Robo exploration und autonome Maschinen SLAM

Projekt-Dokumentation

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Lörrach

von

Ugurtan Can Cetin
Marvin Obert
Linus Fischer
Robin Liebschwager
Luca Wehrle

März 2024

Kurs	TIF21
Betreuer	M.Sc. Felix Hanser

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1
1.2 Problemstellung und -abgrenzung 1
1.3 Zielsetzung 2
1.4 Aufbau der Arbeit 2

2 Grundlagen **3**

2.1 Definition SLAM 3
2.2 Deterministische Roboter und Probabilistische Roboter 3
2.3 Lokalisierung 5
2.4 Rekursive Zustandsschätzung 6
2.5 SLAM-Algorithmen 7

3 Planung und Vorbereitung **10**

3.1 Genutzte Hardware 10
3.2 Entwicklungsumgebung 11

4 Implementierung **12**

4.1 Simulation eines Roboters in einer 2D-Umgebung 12
4.2 Implementierung SLAM 14
4.3 Herausforderungen bei der Implementierung 26

5 Ergebnisse **29**

6 Ausblick **32**

Abbildungsverzeichnis

EV3 Roboter vor Modifizierung (Front) 10
EV3 Roboter vor Modifizierung (Seite) 11
EV3 Roboter nach Modifizierung (Seite) 11
Labyrinth 12
Labyrinth mit Maus Lidar entdeckt 12
Lidar mit Roboter Simulation 13
Klasse Robot 15
Methode get_distances in Klasse Robot 16
Methode calculate_detected_object_positions in Klasse ParticleFilter 17
Klasse Particle 18
Systematic Resampler Algorithmus (Pseudocode) 19
Systematic Resampler Algorithmus (Python) 19
Methode resample_particles in der Klasse ParticleFilter 19
QuadTree Prinzip 20
Methode update_map in Klasse Map 21
QuadTree Visualisierung aus MapTester 22
Methode mainloop in Klasse SLAM 23
Methode execute_phase_two in Klasse SLAM 24
Methode execute_phase_four in Klasse SLAM 25
Ergebnis Map 30
Umgebung 30
Ergebnis Map über Umgebung 30

1 Einleitung

1.1 Motivation

Die Technologie schreitet täglich weiter voran. Computer werden effizienter und mächtiger und sind Teil des normalen Alltags geworden. Auch der Verkehr wird hiervon beeinflusst. Autonome Fahrzeuge sind schon seit einigen Jahren ein Thema und selbstfahrende Fahrzeuge und autonome Maschinen spielen eine immer größter werdende Rolle. Von selbstfahrenden Autos zu Staubsaugern, die sich eigenständig im Raum navigieren und diesen währenddessen säubern. Die Grundlegende Technologie, die diesen Maschinen erlaubt sich selbstständig Fortzubewegen ist SLAM. Eine Technik, die es einem Roboter erlaubt eine Karte der Umgebung zu erstellen und sich währenddessen selbst darin zu lokalisieren.

1.2 Problemstellung und -abgrenzung

Das Problem, welches hier zu lösen ist, ist die Lokalisierung eines Roboters in einer unbekannten Umgebung, während es von dieser unbekannten Umgebung eine Karte erstellt. Hierfür wird die Technik SLAM verwendet, um während der Lokalisierung eine Karte zu erstellen. Ein wichtiger Teil hierbei ist es, den Roboter effizient zu navigieren, sodass eine Karte erstellt werden kann. Aufgrund beschränkter Hardwareressourcen, wird ein leichtgängiger SLAM implementiert und es wird nur mit einem Lidar gearbeitet.

1.3 Zielsetzung

Das Ziel dieses Projekts ist die Entwicklung eines SLAM Roboters. Der Roboter der verwendet wird, ist ein EV3 Brick von Lego Mindstorms und hat begrenzte Ressourcen. Es soll in der Lage sein, eine korrekte Karte der Umgebung zu erstellen und währenddessen sich selbst in der Umgebung zu lokalisieren, um die Karte erweitern zu können. Die Karte soll am Ende gespeichert werden, sodass die Karte nach Durchlauf des SLAM-Codes eingelesen werden kann und evaluiert werden kann.

