)

Ein Datenpaket Dx enthält also vier Bytes, in denen Daten über eine Entfernungsmessung gespeichert sind. Ein Paket enthält vier Entfernungsmessungen, weshalb für einen 360° Scan $90 \cdot (6B + 4 \cdot 4B) = 1980B$ an Daten gesendet werden. (vgl. xv11hacking.wikispaces.com 2014)

Für die Kommunikation mit dem Computer ist ein Bluetoothmodul des Typs $HC-06$ vorgesehen. Hierbei handelt es sich um günstige Massenware aus dem Ausland, weshalb nur mangelhafte Dokumentation zur Verfügung steht. Das Modul verfügt über vier Anschlüsse: Die Versorgungsspannung (GND und $+3{,}3V$) und die Datenleitungen TX und RX der seriellen UART Schnittstelle. Das Modul emuliert dabei eine serielle Verbindung mit dem Computer. (vgl. mikrokopter.de 2014)

Die Baudrate ist auf vorgegebene Stufen einstellbar, in Tests hat sich $460800b/s$ als guter Kompromiss zwischen Geschwindigkeit und Datenverlust herausgestellt.

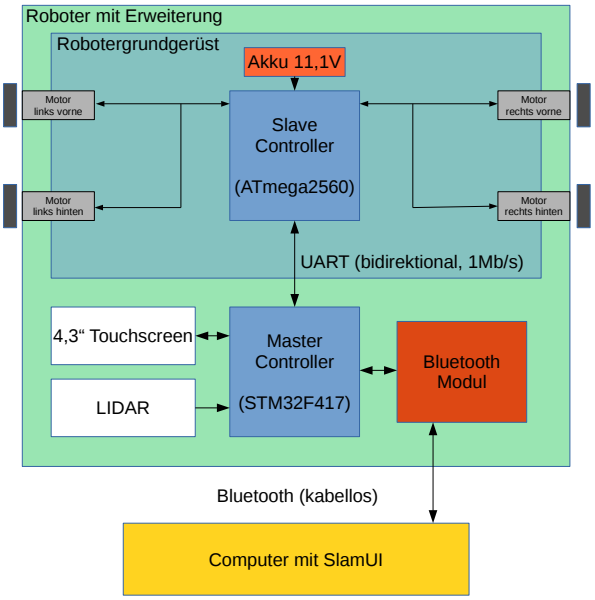


Abbildung 2: Blockschaltbild des gesamten in dieser Arbeit behandelten Systems

Der Roboter mit Erweiterung ist in Abbildung 13b zu sehen.

2.4.3 Kommunikation

Die Kommunikation zwischen Master (ARM) und Slave (AVR) läuft über eine serielle UART Schnittstelle über ein eigens für diesen Anwendungszweck entwickeltes Protokoll mit einer Kommunikationsgeschwindigkeit von $1Mb/s$ ab. Das Protokoll sieht wie folgt aus:

<S><R><N>(N*<D>)[CHK]

- S: Start: 1 Byte: 0xAB
- R: Register/Adresse, Read/Write Flag: 1 Byte
- N: Anzahl der abzufragenden Bytes
- D: Datenbyte: 1 Byte
- CHK: 2 Bytes (LSB/MSB)

Es gibt eine bestimmte Anzahl an 1 Byte breiten Registern auf dem Slave, die vom Master manipuliert oder abgerufen werden können. Eine Anfrage beginnt immer mit dem Startbyte. Als nächstes folgt das Startregister, welches abgerufen oder manipuliert werden soll, danach die Länge der Nachricht (bis zu 127 Bytes) und die Art des Zugriffs (schreiben oder lesen). Wenn die Register ausgelesen werden sollen, werden keine Datenbytes gesendet (N ist dann die Anzahl der Register, die ausgelesen werden sollen), ansonsten folgen N Datenbytes, die mit R als Startregister in die folgenden Register des Slaves geschrieben

werden können. Der Slave antwortet grundsätzlich mit dem selben Protokoll: Bei einer Abfrage muss eine Antwort kommen (der Slave sendet dann eine Schreibanfrage an den Master), bei einer Schreibanfrage vom Master wird eine Bestätigunsnachricht vom Slave gesendet, wenn der Schreibzugriff erfolgreich war, ansonsten wird eine Fehlernachricht gesendet. Der Slave kann auch von sich aus Nachrichten senden, ohne eine Anfrage vom Master vorliegen zu haben. Dies kann nützlich sein, wenn z.B. Tasten auf der Roboterplattform eine Aktion auf der Erweiterungsplattform (dem Master) hervorrufen sollen. Die Tasten müssen dann nicht ständig vom Master abgefragt werden, was eine Menge an Datenverkehr erspart.

Diese bidirektionale Kommunikation hat den Vorteil, dass Daten niemals verloren gehen (wenn eine Fehlernachricht oder gar keine Nachricht vom Slave auf einen Schreibzugriff zurückkommt, wird das Paket einfach erneut gesendet, wenn das nicht funktioniert, muss ein Fehler in der Verbindung vorliegen). Die Checksumme, die eine Addition aller Werte darstellt, sichert das Protokoll zusätzlich ab. Diese Sicherheit ist nötig, damit gerade in Testphasen kein Schaden auf der Roboterplattform entstehen kann, wenn z.B. durch eine fehlerhaft übertragende Geschwindigkeit der Roboter gegen andere Gegenstände fährt. Wenn über einen bestimmten kurzen Zeitraum keine Nachrichten vom Master gesendet wurden, werden die Motoren automatisch abgeschaltet.

2.4.4 Software

Sowohl der Master, als auch der Slave sind vollständig in der Programmiersprache C programmiert. Auf dem Master läuft das freie Echtzeitbetriebssystem „FreeRTOS“ (freertos.org 2014a), für welches sich aufgrund seiner großen Verbreitung, einer Vielzahl von Beispielen und der guten Performance entschieden wurde. Ein Echtzeitbetriebssystem ist grundsätzlich nicht nötig, erleichtert aber das Taskmanagement des Prozessors besonders in einer so komplexen Anwendung erheblich.

2.4.5 GUI

Als GUI unmittelbar auf dem Roboter soll ein 4,3” großer Touchscreen dienen. Die hochauflösende Karte kann so direkt auf dem Roboter visualisiert werden. Der verwendete Mikrocontroller weist über genug Ressourcen für das Farbdisplay auf. Die GUI Bibliothek wurde eigens für diesen Anwendungszweck vom Autor entwickelt und implementiert. Ein Bildschirmfoto des GUI ist in Abbildung 14 zu sehen.

3 Implementation des SLAM Algorithmus

3.1 tinySLAM

Wie bereits in der Einleitung (Kapitel 2.3) erwähnt, ist SLAM auf verschiedene Art und Weisen lösbar. Es gibt sehr komplexe Implementationen, die entsprechend gut funktionieren, aber eine Menge Ressourcen benötigen. DP-SLAM, ein Algorithmus entwickelt 2003 an der Duke University, ist ein Beispiel dafür. Es wird nicht nur die Karte für die Umgebung erstellt, sondern es werden verschiedene Möglichkeiten für Roboterpositionen und Varianten der Karte gespeichert, was einen sehr großen Speicherplatzverbrauch mit sich zieht. (vgl. Eliazar u. a. 2003, S. 1)

Auf der Internetseite `openslam.org` werden verschiedene Lösungsansätze von SLAM zusammengefasst dargestellt (vgl. openslam.org 2014a). Viele der veröffentlichten Algorithmen sind sehr umfangreich, ähnlich wie DP-SLAM, teilweise noch komplexer. Auf dieser Internetseite wird auch auf den tinySLAM Algorithmus verwiesen. Dieser wird vor allem durch seine Kompaktheit beworben:

> „tinySLAM is Laser-SLAM algorithm which has been programmed in less than 200 lines of C-language code.“ (ebd.)

Auch die verwendete Programmiersprache, C, macht tinySLAM sehr interessant für die Verwendung auf dem vorhandenen Roboter, da dieser ebenfalls in C programmiert ist (siehe Kapitel 2.4.4). Leider ist der veröffentlichte Quellcode wenig bis gar nicht kommentiert, sodass es nahezu unmöglich ist, diesen nachzuvollziehen. Es gibt lediglich eine sechsseitige Dokumentation, in der der Algorithmus allerdings nicht als tinySLAM, sondern als „CoreSLAM“ (Steux u. a. 2010) bezeichnet wird. Diese Dokumentation hilft ansatzweise, dennoch wurde der Quellcode (Verfügbar auf openslam.org 2014b) zunächst hinsichtlich seiner Struktur analysiert (siehe Abbildung 5).

Ein ähnliches Ziel wie diese Arbeit verfolgt auch der BreezySLAM Ansatz. Dort soll ein einfacher SLAM Algorithmus in die Programmiersprache Python portiert werden, allerdings nicht mit dem Ziel, möglichst effizient und zuverlässig auf einem Echtzeitbetriebssystem zu laufen (vgl. Bajracharya 2014, S. 9 f.). In der Praxis wird dort ein Raspberry Pi Einplatinencomputer auf dem Neato-XV11 Roboter (aus dem auch das in dieser Arbeit verwendete LIDAR stammt) verwendet, der über wesentlich größere Ressourcen als der hier verwendete 32bit Controller verfügt (vgl. ebd., S. 31 ff.). Die auf github veröffentlichten Ergebnisse dieser Arbeit helfen erheblich beim Verständnis des Algorithmus (vgl. github.com 2015a).

TinySLAM liegt eine zweidimensionale Reihung im SRAM zu Grunde, welche die Karte mit der gegebenen Auflösung darstellt. Die Auflösung liegt normalerweise im Zentimeterbereich. Die Reihung beschreibt also ein kartesisches Koordinatensystem mit der gewünschten Auflösung. Jede Zelle der Karte kann dabei Werte zwischen 0 und 255 annehmen. Dieser Wert kann gewissermaßen als Wahrscheinlichkeit für ein Hindernis gesehen werden: 0 steht für kein Hindernis, 255 steht für ein Hindernis. 127 bedeutet, dass keine Informationen für diese Zelle zur Verfügung stehen, oder dass die Wahrscheinlichkeit für ein Hindernis an dieser Stelle genau so hoch ist wie für kein Hindernis. Wie in Abbildung 3 zu sehen ist, wird der Scan des LIDARs über eine Rampe in die Karte integriert: Angenommen, das LIDAR misst ein Hindernis in einer Entfernung von $1m$. Die polaren Koordinaten der Messung (gegeben ist ein Winkel, in dem ein Hindernis zum LIDAR steht bzw. der Winkel einer der 360 Messungen und eine Länge, nämlich die Entfernung zum Hindernis) werden in kartesische Koordinaten für die Karte umgerechnet. Diese Koordinaten stellen nun den Punkt des Hindernisses für die jeweilige Messung in Relation zur aktuellen Roboterposition dar. Für den Partikelfilter, auf den später noch eingegangen wird, darf allerdings nicht nur gespeichert werden, wo sich ein Hindernis befindet und wo nicht, hier wird die Wahrscheinlichkeit wichtig. Die Entfernungsmessungen weisen immer über ein gewisses Rauschen, eine Ungenauigkeit, auf. Diese Ungenauigkeit muss berücksichtigt werden, weshalb die Werte der Zellen in der Karte, die auf der Linie des Laserstrahls vor dem Hindernis liegen, langsam inkrementiert werden. Danach sinken die Werte der Zellen entlang

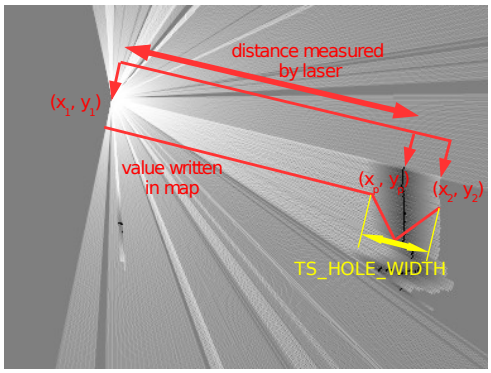


Abbildung 3: Integration des Scans in die tinySLAM Karte (ebd., S. 2). Die V-Form, die die Werte der Karte um das Hindernis bilden, ist klar zu erkennen.

des Strahls wieder, weil hier keine Daten über das Hindernis zur Verfügung stehen. Kern der Integration der in polarer Form vorliegenden Entfernungsmessungen in die kartesische Karte ist der Bresenham Algorithmus zur Rasterung von Linien. (vgl. Steux u. a. 2010, S. 1 f.)

Basierend auf diese Karte kann der Partikelfilter, implementiert als Monte-Carlo Algorithmus, den aktuellen Scan der Umgebung in die Karte einpassen[1]. Der Scan wird dafür erneut in kartesische Koordinaten umgewandelt und dann für mehrere, minimal (auch hier im Zentimeterbereich) von der über die Odometer berechneten Position abweichende Positionen in die Karte eingepasst. An dieser Stelle spielen die Wahrscheinlichkeiten eine elementare Rolle: Die Werte der Zellen, für die der aktuelle Umgebungsscan auch in der Karte ein Hindernis erkennt, werden addiert. Je höher die Summer der Werte ist, desto besser passt der Scan somit in die Karte. Auf diese Art und Weise kann eine fehlerhafte Position korrigiert werden. (vgl. ebd., S. 1 ff.)

Der gesamte Prozess ist in Abbildung 4 visualisiert. Die tatsächliche Implementation weist über eine wesentlich höhere Auflösung und mehr Entfernungsmessungen auf, die ermittelten Summen der Pixel liegen größenordnungsmäßig bei 100000.

Der Quellcode von tinySLAM gliedert sich in vier Sourcecodedateien:

- CoreSLAM_state.c: Oberste Instanz. Verantwortlich für die Organisation aller untergeordneter Prozesse der Bibliothek, organisiert/bereitet Daten für die Weiterverarbeitung auf.
- CoreSLAM_random.c: Generiert Zufallszahlen, beinhaltet eigentlichen Partikelfilter
- CoreSLAM_loop_closing.c: Ähnlich wie CoreSLAM_state.c
- CoreSLAM.c: Eigentlicher, mit 200 Zeilen beworbener Kern (vgl. openslam.org 2014a) von tinySLAM. Beinhaltet Funktionen zum Zeichnen/Aktualisieren der Karte und dem Vergleich des Laserscans mit der Karte.

Nach dieser ersten Analyse wird also klar, dass es sich bei dem Beispielcode um eine komplette Anwendung auf einem Roboter handelt. Lediglich das Hardwareinterface (Verbindung zum Lidar und zu den Odometern der Motoren) muss noch hinzugefügt sowie die Einstellungen in den Konfigurationsdateien angepasst werden.

3.2 Monte-Carlo Suche

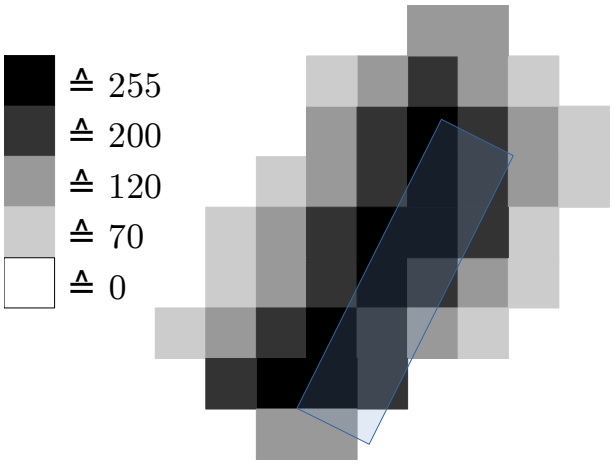
Der eigentliche Partikelfilter (*CoreSLAM_random.c; ts_monte_carlo_search*) besteht lediglich aus 50 Zeilen C-Quellcode. Ihm werden die Streuungen sowie die Anzahl der Versuche des zufälligen Ausprobierens neuer Positionen übergeben. Die Streuung stellt dabei den Bereich dar, um den, ausgehend von der aktuellen Roboterposition, neue Positionen auf ein besseres Passen in die Karte überprüft werden (sowohl die X/Y Komponente, als auch die Ausrichtung bzw. Orientierung des Roboters). Die Überprüfung folgt dabei in zwei Schritten: Im ersten Schritt (für das erste Drittel der Versuche) wird die volle Spannweite der übergebenen Streuung genutzt. Wie gut der aktuelle Scan des LIDARs in die Karte passt, wird dabei mithilfe der Funktion *ts_distance_scan_to_map* ermittelt (siehe auch Abbildung 5). Wenn der Scan für eine zufällige Position innerhalb der Spannweite der Streuung besser in die Karte passt, wird diese Position zwischengespeichert. Im zweiten Schritt, also für die letzten zwei Drittel der Versuche, wird für jede gefundene Position, die besser als die bis zu dem Zeitpunkt am besten passende Funktion in die Karte passt, die Streuung halbiert. Der Algorithmus grenzt den Bereich, in dem sich der Roboter befinden muss, also immer weiter ein, was die Lokalisierung sehr effektiv macht.

3.3 Minimierung des Algorithmus

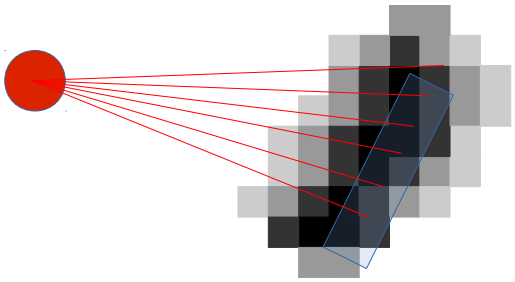
Der eigentliche tinySLAM Algorithmus umfasst lediglich die Datei *CoreSLAM.c*. Die hier enthaltene Funktion ts_map_update und ihre Unterfunktionen werden im Wesentlichen übernommen.

Wichtig ist außerdem die Datei *CoreSLAM_random.c*, in der sich die Funktionen für den Partikelfilter befinden. Wie in Kapitel 3.1 erläutert, werden zum Ermitteln der exakten Position verschiedene Positionen um der mithilfe der Odometer berechneten Position ausprobiert. Der Zufallsgenerator dafür ist in dieser Datei enthalten. Die standard C-Bibliothek verfügt allerdings bereits über einen gut genug funktionierenden Zufallsgenerator, weshalb der tinySLAM Zufallsgenerator nicht benötigt wird.

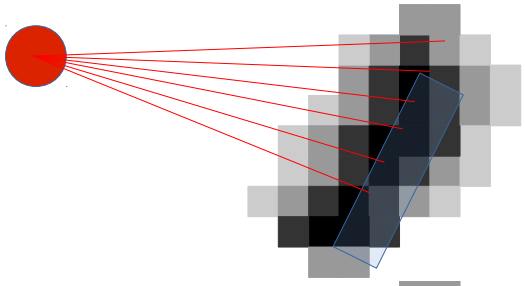
[1] Dieser Schritt ist nicht mehr Teil des eigentlichen tinySLAM Algoritmus, ist aber im weiterführenden Sourcecode, auf den im Folgenden eingegangen wird, enthalten.



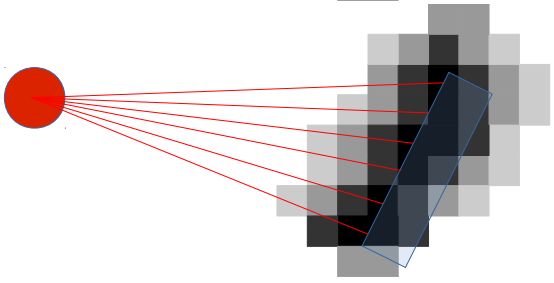
(a) Zu sehen ist ein Objekt (halbtransparenter, bläulicher Kasten) und die daraus resultierende, in dieser Skizze sehr grobe, Karte.



(b) Der Roboter hat sich im Vergleich zu Bild 4a etwas bewegt, die Räder sind dabei aber aufgrund eines Hindernisses durchgedreht (siehe auch Kapitel 2.3). Die Odometer haben dem Algorithmus also mitgeteilt, der Roboter habe sich ein wenig bewegt, in der Realität steht der Roboter allerdings immer noch an der gleichen Stelle und erkennt das Objekt genau so wie in Bild 4a. Die Verschiebung ist hier klar zu erkennen. Die Summe der Pixel, auf denen die Laserstrahlen enden, ist $120 + 200 + 255 + 255 + 200 + 200 = \mathbf{1230}$.



(c) Die Position des Roboters wurde zufällig minimal geändert. Die Summe der Pixel, auf denen die Laserstrahlen enden, ist $120 + 200 + 255 + 255 + 255 + 200 = \mathbf{1285}$, die zufällige Position passt also besser als in Bild 4b.



(d) Die Position des Roboters wurde erneut zufällig minimal geändert. Die Summe der Pixel, auf denen die Laserstrahlen enden, ist nun $255+200+255+255+255+255 = \mathbf{1475}$, die Position wurde also durch den Partikelfilter sehr genau wiedergefunden.

Abbildung 4: Visualisierung des Monte-Carlo Partikelfilters.

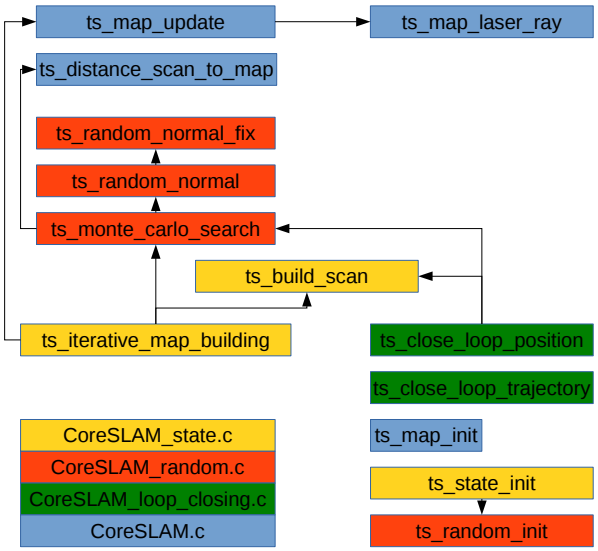


Abbildung 5: Blockschaltbild der Struktur und Abläufe von tinySLAM. Die Texte in den Blöcken sind der jeweilige Funktionsname, die Pfeile stehen für die Verschachtelung der Funktionen.

Die Datei *CoreSLAM_state.c* dient der Organisation und der Verknüpfung des gesamten SLAM Prozesses. Die hier verwendeten Funktionen sind nur schwierig zu übernehmen, da sie zum einen aufgrund der mangelhaften Dokumentation schwer nachzuvollziehen sind und die Verbindung zur Hardware aufgrund der schlechten Dokumentation nicht ohne einen erheblichen Aufwand umsetzbar ist. Einfacher ist es, diesen Teil selbst zu entwickeln und neu zu programmieren, worauf in Kapitel 3.4 eingegangen wird.