1.4 Aufbau der Arbeit

Zunächst werden die Grundlagen des Themas besprochen. Hierbei wird besonders auf verschiedene SLAM-Techniken eingegangen. Darauffolgend werden die verfügbare Hardware und die Entwicklungsumgebung dokumentiert. Darin wird ebenfalls besprochen welchen Roboter wir haben und welche Sensorik wir genutzt haben. Im vierten Kapitel ist der Fokus die Implementierung. Hier wird geklärt, welche SLAM-Technik gewählt wurde, und wie es implementiert wurde. Es wird sehr detailliert auf die einzelnen Programmklassen eingegangen und es wird mit Abbildungen der Quellcodes die Algorithmen erklärt. Daraufhin werden auch die Herausforderungen die während der Implementierung aufgetreten sind. Manche dieser Herausforderungen wurden auch gelöst, und wie das gelöst wurde, wird ebenfalls erklärt. Im fünften Kapitel werden die Ergebnisse des Projektes zusammengefasst und daraufhin wird im Kapitel Äusblick", potenziale Verbesserungen dokumentiert.

2 Grundlagen

2.1 Definition SLAM

imultaneous localization and mapping (kurz: SLAM) ist eine Problem, in der das Ziel ist, dass ein Roboter in einer unbekannten Umgebung sich autonom navigiert und simultan eine Karte der Umgebung erstellt [1] . SLAM besteht aus zwei Hauptaufgaben. Der Lokalisierung, welches das Abschätzen der Position und Orientierung des Roboters relativ zur Karte ist, und das Mapping, was die Konstruktion einer Karte der Umgebung basierend auf Sensordaten ist.

2.2 Deterministische Roboter und Probabilistische Roboter

In der Robotik ist die Unterscheidung zwischen deterministischen und Probabilistischen Ansätzen entscheidend. Während deterministische Roboter nach festen Regeln und Algorithmen gehen [2], navigieren probabilistische Roboter durch die Ungewissheiten der realen Welt mit der Wahrscheinlichkeitstheorie [3]. Es gibt hierbei verschiedene Gründe, warum SLAM nicht deterministisch ist, und diese Gründe unterstreichen auch die Herausforderungen bei der Implementierung von SLAM.

1 (Thrun, Burgard und Fox, 2005, S. 245)
2 (*Deterministic algorithm*, 2021)
3 (Thrun et al., 2005, S. 3)

Ungewissheit in Sensordaten

Sensoren wie LiDAR oder Kameras verursachen Ungenauigkeiten in den Messdaten. Die Ungenauigkeiten sind Resultate von Sensorrauschen, Verdeckung oder Umgebungsschwankungen[1]. Dadurch sind die Daten, die von den Sensoren erhalten werden probabilistisch, weshalb die Wahrnehmung der Umgebung zu Unsicherheiten führt.

Unsicherheit bei der Bewegung

Die Roboterbewegung ist ebenfalls von Ungenauigkeiten geprägt, wie zum Beispiel durch Ausrutschen der Räder oder einem unebenem Gelände. Deswegen kann der Pfad des Roboters durch die Umgebung nicht präzise vorhergesagt werden, was zu Unsicherheiten bei der Positionsschätzung verursacht[2].

Mehrdeutigkeit der Sensordaten

Die Daten, die von den Sensoren erfasst werden repräsentieren Merkmale in der Umgebung und in Umgebungen mit ähnlichen oder mehrdeutigen Merkmalen ist die korrekte Zuordnung zwischen Messungen und Merkmalen problematisch.

Mehrdeutigkeit bei der Erstellung der Karte

Die Erstellung der Karte benötigt die Lösung des Mehrdeutigkeitsproblems der Sensordaten und eine Schätzung der räumlichen Beziehungen zwischen erkannten Features. Die Unsicherheit, die bei der Sensormessung und der Zuordnung der Daten erzeugt wird, vergrößert die Mehrdeutigkeit bei der Erstellung der Karte, weshalb es zu einem probabilistischen Schätzungsproblem wird.
Das ideale Konzept der deterministischen Roboter bietet zwar Einfachheit und Sicherheit, aber die Komplexität der realen Umgebung erfordert einen probabilistischen Ansatz für SLAM aufgrund der Unsicherheit der Umgebung.

[1] (Thrun et al., 2005, S. 2)
[2] (Thrun et al., 2005, p. 2)

2.3 Lokalisierung

Lokalisierung ist die Bestimmung des Ortes, an dem sich der Roboter befindet. Hierfür gibt es verschiedene Begriffe, welche geklärt werden müssen, um zu verstehen, wie der Roboter Lokalisierung betreibt.

Dead Reckoning

Dead Reckoning ist ein sehr interessantes Prozedere, in dem die aktuelle Position des Roboters geschätzt. Hierfür wird mithilfe der vorherigen Position, der Geschwindigkeit und der Richtung die aktuelle Position geschätzt[1]. Der Vorteil hierbei ist es, dass hierfür keine Referenz mit Objekten der Außenwelt benötigt wird. Also wird nur mithilfe von On-Board Encoder Daten die Position abgeschätzt.

Odometrie-basierte Lokalisierung

Odometrie ist ein System, welches genutzt wird, um die Position eines Roboters vorherzusagen, während es sich bewegt. Hierfür gibt es verschiedene Arten, um die Position abzuschätzen. Gängig sind die Kombination von Sensordaten und Rad-Encoder Daten[2]. Hierfür wird mithilfe des Umfangs des Rads und der Drehwinkel die Distanz berechnet, die gerade zurückgelegt wurde. Daraufhin werden Sensordaten von zum Beispiel einem Lidar oder Infrarotsensor genutzt, um die Position des Roboters zu korrigieren[3].
Wenn jedoch der Roboter keine Geschwindigkeitsänderungen hat, dann kann man mit Konstanter Geschwindigkeit die Hardware Odometrie rauslassen, und die Lokalisierung nur mit den Sensordaten abschätzen[4]. Wichtig ist hierbei, dass berücksichtigt wird, dass verschiedene Sensormessungen zu verschiedenen Zeitpunkten aufgenommen werden, und hier durch die Daten nicht direkt zusammenschließen. Wenn man dies berücksichtigt, kann man die Position ohne Verzerrungen abschätzen.

1 (Pao, 2021)
2 (Stachniss, 2022, 1:48)
3 (Stachniss, 2022, 1:29)
4 (Stachniss, 2022, 1:59)

2.4 Rekursive Zustandsschätzung

Rekursive Zustandsschätzung ist ein wichtiges Konzept in der Robotik, besonders in SLAM. Es liefert einen systematischen Weg um den aktuellen Zustands des Roboters basierend auf Sensordaten zu schätzen[1]. Zum Zustand selbst gehören nicht nur die Position des Roboters, sondern auch die Orientierung.

Bayes Filter

Der Bayes Filter ist ein probabilistisches Framework, in dem die Überzeugung des aktuellen Zustands basierend auf neuen Sensordaten und Kontrollinputs aktualisiert wird[2].
Durch iterative Anwendung des Bayes Theorems, verbessert der Filter die Vorhersage des Zustandes durch das Einbeziehen von neuen Informationen durch Sensoren und Kontrollinputs. So wird mit der Zeit die Vorhersage verbessert.

Kalman Filter und Extended Kalman Filter

Der Kalman Filter ist eine probabilistische Methode, um den Zustand zu schätzen. Es besteht aus zwei Schritten, der Vorhersage und Korrektur. In der Vorhersage wird probiert basierend auf Kontrollinputs, den Zustand vorherzusagen, und die Korrektur nutzt die Sensordaten, um potenzielle Fehler zu korrigieren[3].
Das Besondere beim Kalman Filter ist, dass davon ausgegangen wird, dass alle Unsicherheiten und Rauschobjekte Normal verteilt sind (Gaußsche Verteilung) und dass alle Modelle linear sind. Daraus folgt, dass die Modelle, die zur Vorhersage und zur Korrektur genutzt werden, linear sind. In einer linearen Welt wäre der Kalman Filter optimal, aber in einer nicht-linearen Welt ist dieser nicht mehr nutzbar. Hierfür kann man den Extended Kalman Filter verwenden, der mithilfe von Taylor Approximation die nicht-linearen Modelle in Lineare umwandelt[4]. Diese Linearisierung muss nach jedem Schritt durchgeführt werden.