Die Datei *CoreSLAM_loop_closing.c* fällt gänzlich weg. Die Neustrukturierung wird durch Abbildung 6 visualisiert sowie in der darauf folgenden Auflistung detaillierter erklärt.

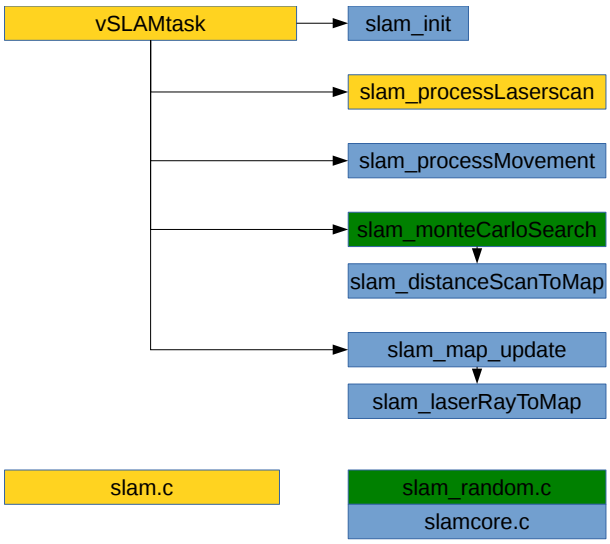


Abbildung 6: Blockschaltbild der optimierten Struktur und Abläufe von tinySLAM

- slam_init: Initialisiert die Karte sowie alle anderen nötigen Umgebungen und Variablen. Wird einmalig aufgerufen.
- slam_processLaserscan: Aufbereitung des vom LIDAR gelieferten Scans. Hier ist vorgesehen, die Bewegung des Roboters zu kompensieren (wenn der Roboter sich mit einer Geschwindigkeit von $0{,}3\frac{m}{s}$ bewegt und das LIDAR sich mit $5Hz$ dreht, für eine 360° Umdrehung also $200ms$ benötigt, bewegt sich der Roboter in dieser Zeit bereits $\frac{0{,}3\frac{m}{s}}{5s^{-1}} = 0{,}06m$, also dem sechsfachen der Auflösung der Karte). Diese Latenz sorgt für Ungenauigkeiten in der Karte.
- slam_processMovement: Berechnung der Positionsänderung des Roboters über Odometer

- slam_monteCarloSearch: Partikelfilter/Selbstlokalisierung in der Karte
- slam_map_update: Integriert den aktuellen LIDAR Scan in die Karte und vervollständigt sie somit.

3.4 Implementation

SLAM kann weniger als konkreter Algorithmus, sondern eher als ein Verfahren, das aus mehreren Algorithmen besteht, gesehen werden (siehe Kapitel 2.3), was sowohl in der alten tinySLAM Struktur (Abbildung 5) als auch in der überarbeiteten Struktur (Abbildung 6) ersichtlich wird. Die Schritte gliedern sich in:

1. Über Odometer die theoretische Bewegung des Roboter berechnen und die Position in der Karte so aktualisieren.
2. Über den Partikelfilter die Position in der Karte genau bestimmen.
3. Die Karte nun aktualisieren bzw. erweitern.

Dafür wurde die Datei *slam.c* erstellt, welche über das Echtzeitbetriebssystem (siehe Kapitel 2.4.4) periodisch die Abläufe regelt. Ausschlaggebend für einen Durchlauf der oben genannten drei Schritte ist das Vorhandensein von neuen Daten des LIDARs. Das Auswerten des LIDARs erfolgt zunächst interruptbasiert. Wenn das UART Modul des Mikrocontrollers ein Byte vom LIDAR empfängt, wird dieses zur Pufferung in eine Warteschlange gesendet. Von einem vom SLAM Task unabhängigen Task werden die Informationen weiterverarbeitet (siehe Protokoll, Kapitel 2.4.2). Diese beiden Tasks synchronisieren sich durch eine Semaphore (vgl. freertos.org 2014b), sobald das LIDAR eine neuen 360° Scan beginnt. Je nachdem, ob an dieser Stelle eine neue Karte erstellt wird oder ob die Karte bereits angefangen wurde und vervollständigt werden muss, verzweigt sich der Task an dieser Stelle. Wenn ersteres der Fall ist, werden die Schritte 1) und 2) übersprungen, ansonsten wird als nächstes die Positionsänderung des Roboters ermittelt. Nachdem der SLAM Task einmal durchlaufen ist, sendet dieser wiederum eine Semaphore zum Task, der für die Navigation und Koordination der eigentlichen Bewegung des Roboters zuständig ist.

Die verwendeten Odometer haben eine Auflösung von 1°, bei einer vollen Radumdrehung also 360 Ticks (Cytron 2011, S. 2 f.). Da es sich bei dem Fahrgestell um einen differentiellen Antrieb handelt (siehe Kapitel 2.1), bei dem zwei getrennt angetriebene Räder auf der selben Achse liegen, stehen zwei Encoderwerte zur Verfügung. Es kann also ermittelt werden, wie weit sich die linke und rechte Seite des Roboters vorwärts bewegt hat und darüber kann berechnet werden, in welche Richtung der Roboter nun zeigt und wie weit er sich von der Ausgangsposition fortbewegt hat. Entscheidend dafür ist zum einen die Länge der Achse, also der Abstand der Räder, und zum anderen der Umfang der Räder. Über diese Daten kann berechnet werden, wie weit der Roboter sich mit den aktuellen Geschwindigkeiten der Räder bewegt hat (nicht nur Geradeaus, sondern auch bei Kurvenfahrten). (vgl. Oubbati 2012b)

Die Monte-Carlo Suche erfolgt nach der ersten Schätzung der aktualisierten Position über die Odometer. Wie in Kapitel 3.1 erläutert, werden verschiedene Positionen im Zentimeterbereich um die über die Odometer geschätzte Position ausprobiert, um eine eventuell besser passende Position dafür zu finden. Im oben genannten Beispiel ist nach drei Versuchen die Position wiedergefunden worden, es handelt sich dabei aber nur um eine Skizze. Die in der Anwendung zu verarbeitende Datenmenge ist erheblich größer. In dem tinySLAM Beispielcode wird die Monte-Carlo Suche 100000 mal durchgeführt (vgl. openslam.org 2014b). Bei der dort verwendete Plattform läuft der Partikelfilter selbst allerdings auf einem Desktop-PC, der wesentlich mehr Leistung hat, als der hier verbaute ARM, weshalb die Anzahl der Versuche drastisch gesenkt werden muss (siehe auch Kapitel 4.1). (vgl. Steux u. a. 2010, S. 3 f.)

Probleme bei der Implementation gab es vor allem aufgrund der mangelhaften Dokumentation. Die Funktionen von tinySLAM sind nicht kommentiert, weshalb sehr viele Probleme durch Ausprobieren behoben werden mussten. Die Karte wurde bspw. ursprünglich spiegelverkehrt gespeichert, weil die Entfernungsmessungen des LIDARs bei dem hier verwendetem LIDAR in einer anderen Reihenfolge berücksichtigt wurden. Im tinySLAM Beispiel wird außerdem ein LIDAR mit einer höheren Auflösung, allerdings einer kleineren Spannweite (keine 360° Messungen) verwendet.

Zum ersten Testen des Algorithmus muss der Roboter sich selbstständig bewegen können. Dafür wurde ein sehr simpler Wall-Follower implementiert, der Roboter fährt also im Prinzip durch die Umgebung und vermeidet dabei Kollisionen. Dies diente als Zwischenschritt bis zur Implementation des Abfahrens von Wegpunkten (Kapitel 5.1).

4 Optimierung des Algorithmus

4.1 Geschwindigkeit

Ein großes Problem bei der Ausführung von SLAM auf dem Roboter ist die Geschwindigkeit. Bedingt ist dieses Problem durch die zwar insgesamt hohe, allerdings für tinySLAM nicht ausreichende Performance des verwendeten ARMs, weshalb die Anzahl der Versuche für das Ausprobieren neuer Positionen erheblich gedrosselt werden muss (siehe Kapitel 3.4).

Die Funktion *slam_distanceScanToMap* bewertet, wie gut der aktuelle Scan des LIDARs in die Karte passt (siehe auch Abbildung 5). Da für jeden der 360 Messungen geprüft werden muss, welchen Wert die Zelle der Karte an der Stelle hat und relativ viele Fließkommaberechnungen durchgeführt werden müssen (diverse Umrechnungen von polare in Kartesische Koordinaten), benötigt dies relativ viel Zeit. Jede zufällige Position muss dabei für die Monte-Carlo Suche bewertet werden, diese Zeit muss also zusätzlich noch mit der Anzahl der Versuche multipliziert werden (siehe auch Kapitel 3.2).

Für die Bestimmung der Passgenauigkeit des Scans werden allerdings nicht zwangsläufig alle 360 Messwerte benötigt. Es reicht völlig, jeden 10. Wert zu nutzen, sodass für diese Bestimmung nur noch 36 Umrechnungen durchgeführt werden müssen. So kann die Zeit, die benötigt wird, um einen Scan zu verarbeiten (die Positionsbestimmung) um den Faktor 10 beschleunigt werden. Für die Integration des Laserscans in die Karte werden alle 360 Messungen verwendet. Die Anzahl der Versuche kann trotzdem nicht, wie in der Originalversion, um 100000 liegen (siehe Kapitel 3.4). 1000 Versuche reichen i.d.R. aus.

4.2 Sonstiges

Die tinySLAM Kartendatenstruktur sieht eine zweidimensionale Reihung vor. Zur Speicherung mehrerer Ebenen kann eine dritte Dimension hinzugefügt werden, kann aber nicht als Speichermöglichkeit einer dreidimensionalen Umgebung gesehen werden.

Bei den Tests mit dem Wall-Follower hat sich gezeigt, dass die Karte sich nach einiger Zeit dreht (siehe Abildung 7). Der Scan des LIDARs wird über einen $\alpha\beta$ Filter (Steux u. a. 2010, S. 2) in die Karte integriert. Über einen Wert zwischen 0 und 100 kann eingestellt werden, we stark der Scan in die Karte integriert wird. Ein Wert von 0 bedeutet gar keine Integration, ein Wert von 100 bedeutet volle Integration (komplette Überschreibung der bereits vorhandenen Werte). Die Ursache der Drehung der Karte lag zum einen an der zu dem Zeitpunkt nicht vorhandenen Synchronisation zwischen LIDAR und SLAM Task (siehe oben), aber auch die Bewegung des Roboters spielt eine erhebliche Rolle. Bei einer ständigen Geradeausfahrt dreht sich die Karte nicht, das heißt, die Drehung in der Karte wird durch die Drehung des Roboters in der Umgebung verursacht. Die in Kapitel 3.3 erwähnte Latenz gilt auch für die Drehung des Roboters um die eigene Achse. Wenn der Roboter ständig im Kreis fährt (was gerade bei Wall-Followern der Fall ist), verstärkt das diesen Effekt zusätzlich. Das soll durch eine verminderte Integration in die Karte bei Kurvenfahrten verhindert werden. Der Integrationsfaktor sollte generell nicht höher als 10 sein. Das ist wichtig, damit bei einer ungenauen Selbstlokalisierung der Fehler nicht zu stark ins Gewicht fällt und die genauen Lokalisierungen überwiegen. Der Integratonsfaktor soll also zwischen 0 und 10 liegen und abhängig von der Rotationsgeschwindigkeit des Roboters sein.