1 (Thrun et al., 2005, p. 9)
2 (Stachniss, 2020*a*, 0:23)
3 (Stachniss, 2020*b*, 0:48)
4 (Thrun et al., 2005, p. 48)

Partikelfilter

Der Partikel Filter, auch Monte Carlo Lokalisierung, ist ähnlich zum Kalman Filter. Es wird auch verwendet, um den Zustand des Roboters zu schätzen, funktioniert jedoch auch bei nicht-linearen Welten und geht nicht davon aus, dass das Rauschen und die Unsicherheiten Normalverteilt sind. Es nutzt eine Menge an Partikeln, welche Hypothesen des Zustands sind[1]. Dies bedeutet, bei einer Menge von 500 Partikeln, hat das System 500 Hypothesen, die den aktuellen Zustand abbilden könnten.
Im Fall von SLAM, könnte ein Partikel die Koordinaten und die Orientierung des Roboters repräsentieren. Zunächst wird ein Zustand vorhergesagt, in dem Kontrollinputs genutzt werden. Auf diese Vorhersage wird dann Rauschen angewendet. Dann wird der Zustand mithilfe von Sensordaten korrigiert. Die Wahrscheinlichkeit, dass der Roboter sich nun in diesem Zustand befindet wird berechnet, und als Gewicht dem Partikel zugewiesen.
Je höher das Gewicht, desto höher ist die Wahrscheinlichkeit, dass der Partikel den tatsächlichen Zustand repräsentiert. Daraufhin wird ein Resampling durchgeführt, in der die Partikel mit hohem Gewicht repliziert werden, und Partikel mit niedrigem Gewicht nicht mehr berücksichtigt werden.

2.5 SLAM-Algorithmen

Nun folgen verschiedene SLAM-Algorithmen, die auf verschiedenen eben besprochenen Prinzipien beruhen.

EKF-SLAM

EKF-SLAM nutzt die Erweiterung des Kalman Filters, um die Roboterpose und die Umgebungskarte abzuschätzen[2]. Hierfür wird zunächst ein Zustandsraum definiert, welcher die Pose des Roboters, also Koordinaten und Blickrichtung, sowie die Koordinaten von Landmarks. Daraufhin wird die Bewegung des Roboters vorhergesagt und der Zustand wird dann aktualisiert. Die Modelle sind nicht linear, weshalb diese linearisiert werden müssen, um den (Extended) Kalman Filter anwenden zu können. Daraufhin werden mithilfe von Sensoren Landmarks identifiziert und dann genutzt, um den Zustand des Roboters zu aktualisieren.

[1] (Stachniss, 2020*c*, 7:25)
[2] (Ahmed, 2023, Filter Based SLAM - Extended Kalman Filter)

Fast-SLAM

FAST-SLAM ist ein Algorithmus, welcher Partikelfilter anwendet, um die Karte zu erstellen und den Roboter zu lokalisieren. Im Gegensatz zum EKF-SLAM wird nicht nur eine Schätzung der Pose des Roboters gemacht, sondern viele verschiedene Schätzungen, die als Partikel repräsentiert werden[1]. Dies erlaubt dem Algorithmus mit hoher Genauigkeit zu arbeiten, insbesondere in komplexen Umgebungen. Zunächst werden die Partikel generiert, daraufhin wird eine Vorhersage gemacht, die als zukünftige Bewegung des Roboters für jedes einzelne Partikel angewendet wird. Daraufhin werden für jedes Partikel mithilfe von Umgebungsmessungen die Gewichte angepasst, die angeben, wie hoch die Wahrscheinlichkeit ist, dass das Partikel die wirkliche Pose des Roboters widerspiegelt[2]. Dann wird ein Resampling durchgeführt, bei dem die Partikel mit höherer Gewichtung eine neue Generation erzeugen, und die Partikel mit niedrigen Gewichten ignoriert werden.

Graph-SLAM

Graph-SLAM behandelt das SLAM-Problem als ein Graph Problem[3]. Das bedeutet, dass die Positionen als Knoten dargestellt werden und die Kanten die Bewegungen des Roboters zwischen Knoten. Zunächst wird der Zustand des Systems mit einem gerichteten Graphen repräsentiert. Daraufhin wird der Graph in einzelne Faktoren geteilt, die die Beziehung zwischen den Roboterpositionen darstellen. Dann wird eine Pose-graph Optimization durchgeführt. Hierbei wird versucht die Fehler der Posen Schätzungen zu minimieren. Die Fehler werden erzeugt durch Odometrie oder Loop-Closure[4]. Wenn ein Roboter an einer Position ist, an der es vorher schon gewesen ist, kann es bei Erkennung dieses Loops versuchen, die Gesamten Pose-Nodes zu optimieren.

1 (Stachniss, 2013, 0:30)
2 (Stachniss, 2013, 29:00)
3 (Ahmed, 2023, Graph Based SLAM)
4 (Ahmed, 2023, Graph Based SLAM)

Grid-SLAM

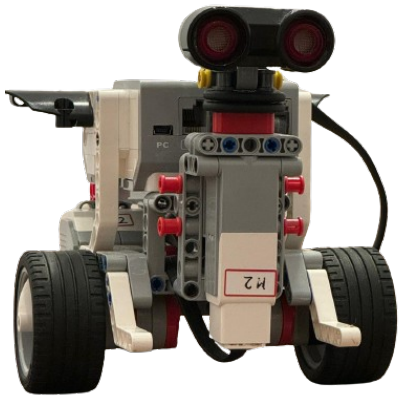
Das Grid-SLAM ist eines der intuitiven Algorithmen. Hierbei wird die Umgebung als ein Raster gespeichert. Das Raster ist in Zellen mit konstanter Größe eingeteilt, wobei ein Wert in der Zelle ist. Der Wert repräsentiert, ob diese Zelle ein Hindernis hat oder nicht. Mit direkten Sensordaten wie zum Beispiel mit einem LiDAR ist es recht unkompliziert Grid-SLAM zu implementieren. Jedoch ist Grid-SLAM nicht sehr genau, und kann zu hohem Speicheraufwand führen[1].

[1] (*SLAM - Grid-based FastSLAM Introduction to Mobile Robotics*, n.d., Folie 11)

3 Planung und Vorbereitung

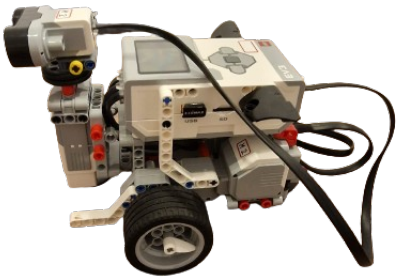
3.1 Genutzte Hardware

Die Hardware, welche uns für dieses Projekt zur Verfügung gestellt wurde, sind 2 Lego Mindstorms EV3 Bricks. Diese haben jeweils drei Motoren, zwei für die Reifen und einen weiteren, an dem ein Sensor befestigt wurde. Diese Hardware ist definitiv ausreichend, um einen SLAM-Roboter zu implementieren, weil wir Möglichkeiten zur Rotationskontrolle und Distanzmessung haben, welche essenziell für SLAM sind. Nichtsdestotrotz hat der Brick ein 300MHz ARM9 Prozessor und 64MB RAM18, weshalb die Performance niedrig ist und das auch bei der Implementierung berücksichtigt werden muss.
Der Roboter der uns zur Verfügung gestellt wurde, war von den vorherigen Studenten aufgebaut, wie in Abbildung a.



(a) EV3 Roboter vor Modifizierung (Front)

Dieser Aufbau ist zwar voll-funktionsfähig, aber wir haben trotzdem eine kleine Änderung durchgeführt, welche in Abbildungen b und c deutlich werden. Wir haben den "Hals"des Roboters erweitert, sodass wir an die Anschlüsse vom Roboter kommen können, ohne den Roboter jedes Mal auseinander bauen zu müssen. Daher, dass wir den Code nach jeder Änderung am Computer auf den Roboter geladen haben, war es sehr nützlich, wenn das Kabel dauerhaft drinnen war, bis wir einen größeren Durchlauf testeten.