Abbildung 7: Drehung in der Karte nach 20 Minuten Laufzeit

Die Rotationsgeschwindigkeit ist gegeben durch den Betrag der Differenzen der Motorgeschwindigkeiten: $\omega = |v_l - v_r|$. Bei einer Geradeausfahrt wäre dieser Wert 0, in dem Fall soll der Integrationsfaktor aber 10 sein, weshalb gilt: $fac = 10 - \omega$ bzw. $fac = 10 - |v_l - v_r|$. Für den Fall, dass ω größer als 10 ist, muss fac nun noch begrenzt werden, sodass es zwischen 0 und 10 liegt.

Der Zusammenhang zwischen Rotationsgeschwindigkeit und Integrationsfaktor ist nun linear. Versuche haben ergeben, dass ein quadratischer Funktionszusammenhang keine bemerkbaren Besserungen im

Vergleich zum linearen Zusammenhang mit sich bringt.

Die Rotation in der Karte wurde auf diese Art und Weise behoben. Vorraussetzung, dass das funktioniert, ist, dass der Roboter sich möglichst wenig dreht sondern nevorzugt geradeaus fährt, da die Karte ansonsten nicht aktualisiert wird (dieses Problem kann dadurch behoben werden, dass der Faktor nicht auf Werte zwischen 0 und 10 begrenzt wird, sondern z.B. auf Werte zwischen 3 und 10, sodass die Karte niemals nicht aktualisiert wird.

Neben dem Integrationsfaktor für die Karte ist die Anzahl der Versuche der Monte-Carlo Suche, also wie oft eine neue Position in der gegebenen Streuung auf ein besseres Passen in die Karte getestet wird, variabel. Je höher dieser Wert ist, desto genauer ist die Lokalisierung (weil mehr mögliche Positionen ausprobiert werden), aber desto länger dauert die Lokalisierung (weil bei jedem Scan die Qualität der Passgenauigkeit in die Karte überprüft werden muss). Die Laufzeit steigt dabei linear an (siehe auch Abbildung 9). Da das LIDAR alle $200ms$ einen neuen Scan liefert, stehen nur etwa $150ms$ für die Verarbeitung des Scans, also des gesamten Scan Prozesses, zur Verfügung (die restlichen $50ms$ sollen anderen Tasks zur Verfügung stehen. Das ist relativ wenig, was sich durch ein bei aktiver Lokalisierung träges GUI auf dem Roboter bemerkbar macht, was allerdings nicht schlimm ist, da die Kartenerstellung vorrang über alle anderen Prozesse haben soll). Die Laufzeit des SLAM Tasks wird nun über einen Regler über die Anzahl der Versuche für die Monte-Carlo Suche auf etwa $150ms$ geregelt. So wird für die zur Verfügung stehende Zeit immer ein Optimum an Versuchen durchgeführt. Ein statischer Wert ist nicht so effektiv, da die Zeit außerdem davon abhängig ist, wie viele Laserstrahlen tatsächlich noch in der Karte enden. Die Laserstrahlen, die außerhalb der Größe des Speichers der Karte auf ein Hindernis treffen, müssen nicht berücksichtigt werden, was dazu führt, dass u.U. mehr Zeit zur Verfügung steht, die dann für weitere Versuche genutzt werden kann.

4.3 Analyse des Prozesses

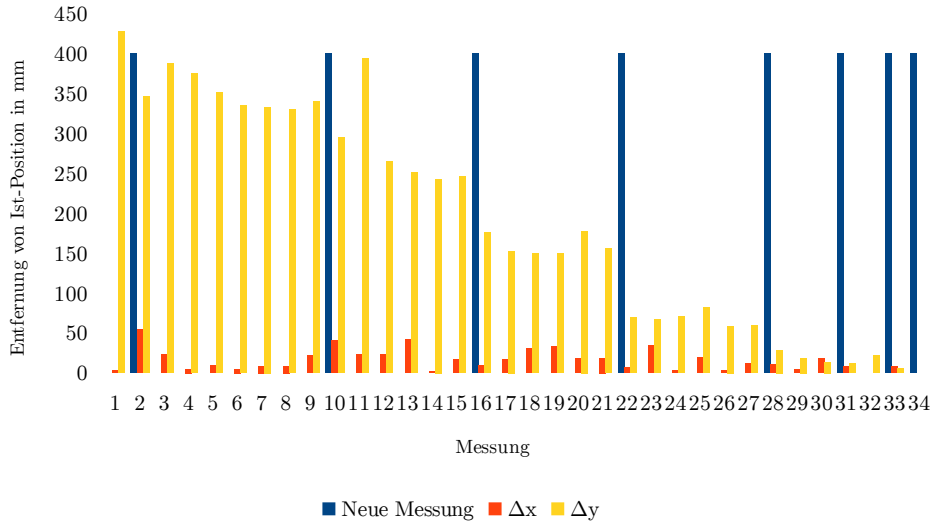


Abbildung 8: Versuch: Arbeitsweise der Monte-Carlo Suche

Abbildung 8 zeigt das Ergebnis des Partikelfilters. In dem Versuch wurde der Roboter innerhalb von ca. einer Sekunde bei Stillstand etwa $40cm$ nach vorne geschoben. Dieser Versuch stellt eine Extremsituation dar, normalerweise würde die Roboterposition niemals so extrem von der realen Position abweichen, er zeugt dennoch von der guten Funktionalität des Algorithmus.

Es wurde die Position des Roboters über die serielle Schnittstelle per Bluetooth geloggt und im Nachhinein von den gegebenen absoluten Positionsdaten die Endposition, also die Position, auf die der Roboter geschoben wurde, subtrahiert und der Betrag gebildet, sodass Δx (orange) und Δy (gelb) die Abweichung von der tatsächlichen Position darstellen, die Werte also die relative Position bzw. die Abweichung vom tatsächlichen Ort sind. Eine Position wurde nur zum Computer gesendet, wenn der Scan besser in die Karte passt, als der bis zu dem Zeitpunkt beste Scan, weshalb es insgesamt nur 32 Messwerte gibt, obwohl die Anzahl der Versuche der Monte-Carlo Versuche hier bei ca. 1000 lag (alle anderen zufälligen Positionen passten also schlechter, als die bis dahin beste Position in die Karte). Die blauen Linien stellen die Zeitpunkte dar, an denen das LIDAR sich erneut um 360° gedreht hat, also $200ms$ vergangen sind.

Während des Logs wurden 6 neue Monte-Carlo Suchen gestartet, es sind also 1,2s vergangen.

Es ist erkennbar, dass y-Position fast linear von ca. 400mm auf 0mm sinkt, während sich die x-Position nur minimal ändert. Dies ist dadurch zu erklären, dass der Roboter parallel zur y-Achse verschoben wurde und nicht etwa diagonal.

Die Abstände zwischen den Starts eines neuen Scans werden immer kleiner, je näher die ermittelte Position an der tatsächlichen Position liegt. Je näher die ermittelte Position an der tatsächlichen Position liegt, desto kleiner wird der Bereich für Positionen, in denen der Scan noch besser in die Karte passt, es ist also schwieriger, eine noch besser passende Position zu finden, weshalb dieses Ereignis dann nur noch seltener eintritt.

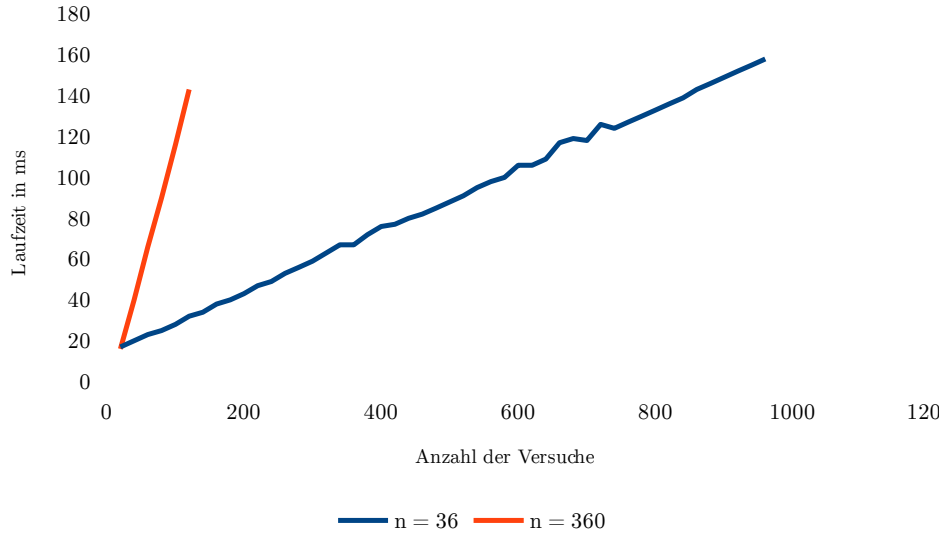


Abbildung 9: Versuch: Laufzeit der Monte-Carlo Suche

Abbildung 9 zeigt einen linearen Zusammenhang zwischen Anzahl der Versuche der Monte-Carlo Suche und Laufzeit der Suche. Dieser Test wurde einmal mit $n = 36$ (blaue Gerade) und einmal mit $n = 360$ (orange Gerade) Entfernungsmesswerten, für die die Passgenauigkeit in die Karte geprüft wurde, durchgeführt.

Die Steigung der Kurven ergibt sich aus $m = \frac{\Delta y}{\Delta x}$: $m_{blau} = \frac{158ms-17ms}{960-20} = 0{,}15$ und $m_{orange} = \frac{143ms-16ms}{120-20} = 1{,}27$. Durch den Quotienten der beiden Steigungen $\frac{\Delta y}{\Delta x} = 8{,}4\overline{6}$ kommt man erneut ungefähr auf den Faktor 10 (siehe Kapitel 4.1). Durch andere Faktoren und variierender Laufzeiten anderer Tasks des Echtzeitbetriebssystems ist dieser Faktor nicht exakt 10.

5 Erweiterungen

5.1 Abfahren von Wegpunkten

5.1.1 Autonomie

Die autonome Navigation ist wichtiger Bestandteil von Rettungsrobotern oder Rovern in der Raumfahrt. (vgl. JPL 2013)

Für die Erstellung der Karte muss der Roboter sich durch die Umgebung bewegen, um die Karte zu vervollständigen. Der in Kapitel 3.4 implementierte Wall-Follower ist dafür nur unzureichend geeignet. Es ist nicht garantiert, dass die Umgebung vollständig kartiert wird, wenn es z.B. Inseln gibt, also Teilgebiete, die nicht über einen Wall-Follower erreicht werden können.

Autonomie bedeutet in der Robotik grundsätzlich ein selbstständiges Handeln eines Systems. Zunächst muss der Grad der Autonomie unterschieden werden. Je komplexer die Umgebung des Roboters (also das Arbeitsumfeld, Rettungsroboter müssen mit komplexeren Situationen umgehen können als Industrieroboter, die immer die selben Arbeitsschritte erledigen, aber trotzdem autonom arbeiten), desto höher ist der Grad der Autonomie. (vgl. Oubbati 2012a)

Kernproblem bei einer Erweiterung des autonomen Verhaltens des Roboters ist die Entscheidung, welche Punkte in der Umgebung angefahren werden sollen. Um dies autonom vonstatten gehen zu lassen, müssten alle Punkte in der Karte betrachtet werden, die noch keine Informationen enthalten und entschieden werden, ob die Punkte erreicht werden können. Diese Schritte sind sehr rechen- und speicherintensiv, weshalb die Punkte, die vom Roboter angefahren werden, von einem Menschen vorgegeben werden sollen. Der Roboter soll dann selbstständig den Weg zu diesen Punkten finden. Diese Vorgehensweise macht den Roboter halbautonom. Die Wegpunkte, die besucht werden sollen, werden vorgegeben, der Roboter übernimmt dann die Erstellung der Karte und die Navigation zu diesen Punkten.

Diese Vorgehensweise wird auch in der unbemannten Raumfahrt eingesetzt. Marsrover, wie z.B. die Rover Opportunity und Spirit, handeln nicht vollautonom. Es müssen bestimmte wissenschaftliche Fragestellungen des Menschen beantwortet werden, weshalb an Punkte navigiert wird, die bei der Beantwortung dieser Fragestellungen relevant sind - diese Entscheidung wird also dem Menschen überlassen. Wie der Roboter dann an das Ziel gelangt, muss autonom vom Roboter entschieden werden. Das ist sinnvoll und auch elementar wichtig, weil keine exakten Daten über die Marsoberfläche zur Verfügung stehen und die detaillierten Karten für die Navigation deshab vom Rover selbst erstellt werden müssen. Ein weiterer wichtiger Aspekt ist die Laufzeit des Signals. Dies benötigt mehrere Minuten von der Ausstrahlung auf der Erde bis zur Ankunft auf dem Mars, ohne eine Autonomie des Roboters sondern mit direkter Steuerung würde der Rover sich nur wenige Zentimeter pro Tag bewegen, da ständig von Menschen beurteilt werden muss, ob das Gelände befahrbar ist. (vgl. JPL 2013)

Ähnlich ist die Situation bei Rettungsrobotern: Autonomes Verhalten ist hier nicht immer gewünscht. Speziell bei der Suche nach Opfern ist es sinnvoll, den Menschen enscheiden zu lassen, welche Gebiete inspiziert werden sollen, weil die Wahrscheinlichkeit, an bestimmten Stellen Opfer zu finden z.B. höher ist.

5.1.2 Implementation

Verwaltung der Wegpunkte

Die Wegpunkte auf dem Roboter sollen in der Reihenfolge, in der sie angelegt wurden, abgefahren werden. Wegpunkte sollen außerdem gelöscht werden können und zwischen zwei Wegpunkten ein zusätzlicher Wegpunkt eingefügt werden können. Hier wurde sich für eine doppelt verkettete Liste entschieden.

Eine verkettete Liste besteht aus einer variablen Anzahl an Elementen mit Eigenschaften, die jeweils einen Verweis auf das vorherige und das nächste Element in der Liste haben. (vgl. Wolf 2009, S. 717 ff.) Für die Wegpunkte sind zunächst eine ID, also eine eindeutige Identifikationsnummer eines Wegpunktes (damit gezielt Eigenschaften von Wegpunkten geändert werden können oder Wegpunkte gelöscht werden können), sowie Positionsdaten der Wegpunkte vorgesehen (siehe Abbildung 10). So können die Wegpunkte sehr effektiv verwaltet werden, eine Erweiterung der Wegpunkte mit weiteren Eigenschaften ist sehr einfach möglich.

In der Liste können Elemente nicht direkt angesprochen werden, sondern die Liste muss durchsucht werden, um bspw. einen Wegpunkt mit einer bestimmten ID zu finden. Eine doppelt verkettete Liste ist

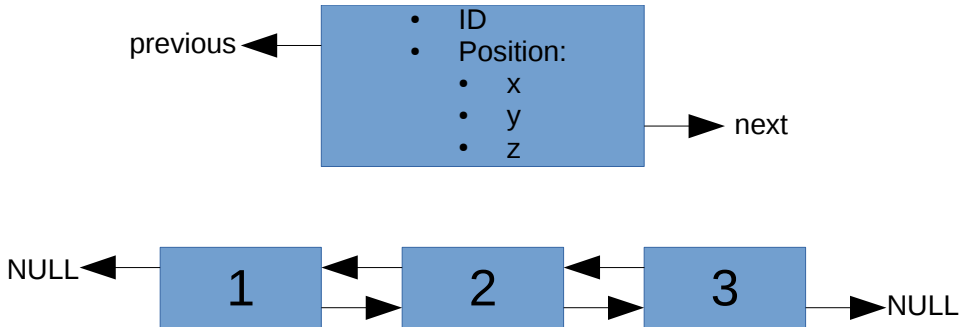


Abbildung 10: Doppelt verkettete Liste

für diese Anwendung nicht zwangsläufig notwendig, aus Performancegründen wurde sich allerdings dafür entschieden. Da bei der Anwendung mit Wegpunkten sehr oft die letzten Elemente der Liste angesprochen werden müssen, muss die Liste so nicht jedes Mal von vorne bis nach hinten durchgegangen werden, um das Element zu finden.

Beim Hinzufügen von Elementen einer verketten Liste wird i.d.R. dynamisch zur Laufzeit Speicher vom Heap allokiert (vgl. Wolf 2009, S. 719). Bei eingebetteten Systemen mit relativ wenig Ressourcen, wie auch der Roboter eines ist, wird dies meistens vermieden und der Speicher statisch beim Hochfahren des Systems allokiert. So kann es nicht zu unvorhersehbaren Problemen kommen, weil aufgrund eines möglichen Fehlers kein Speicher mehr zur Verfügung steht. Aktuell wird zu Beginn Speicher für 100 Wegpunkte in einer Reihung von Wegpunkten reserviert. Die ID der Wegpunkte wird zu Beginn mit -1 initialisiert. Wenn nun ein neuer Wegpunkt hinzugefügt wird, nimmt die Funktion zum Hinzufügen eines Wegpunktes das erste Element mit der ID -1 aus der Reihung. Für jeden hinzugefügten Wegpunkt wird die ID inkrementiert, die Eigenschaften (Position) initialisiert sowie die Pointer auf das nächste und letzte Element in der Liste angepasst. Beim Löschen wird die ID allerdings nicht dekrementiert, sodass die ID eine fortlaufende, einmalige Nummer darstellt und keine Aussage über die aktuelle Anzahl der Wegpunkte macht. Wird ein Wegpunkt gelöscht, wird die ID auf -1 gesetzt und die Pointer der umgebenden Wegpunkte entsprechend verbunden, das Element kann so beim nächsten Hinzufügen eines neuen Wegpunktes quasi „recycelt“.

Der Start und das Ende der Liste können über die Wegpunkte erkannt werden, deren Pointer für den nächsten bzw. vorherigen Wegpunkt noch nicht referenziert wurden.

Anfahren der Wegpunkte

Für eine Navigation zu den Wegpunkten gibt es verschiedene Ansätze. Eine Möglichkeit ist die direkte Berechnung des Pfades. Dafür gibt es eine Vielzahl an verschiedenen Algorithmen, grundsätzlich suchen diese Algorithmen aber lediglich den kürzesten Pfad zwischen zwei Punkten, die besuchbar sein können müssen - d.h. hier ist man wieder beim Ursprungsproblem (siehe Kapitel 5.1.1). Der Roboter ist grundsätzlich in der Lage, die nötigen Berechnungen durchzuführen, der Einfachheit halber wird allerdings zunächst davon ausgegangen, dass ein Wegpunkt immer von den Nachbarwegpunten aus sichtbar ist. Wenn die Wegpunkte also festgelegt werden, darf sich kein Hindernis zwischen zwei Wegpunkten befinden, sodass der Roboter einfach immer den nächstgelegenen Wegpunkt anpeilt und direkt in die Richtung des Wegpunktes fährt. Dieser Ansatz kann ggf. mit dem Wall-Follower kombiniert werden, sodass der Wall-Follower ein konkretes Ziel hätte. Bei einer zukünftigen Implementation von Pfadberechnungsalgorithmen kann diese Anpeilung beibehalten werden, denn bei Pfadberechnungen muss ein Raster bestimmt werden, das eine kleinere Auflösung als die Karte hat, die einzelnen definitiv besuchbaren Zwischenpositionen bzw. Zellen müssten dann auch über eine Peilung angefahren werden.

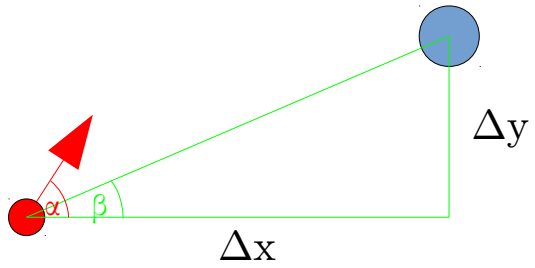


Abbildung 11: Anpeilen von Wegpunkten

Für die Peilung muss die Abweichung der Orientierung, also des Winkels des Roboters, von dem Ort des Wegpunktes in Relation zum Roboter ($\alpha - \beta$) berechnet werden (über den Wert können dann über einen Regler die Geschwindkeiten der linken und rechten Räder berechnet werden, um die Abweichung so klein wie möglich zu halten). Die Orientierung des Roboters α ist bekannt. Über den Satz des Pythagoras

$d = \sqrt{\Delta x^2 + \Delta y^2}$ wird zunächst der aktuelle Abstand zwischen Roboter und Wegpunkt berechnet. Der Winkel wird nun durch eine einfache Umrechnung der kartesischen Koordinaten in polare Koordinaten bestimmt. Dafür ist eine Fallunterscheidung notwendig, da das Vorzeichen von Δy durch die Quadrierung in d nicht mehr enthalten ist: $\beta = \arccos(\frac{\Delta x}{d})$ für $(\Delta y \geq 0)$ und $\beta = \arccos(-\frac{\Delta x}{d}) - \pi$ für $(\Delta y < 0)$.

5.2 PC User Interface

Der Aspekt Autonomie ist, wie in Kapitel 5.1.1 erläutert, sehr wichtig. Da der Roboter selbst ohne Eingreifen eines Menschen arbeiten soll, muss ein Interface geschaffen werden, mit dem sich der Roboter fernsteuern lässt. Dabei soll zum einen die Karte ständig und in Echtzeit eingesehen werden können und zum anderen sollem dem Roboter in Echtzeit die Wegpunkte mitgeteilt werden können.