(b) EV3 Roboter vor Modifizierung (Seite)

(c) EV3 Roboter nach Modifizierung (Seite)

3.2 Entwicklungsumgebung

Das Projekt will vollständig in Python implementiert sein. Für die Kontrolle der Lego Mindstorms EV3 nutzen wir die Library ev3dev2 und die Library pybricks. Diese ermöglichen uns mit den internen Sensoren des Bricks zu arbeiten, die Motoren genau zu steuern und Signale an den Lidar oder an den Infrarotsensor zu senden mithilfe der dazugehörigen Portnummern. Der Code selbst wird mit- hilfe von Git versioniert.

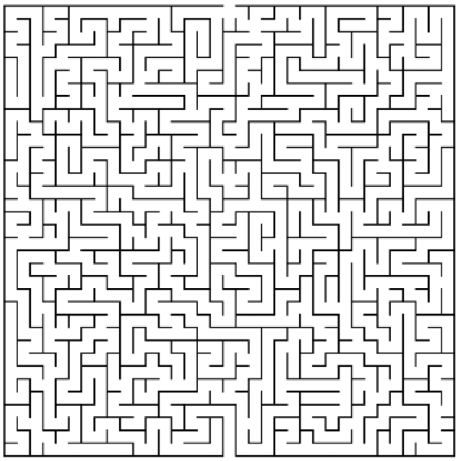
4 Implementierung

4.1 Simulation eines Roboters in einer 2D-Umgebung

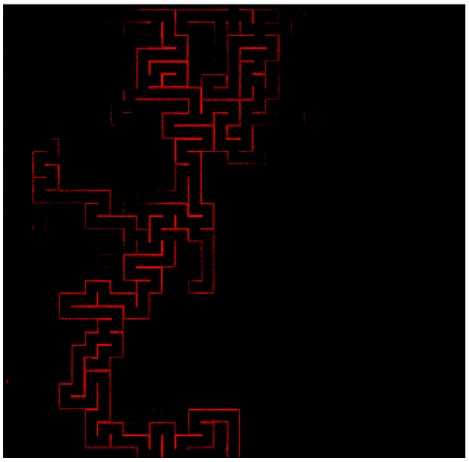
Zunächst war ein Problem, welches wir hatten, dass nur zwei Geräte verfügbar waren und wir alles Online aus machen mussten. Hierfür haben wir verschiedene Simulationen des Lidar Prinzips implementiert.

Lidar Prinzip Simulation

Ein Programm simuliert das Prinzip vom Lidar. Zunächst wird bei jeder Bewegung der Computermaus geschaut, wo im Umkreis Hindernisse sind. Dazu werden Lidar Signale abgeschickt. Wenn diese an ein Objekt treffen, wird dort ein roter Punkt gesetzt. Es muss ein Bild beigefügt werden, welches mit schwarzen Pixeln diese Hindernisse widergespiegelt. Alles wird in einem pygame Fenster angezeigt.



(a) Labyrinth



(b) Labyrinth mit Maus Lidar entdeckt

In Abbildung a sieht man, wie die Karte aussah, bevor Sie in das Programm geladen wird. Das Programm zeigt anfangs nur einen schwarzen Bildschirm, und nimmt Hindernisse um die Maus während der Laufzeit wahr und setzt die Anstoßpunkte des virtuellen Lidars. Dadurch sieht dann das pygame Fenster aus wie in Abbildung b. Dieser Prototyp war sehr hilfreich, um zu verstehen, wie mithilfe der Ausgangs- position, des Winkels und einer Distanz den Punkt eines Hindernisses berechnen können.