5.2.1 Kommunikation mit dem PC

Für die Kommunikation mit dem PC, der die externe Mensch/Maschine Schnittstelle darstellt, wurde ein eigenes Protokoll für die Kommunikation über das eine serielle Schnittstelle emulierende Bluetoothmodul geschaffen. Das Protokoll sieht wie folgt aus:

[S][L][CHK][ID](N*<D>)

- S: Startsequenz: 8 Bytes (*"PCUI_MSG"*)
- L: Länge der Nachricht: 2 Bytes (LSB/MSB)
- CHK: 4 Bytes (LSB ... MSB)
- ID: 3 Bytes
- D: Datenbyte: 1 Byte

Dieses Protokoll kann zuverlässig sehr große Datenpakete übermitteln (theoretisch bis zu 65536 Bytes), ist allerdings nicht bidirektional, es wird allerdings in beide Richtungen verwendet (der Roboter sendet die Karte und der PC die abzufahrenden Wegpunkte). Da die Daten der Karte vom Roboter kontinuierlich gesendet werden, wird bei Datenverlust das Paket ignoriert und beim nächsten Mal empfangen. Die Form des Pakets unterscheidet sich allerdings noch in der ID. Für jede ID kann eine beliebige weitere Paketform definiert werden, die dann entsprechend vom Empfänger interpretiert werden können muss. Aktuell gibt es folgende IDs:

- **MPD:** *Map Data*: 13 Bytes: Enthält alle relevanten Informationen über die Karte (Auflösung, Größe, Roboter Position).
- **MAP:** *Map*: (Anzahl der Zellen in der Karte in X-Richtung + 3) Bytes: Enthält eine Zeile der Karte (nicht kodiert) sowie Information darüber, welche Zeile.
- **MAR:** *Map Runlength*: (Anzahl der Bytes Variabel): Enthält eine Zeile der Karte (Lauflängenkodiert) sowie Information darüber, welche Zeile.
- **LWP:** *List Waypoints*: (Anzahl der Wegpunkte · 9 + 2) Bytes: Enthält Informationen über Wegpunkte (Position, ID sowie ID des vorherigen Wegpunktes) sowie die Anzahl.

Die Karte wird zeilenweise übertragen. Der Grund dafür ist die Notwendigkeit einer Checksumme sowie der andernfalls große Speicher- und Zeitverbrauch: Für die Pakete, die gesendet werden, muss eine Checksumme berechnet werden. Das Senden eines Bytes über die serielle Schnittstelle dauert eine gewisse Zeit (bei der eingestellten Baudrate von $460800b/s$ genau $t_{Byte} = \frac{1}{460800b/s} \cdot 8 = 1{,}736\overline{1} \cdot 10^{-5}s$). Das Senden funktioniert wie das Empfangen interruptbasiert, die zu sendenden Bytes kommen also zunächst in eine Warteschlange (oder auch Queue bzw. FIFO (vgl. freertos.org 2014b)), die dann byteweise abgearbeitet wird, wenn das letzte Byte gesendet wurde und der Sendeinterrupt somit erneut aufgerufen wird. Angenommen, die Karte hat eine Größe von $300 \cdot 300$ Zellen (bei einer Auflösung von $1cm$ entspräche das einer Größe von $3m \cdot 3m$). Das Senden der kompletten Karte würde dann $t_{Byte} \cdot 300 \cdot 300 = 1{,}5625s$ dauern. Wenn die Karte also nicht zeilenweise, sondern komplett übertragen werden würde, müsste entweder eine Kopie der kompletten Karte gemacht werden (wofür aber nicht genug Speicher zur Verfügung steht) oder die Karte dürfte während dieser Zeit nicht vom SLAM Task bearbeitet werden, weil die Checksumme

sonst nicht mehr passt (weil die Karte während der Übertragung noch geändert wurde und die Checksumme vorher berechnet werden muss). Da beides keine Option ist, muss die Karte zeilenweise übertragen werden. Für das Puffern einer Zeile steht genug Speicher zur Verfügung. Es wird also zunächst eine Zeile der Karte zwischengespeichert, davon die Checksumme berechnet und diese Zeile dann gesendet.

Die Karte besteht im Wesentlichen aus entweder grauen, weißen oder schwarzen Zellen und Graustufen, wobei sich oft viele graue und weiße Zellen in einer Reihe befinden. Deshalb ist es sinnvoll, die Karte teilweise Lauflängenkodiert (fh–duesseldorf.de 2015) und teilweise (wenn die Lauflängenkodierung länger als eine Übertragung ohne Kodierung ist) ohne Kodierung zu übertragen.

5.2.2 Processing

Processing ist eine Community, Entwicklungsumgebung mit Präprozessor sowie Bibliothek zur einfacheren Benutzung der Programmiersprache Java. Es bietet vollen Zugriff auf alle Java Bibliotheken und bietet eigene Funktionen, die den Umgang erheblich vereinfachen. Processing soll Schülern und Künstlern den Zugang zur Programmierung vereinfachen. (vgl. github.com 2015b)

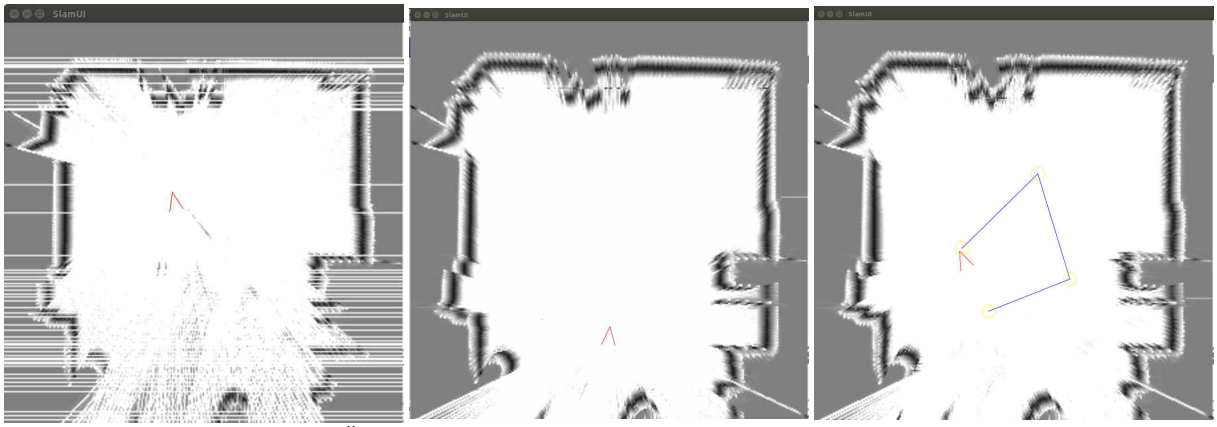
Für die Programmierung des User Interfaces in Java/Processing wurde sich aufgrund der sehr einfachen Benutzung und der insgesamt kleinen Größe des Projekts entschieden.

Die Entwicklungsumgebung ist sehr primitiv, es ist aber prinzipiell möglich, auch komplexere Entwicklungsumgebungen wie Eclipse zu benutzen. (vgl. ebd.)

5.2.3 User Interface

Das User Interface selbst besteht lediglich aus einem Fenster, das die Karte anzeigt. Das Fenster ist skalierbar, mit dem Fenster skaliert sich auch die Karte, sodass die Karte detailliert betrachtet werden kann.

Die Wegpunkte werden mit einem Mausklick (die Taste ist dabei egal) hinzugefügt. Sobald es mehrere Wegpunkte gibt, werden sie mit einer Linie miteinander verbunden. Befindet man sich mit der Maus über einem Wegpunkt, wird dieser farbig hervorgehoben. Wird dann die linke Maustaste gedrückt, kann der Wegpunkt verschoben werden. Ein Klick mit der rechten Maustaste löscht den Wegpunkt.



(a) Karte kurz nach Beginn der Übertragung im Aufbau. Die zeilenweise Übertragung ist gut erkennbar. Einzelne Zeilen werden sporadisch übersprungen, weil die Checksumme nicht übereinstimmte.

(b) Fertig übertragene Karte (i.d.R. nach wenigen Sekunden).

(c) Karte mit eingezeichneten Wegpunkten.

Abbildung 12: SlamUI. Die Karte zeigt einen Ausschnitt eines $3m$ breiten Raumes.

6 Schlussbetrachtung

Innerhalb von 291 Arbeitsstunden konnte der zuvor zur Teilnahme am RoboCup genutzte Roboter um eine umfangreiche und leistungsfähige Hardware und einen gut funktionierenden SLAM Algorithmus erweitert werden. Es wurden insgesamt ca. 2000 Zeilen C-Code auf der Roboterplattform und ca. 7000 Zeilen C-Code auf der Erweiterungsplatine programmiert. Externe Bibliotheken und das Echtzeitbetriebssystem eingeschlossen besteht das Projekt aus insgesamt ca. 60000 Zeilen C-Code.

Der tinySLAM Algorithmus ist im Vergleich zu anderen Algorithmen sehr primitiv und dürfte zu den primitivsten SLAM Algorithmen gehören, dennoch ist die Funktionsweise nach Einarbeitung in den Quellcode trotz mangelhafter Dokumentation ersichtlich, logisch und einfach verständlich.

Der Vorteil des tinySLAM Algorithmus gegenüber anderen SLAM Algorithmen ist das Wegfallen des Extrahierens der Landmarks. Der Partikelfilter ist sehr flexibel. Selbst sich bewegende Gegenstände oder Personen in Reichweite des LIDARs sorgen für keine Probleme, solange genügend andere befestigte Gegenstände in Reichweite sind. Außerdem ist der Algorithmus so unabhängig von der Form der Gegenstände. Sowohl Wände, als auch bspw. Stuhlbeine dienen so als Landmark (es müssen nicht separat Linien und Punkte aus den Scans extrahiert werden).

Insgesamt arbeitet der Algorithmus sehr schnell, optimiert werden kann dennoch an vielen Stellen. Auf eine tiefergehende Analyse wurde in dieser Arbeit verzichtet.

Der tinySLAM Algorithmus kommt allerdings nicht gut mit Umgebungen zurecht, die sehr länglich sind, z.B. Korridore oder Flure. Dies ist durch die Monte-Carlo Suche zu begründen. Der Scan des LIDARs ließe sich überall entlang des Korridors einordnen. Selbst mit Unterstützung durch Odometer wird sich der Algorithmus immer wieder selbst falsch korrigieren.

Ungenauigkeiten zeigen sich auch in sehr verwinkelten Umgebungen. Es müssen noch viele Experimente mit den Parametern des Algorithmus gemacht werden (speziell die Rampe, die die Werte in der Karte bis zum Hindernis fallen und steigen lässt). Problematisch ist auch die Anzahl der Iterationen, die mit etwa 1000 auf dem verwendeten Computer nur relativ gering ist im Vergleich zu 100000 im tinySLAM Beispiel. Hier ist allerdings der Vorteil der kompletten Kapselung des aufgebauten Systems zu nennen. Der Roboter kann auch ohne einen externen Computer arbeiten.

Das Grundgerüst des Roboters verfügt außerdem über eine sehr leistungsfähige IMU, die den SLAM Prozess zukünftig erheblich optimieren und unterstützen kann.

Mit einer Kartengröße von $3m \cdot 3m$ und einer Auflösung von $1cm$ ist der SRAM des $32bit$ Mikrocontrollers nahezu vollständig belegt. Diese Größe ist für Demonstrationszwecke ausreichend. Für eine sinnvolle Verwendung des Roboters muss der SRAM aber erweitert werden. Denkbar ist die Verwendung einer anderen Version des ST Discoveryboards, welches über ein $8MB$ großes, externes SRAM verfügt, in das eine Karte mit einer Kantenlänge von bis zu $\sqrt{8MB} \approx 2828$ Zellen, was bei einer Auflösung von $1cm$ eine Karte mit einer Kantenlänge von $28m$ entspricht, speichern könnte. (vgl. ST 2014a)

Der Roboter verfügt weiterhin über zwei IR Temperatursensoren, die berührungslos die Temperatur links und rechts neben dem Roboter messen können und zum Detektieren der verschütteten, wärmeabstrahlenden Opfer im RoboCup Wettbewerb dienten. Hier ist eine Integration der Werte dieser Sensoren in die Karte denkbar, sodass die Karte der physischen Umgebung auch Informationen über die Wärmeverteilung und so über eventuell verschüttete Opfer gibt.

Die Navigation ist aktuell ungenau, der Pfad, den der Roboter zu fahren hat, ist noch sehr stark vom Benutzer der Software auf dem Computer vorzugeben. Hier ist eine Implementation von (komplexeren) Pfadberechnungsalgorithmen denkbar. Danach kann auch ein vollautonomer Modus implementiert werden, in dem der Roboter selbst einschätzt, welche Gebiete der Umgebung es sich noch lohnen könnte zu besuchen.

Die Software auf dem PC zur Darstellung der Karte und zum Setzen der Wegpunkte ist aktuell noch sehr primitiv. Hier ist eine Erweiterung um ein richtiges GUI denkbar, um den Roboter komplett fernsteuern zu können und aus der Ferne Fehler diagnostizieren zu können. Der Verbindungsaufbau zum Bluetooth Modul erfolgt aktuell noch mit einem externen Tool, die serielle Schnittstelle muss bei Bedarf im Quellcode geändert werden.

Literaturverzeichnis

Atmel (2011). *8-bit Atmel Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash.* Atmel Corporation. San Jose. URL: atmel.com/images/doc2549.pdf (besucht am 15. 03. 2014).

Bajracharya, Suraj (2014). *BreezySLAM: A Simple, efficient, cross-platform Python package for Simultaneous Localization and Mapping.* Lexington: Washington und Lee University. URL: home.wlu.edu/~levys/students/BreezySLAM_SurajBajracharya.pdf (besucht am 01. 11. 2014).

Bonilla, Roberto, Fredrik Lofgren, Tiago Caldeira, Naomi Chikuma, Elizabeth Mabrey und Kai Hannemann (2013). *RoboCup Junior Rescue B - Rules 2013.* URL: rcj.robocup.org/rcj2013/rescueB_2013.pdf (besucht am 09. 12. 2014).

Cytron (2011). *DC Geared Motor with Encoder MO-SPG-30E-XXXK.* Cytron Technologies. Johor. URL: robotshop.com/content/ZIP/documentation-spg30e-30k.zip (besucht am 30. 12. 2014).

fh–duesseldorf.de (2015). *Queues, Mutexes, Semaphores... [Inter-task communication and synchronisation].* URL: swlab.et.fh-duesseldorf.de/pc_pool/lernmodule/multimediadateien/Kapitel63.htm (besucht am 04. 01. 2015).

Eliazar, Austin und Ronald Parr (2003). *DP-SLAM: Fast, Robust Simultaneous Localization and Mapping Without Predetermined Landmarks.* URL: people.ee.duke.edu/~lcarin/Lihan4.21.06a.pdf (besucht am 28. 12. 2014).

Engel, Jakob, Jürgen Sturm und Daniel Cremers (2013). *Camera-Based Navigation of a Low-Cost Quadrocopter.* München: Technische Universität München. URL: vision.in.tum.de/_media/spezial/bib/engel12iros.pdf (besucht am 26. 12. 2014).

freertos.org (2014a). *FreeRTOS^{TM}. The Market Leading, De-facto Standard, Cross Platform RTOS.* URL: freertos.org (besucht am 28. 12. 2014).

— (2014b). *Queues, Mutexes, Semaphores... [Inter-task communication and synchronisation].* URL: freertos.org/Embedded-RTOS-Binary-Semaphores.html (besucht am 30. 12. 2014).

Fujii, Takashi (2005). *Laser Remote Sensing.* New York: Springer.

github.com (2015a). *BreezySLAM.* URL: github.com/simondlevy/BreezySLAM (besucht am 11. 01. 2015).

— (2015b). *Processing FAQ.* URL: github.com/processing/processing/wiki/FAQ (besucht am 04. 01. 2015).

JPL (2013). *Autonomous navigation.* Jet Propulsion Laboratory, California Institute of Technology. Pasadena. URL: mars.nasa.gov/mer/home/posters/OpportunityPosterBack.pdf (besucht am 25. 12. 2014).

mikrokopter.de (2014). *HC-06 Bluetooth Modul.* URL: wiki.mikrokopter.de/en/HC-06 (besucht am 27. 12. 2014).

neatorobotics.com (2014). *Neato XV Essential. Der Roboter-Staubsauger für den kleinen Geldbeutel.* URL: neatorobotics.com/de/robot-vacuum/xv/ (besucht am 27. 12. 2014).

openslam.org (2014a). *OpenSLAM – Give your algorithm to the community.* URL: openslam.org (besucht am 28. 12. 2014).

— (2014b). *tinyslam - Revision 22.* URL: svn.openslam.org/data/svn/tinyslam/trunk/ (besucht am 28. 12. 2014).

Oubbati, Mohamed (2012a). *Einführung in die Robotik: Autonome Mobile Roboter*. Ulm: Universität Ulm. URL: uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.130/Mitarbeiter/oubbati/RobotikWS1113/Folien/AMR.pdf (besucht am 04. 01. 2015).

— (2012b). *Einführung in die Robotik: Differentialsantrieb*. Ulm: Universität Ulm. URL: uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.130/Mitarbeiter/oubbati/RobotikWS1113/Folien/Differentialantrieb.pdf (besucht am 27. 12. 2014).

Riisgaard, Søren und Morten Rufus Blas (2005). *SLAM for Dummies - A Tutorial Approach to Simultaneous Localization and Mapping*. Cambridge: Massachusetts Institute of Technology. URL: ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf (besucht am 26. 12. 2014).

robocup.org (2014). *Rescue Robot League*. URL: robocup.org/robocup-rescue/robot-league/ (besucht am 27. 12. 2014).

SHARP (2006). *GP2D120 Optoelectronic Device*. SHARP Corporation. Osaka. URL: sharpsma.com/webfm_send/1205 (besucht am 14. 03. 2014).

ST (2014a). *32F429IDISCOVERY Discovery kit for STM32F429/439 lines*. STMicroelectronics N.V. Genf. URL: st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00094498.pdf (besucht am 07. 01. 2015).

— (2014b). *STM32F4DISCOVERY Discovery kit for STM32F407/417 line*. STMicroelectronics N.V. Genf. URL: st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00037955.pdf (besucht am 27. 12. 2014).

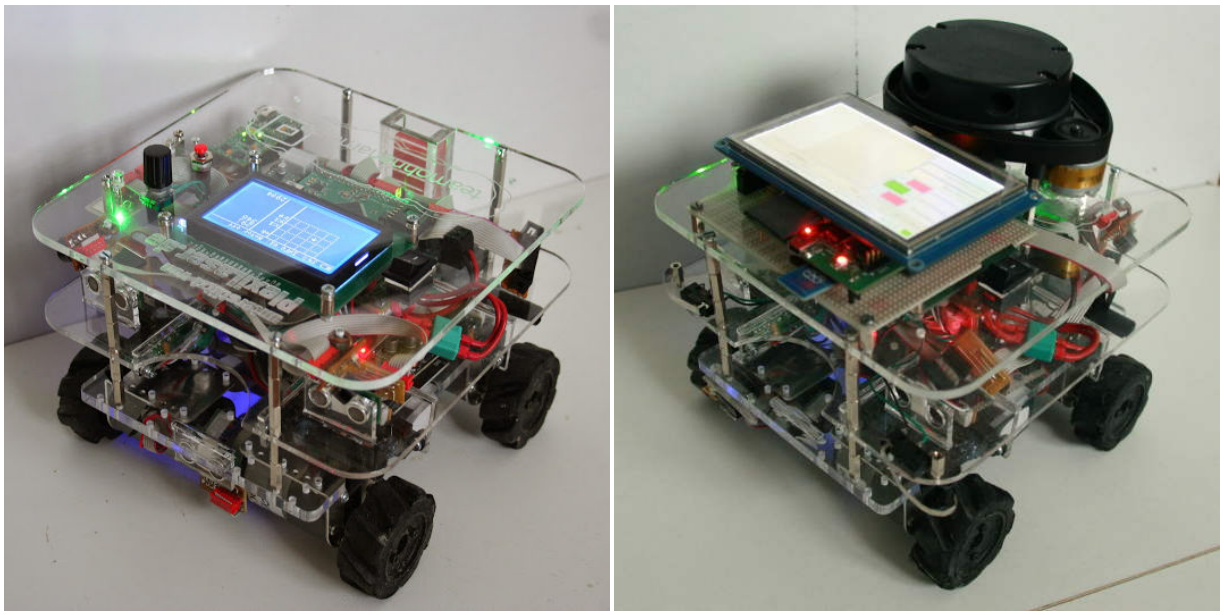
Stachniss, Cyril (2006). »Exploration and Mapping with Mobile Robots«. Dissertation. Albert-Ludwigs-Universität Freiburg. URL: informatik.uni-freiburg.de/~stachnis/pdf/stachniss06phd.pdf (besucht am 28. 12. 2014).

Steux, Bruno und Oussama El Hamzaoui (2010). *CoreSLAM: a SLAM Algorithm in less than 200 lines of C code*. Paris: Mines ParisTech - Center of Robotics. URL: researchgate.net/publication/228374722_CoreSLAM_a_SLAM_Algorithm_in_less_than_200_lines_of_C_code (besucht am 26. 12. 2014).

Wolf, Jürgen (2009). *C von A bis Z*. Bonn: Galileo Computing.

xv11hacking.wikispaces.com (2014). *LIDAR Sensor*. URL: xv11hacking.wikispaces.com/LIDAR+Sensor (besucht am 27. 12. 2014).

A Bilder



(a) RoboCup Junior Rescue B Roboter von `teamohnename.de`

(b) Roboter mit aufgesetzter Erweiterung. Hinten das LIDAR, vorne die Platine mit dem Mikrocontroller, abgedeckt durch das Display.

Abbildung 13: Verwendeter Roboter

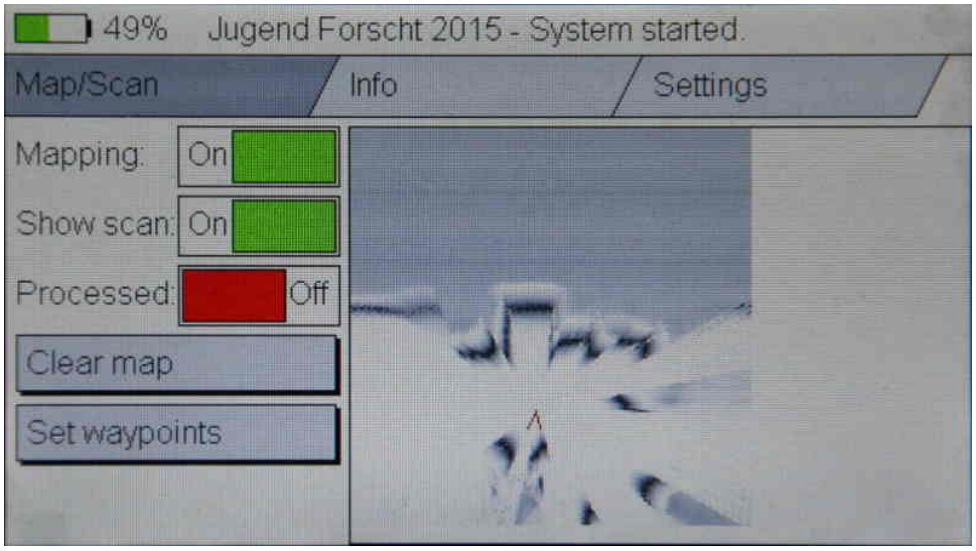


Abbildung 14: GUI auf dem Roboter

B Quellcode

B.1 Datenstrukturen

```
#define WHEELDIST                260 //Distance between two wheels
#define WHEELRADIUS               26
#define TICKSPERREV              360

#define LASERSCAN_ANGLE          360 //In degree
#define LASERSCAN_POINTS         360 //Amount of scans per LASERSCAN_ANGLE -> LASERSCAN_POINTS/LASERSCAN_ANGLE = (HAS TO BE A NATURAL NUMBER!!!!) is resolution of Laserscanner
#define LASERSCAN_NODATA           0 //Var of Laserscan if no data available

#define ODOMETER_TICKS_PER_REV   360 //Odometer ticks per revolution
#define WHEEL_RADIUS              26 //In mm

//Map
#define MAP_SIZE_X_MM            3000
#define MAP_SIZE_Y_MM            3000
#define MAP_SIZE_Z_LAYERS           1   //Amount of layers of the map
#define MAP_RESOLUTION_MM          10
#define MAP_NAVRESOLUTION_FAC       3 //Resolution of navigation cells in MAP_RESOLUTION_MM * MAP_NAVRESOLUTION_FAC mm (on each navresolution cell there come MAP_NAVRESOLUTION_FAC^2 MAP_SIZE_X_MM / MAP_RESOLUTION_MM cells)
#define MAP_NAV_SIZE_X_PX    MAP_SIZE_X_MM / (MAP_RESOLUTION_MM * MAP_NAVRESOLUTION_FAC)
#define MAP_NAV_SIZE_Y_PX    MAP_SIZE_Y_MM / (MAP_RESOLUTION_MM * MAP_NAVRESOLUTION_FAC)

#define MAP_VAR_MAX              255 //Overflow of map pixel
#define MAP_VAR_MIN                0 //Underflow of map pixel

#define IS_OBSTACLE              255 //Obstacle with 100% certainty
#define NO_OBSTACLE                0 //Obstacle with 0% certainty

//Coordinates: location in room (x, y, z)
typedef struct {
  float x;
  float y;
  int8_t z;
} slam_coordinates_t;

// Position: coordinates and orientation (angle (psi))
typedef struct {
  slam_coordinates_t coord;
  float psi;
} slam_position_t;

//Datastruct: (Pointer to) all relevant sensor/hardware information of the robot
typedef struct {
  int32_t *odo_l; //Odometer left
  int32_t *odo_r; //Odometer right
  int32_t odo_l_old; //Last odometer value after call of slam_processMovement
  int32_t odo_r_old;  //"
  int16_t lidar[LASERSCAN_POINTS]; //Laserscan data
} slam_sensordata_t;

typedef u_int8_t slam_map_pixel_t;

//Raw Map
```

```
typedef struct {
  slam_map_pixel_t px[MAP_SIZE_X_MM / MAP_RESOLUTION_MM][MAP_SIZE_Y_MM /
    MAP_RESOLUTION_MM][MAP_SIZE_Z_LAYERS];
} slam_map_t;

//Container of all SLAM information:
typedef struct {
  slam_position_t robot_pos;
  slam_sensordata_t sensordata;
  slam_map_t map;
} slam_t;
```

Listing B.1: In den folgenden Listings genutze Datenstrukturen

B.2 Monte-Carlo Suche

```
////////////////////////////////////////////////////////////////////////////////
/// \brief slam_monteCarloSearch
///   Function for correcting matching the Laserscan into the map.
/// \param slam
///   Slam container structure containing robot position and lidar data
/// \param sigma_xy
///   spreading of the values around the robot position for trying new positions
/// \param sigma_psi
///   spreading of the values around the robot orientation...
/// \param stop
///   Amount of tries
/// \return
///   Value proportional to the degree of matching

int slam_monteCarloSearch(slam_t *slam, int sigma_xy, int sigma_psi, int stop)
{
  slam_position_t currentpos; //Stores position with the current spreading
  slam_position_t bestpos; //Stores position with the best matching position
  slam_position_t lastbestpos; //Stores position with the current spreading if a better
    matching position was found. Used after 1/3 of stop!
  int currentdist; //Stores current value of degree of matching of the laserdata
  int dist_best, lastdist_best; //Stores the best and last best value of degree of
    matching of the laserdata

  currentpos = bestpos = lastbestpos = slam->robot_pos; //Initialize with robot position
  dist_best = lastdist_best = currentdist = slam_distanceScanToMap(slam, &currentpos); //
    initialize with current degree of matching

  for(int i = 0; i < stop; i++)
  {
    currentpos = lastbestpos;
    currentpos.coord.x += (float)((rand() % (sigma_xy * 2)) - sigma_xy); //Generate a
    random number between -sigma_xy and sigma_xy
    currentpos.coord.y += (float)((rand() % (sigma_xy * 2)) - sigma_xy);
    currentpos.psi += (float)((rand() % (sigma_psi * 2)) - sigma_psi);

    currentdist = slam_distanceScanToMap(slam, &currentpos); //evaluate this position

    if(currentdist > dist_best) //This position matches better than the best position
    until now
    {
      dist_best = currentdist; //Overwrite with current match
      bestpos = currentpos; //Overwrite best position
    }

    if((i > (stop / 3)) && (dist_best > lastdist_best)) //Use lastbestpos as start
    position in every new iteration from now
    {
      lastbestpos = bestpos;
      lastdist_best = dist_best;
      sigma_xy >>= 1; //Division by 2 via bitshifting (very fast): Lower spreading
      sigma_psi >>= 1;
    }
  }

  slam->robot_pos = bestpos;

  return dist_best;
```

```
}
```

Listing B.2: Monte-Carlo Suche

B.3 Bewertungsfunktion

```
////////////////////////////////////////////////////////////////////////////////
/// \brief slam_distanceScanToMap
///   Matches the Laserscan on the given position in the map
/// \param slam
///   slam container structure containing the newest lidar scan
/// \param position
///   position in the map that shall be compared by the lidar scan
/// \return
///   number that is proportional to the ambiguity (around 230000 fully matching),
///   -1 if no match found

int slam_distanceScanToMap(slam_t *slam, slam_position_t *position)
{
  float c, s, lidar_x, lidar_y;
  int i, x, y, nb_points = 0;
  float sum = 0;

  c = cosf((position->psi) * M_PI / 180); //Calculate it here, not nessesary to calculate
     in every iteration in the loop
  s = sinf((position->psi) * M_PI / 180);
  // Translate and rotate scan to robot position
  // and compute the distance
  for (i = 0; i < LASERSCAN_POINTS; i += 10) //LASERSCAN_POINTS: 360. For every 10th
    measurement.
  {
    if(slam->sensordata.lidar[i] != LASERSCAN_NODATA) //If the quality of the measurement
     is high enough and not out of range
    {
      lidar_x = (slam->sensordata.lidar[i] * sinf(i * (M_PI / 180))); //Convert from
    polar to cartesian
      lidar_y = (slam->sensordata.lidar[i] * cosf(i * (M_PI / 180)));

      x = (int)floorf((position->coord.y + c * lidar_x - s * lidar_y) / MAP_RESOLUTION_MM
     + 0.5); //Calculate the point in which the Measurement ends as seen from the robot.
      y = (int)floorf((position->coord.x + s * lidar_x + c * lidar_y) / MAP_RESOLUTION_MM
     + 0.5); //Workaround: y- and y- position has to be changed due to strange mirroring
    error...

      if((x >= 0) && (x < (MAP_SIZE_X_MM/MAP_RESOLUTION_MM)) &&
         (y >= 0) && (y < (MAP_SIZE_Y_MM/MAP_RESOLUTION_MM))) //Point lies inside the map
     size!
      {
        sum += *(&slam->map.px[0][0][slam->robot_pos.coord.z] + y * (MAP_SIZE_Y_MM /
    MAP_RESOLUTION_MM) + x); //Access array by pointer-arithmetics, add value to sum
        nb_points++;
      }
    }
  }
  if (nb_points) sum = sum * 1024 / nb_points; //Calculate all-in-all value for returning
  else sum = -1;
  return (int)sum;
}
```

Listing B.3: Bewertungsfunktion zur Bewertung der Passgenauigkeit des Laserscans in die Karte.

B.4 Verarbeitug der Odometerdaten

```
////////////////////////////////////////////////////////////////////////////////
/// \brief slam_processMovement
///   Transfers the driven encoder distance to a cartesian posisition and adds it to the
///   old robot position in the slam structure.
/// \param slam
///   slam container structure
/// \param mot
///   motor information structure

```

```
10 void slam_processMovement(slam_t *slam)
11 {
12   float dl_enc, dr_enc; //Driven distance (since last function call) in mm.
13   float dx = 0, dy = 0, dpsi = 0, dist_driven = 0;
14
15   dl_enc = (*slam->sensordata.odo_l - slam->sensordata.odo_l_old) * 2 * WHEELRADIUS *
       M_PI / TICKSPERREV; //Calculate difference driven distance in mm
16   dr_enc = (*slam->sensordata.odo_r - slam->sensordata.odo_r_old) * 2 * WHEELRADIUS *
       M_PI / TICKSPERREV;
17   slam->sensordata.odo_l_old = *slam->sensordata.odo_l; //Nessesary for next interation (
       difference)
18   slam->sensordata.odo_r_old = *slam->sensordata.odo_r;
19
20   if(fabsf(dl_enc - dr_enc) > 0) //If robot has driven a curve
21   {
22     float r = -WHEELDIST * (dl_enc + dr_enc) / (2 * (dr_enc - dl_enc)); //Helpervariable
23     dpsi = -(dr_enc - dl_enc) / WHEELDIST; //Calculate change of rotation (orientation)
       of robot
24
25     dx =  r * sinf(dpsi + (slam->robot_pos.psi * 180 / M_PI))
26         -r * sinf((slam->robot_pos.psi * 180 / M_PI)); //Calculate change of cartesian x
       in relation to last robot position
27     dy = -r * cosf(dpsi + (slam->robot_pos.psi * 180 / M_PI)) +
28           r * cosf((slam->robot_pos.psi * 180 / M_PI)); //...y
29
30     dpsi *= 180 / M_PI; //Convert radian to degree
31   }
32   else // basically going straight
33   {
34     dx = dl_enc * cosf(slam->robot_pos.psi * M_PI / 180); //No change of rotation
35     dy = dr_enc * sinf(slam->robot_pos.psi * M_PI / 180);
36   }
37
38   dist_driven = sqrtf(dx * dx + dy * dy); //Driven distance
39
40   slam->robot_pos.coord.x += dist_driven * cosf((180 - slam->robot_pos.psi + dpsi) * M_PI
       / 180); //The calculated values were given as change of position in relation to the
       last robot position. Now we have to calculate the new absolute position.
41   slam->robot_pos.coord.y += dist_driven * sinf((180 - slam->robot_pos.psi + dpsi) * M_PI
       / 180);
42   slam->robot_pos.psi += dpsi;
43 }
```

Listing B.4: Funktion zur Berechnung der Bewegung des Roboters über Odometer

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die Arbeit selbständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Arbeit, die im Wortlaut oder im wesentlichen Inhalt anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe. Verwendete Informationen aus dem Internet sind vollständig im Ausdruck zur Verfügung gestellt worden.

Jan Blumenkamp
Hunteburg, 24. März 2015