Robot Lidar Simulation

Ein weiteres Programm, welches entwickelt wurde um in der Implementierung des Lidar Roboters zu helfen ist eine Simulation des Roboters. Die Simulation selbst ist zwar sehr primitiv, aber war hilfreich bei der Entwicklung, besonders für die Gruppenmitglieder, die den Code nicht direkt am EV3 Brick testen konnten. Dieser Code ermöglicht es mit Befehlen an die Roboter Klasse den Roboter zu steuern und Daten eines virtuellen Lidars abzufragen. Die Steuerung des Roboters ist zu demonstrativen Zwecken auch mit der Tastatur möglich. Hierbei kann man die Tasten W und S nutzen, um nach vorne bzw. nach hinten zu laufen, je nach Orientierung. Der Roboter kann mithilfe der Tasten D und A rotiert werden. Jedoch sind die Aktionen nicht kontinuierlich, sondern schrittweise. Außerdem kann mithilfe der Leertaste ein simuliertes Lidar genutzt werden um schwarze Pixel (Hindernisse) zu erkennen. Der Aufschlagort wird dann rot markiert (siehe Abb. c)

(c) Lidar mit Roboter Simulation

4.2 Implementierung SLAM

Für unser Projekt haben wir uns grundsätzlich für SLAM mit Partikelfiltern entschieden. Partikelfilter sind skalierbar und kann auf nicht-lineare Systeme angewendet werden. Der Partikelfilter ist jedoch rechenaufwändig, da man bestimmte Schritte für jeden einzelnen Partikel durchführen muss. Aber wir fanden das Prinzip und die Arbeitsweise des Partikelfilter so interessant, dass wir es entwickeln wollten und uns trotz der Nachteile dafür entschieden.

Planung der Implementierung

Nachdem wir uns entschiedene hatten, einen Partikelfilter anzuwenden, mussten wir überlegen, welche Komponenten unsere Anwendung haben muss. Eins war deutlich, wir benötigten eine Zentralsteuerung. Eine Main-Klasse, von der aus der SLAM-Vorgang initialisiert und durchgeführt wird. Nach Überlegungen sind wir auf den Entschluss gekommen, dass wir insgesamt 5 Hauptklassen haben werden. Eine Klasse für die generelle Steuerung des Algorithmus, in der sich auch die main-Methode befindet, die zum Start ausgeführt wird. Dann eine Klasse welche den Partikelfilter repräsentiert. Hier werden die Schritte die während dem Partikelfilter durchgeführt werden gelagert. Eine weitere Klasse wäre für die Karte. Diese Klasse besitzt dann Methoden zur Ausgabe, Erweiterung und Aktualisierung der Karte, die zu jeder Zeit aufgerufen werden können. Zum Schluss haben wir noch zwei Klassen geplant, eine für die Roboter Steuerung und eine Klasse für die Steuerung des Sensors. Mit diesem Plan sind wir an die Entwicklung gegangen. Nun folgt für jede Klasse eine Erklärung was dort durchgeführt wird und teilweise wird erläutert, wie diese Schritte durchgeführt werden.

Interface Sensorik und Robot

Um den Roboter anzusteuern wird eine Klasse namens **Robot** erstellt (siehe Abb. d). Diese besteht aus insgesamt 3 Methoden. Diese Methoden rotieren den Roboter, bewegen den Roboter oder beenden das ganze Programm.

```python
class Robot:

    def __init__(self, max_minutes=5):
        self.tank = MoveTank(OUTPUT_A, OUTPUT_B)

        self.timestamp = None
        self.minutes = max_minutes

    def move_forward(self):
        self.tank.on_for_seconds(SpeedPercent(15), SpeedPercent(15), 1)

    def rotate_robot(self, angle):
        self.timestamp = time.perf_counter() if self.timestamp is None else self.timestamp

        if angle > 0:
            self.tank.on_for_degrees(50, -50, angle)
        else:
            self.tank.on_for_degrees(-50, 50, abs(angle))

    def time_limit_exceeded(self):
        return (time.perf_counter() - self.timestamp) >= 60 * self.minutes
```

(d) Klasse Robot

Der Code wird terminiert, wenn die Methode *time_limit_exceeded(self)* true zurückgegeben. Wann dies geschieht, wir bei der Initialisierung von Robot definiert. Der default Wert ist hierbei 5 Minuten, aber dies kann im Code bei der Erstellung einer Robot Instanz angepasst werden.

rotate_robot(self, angle) rotiert den Roboter um den angegebenen Winkel angle. Der angle gibt in diesem Fall die Richtung an, in der die Distanz am größten ist. Die Distanzen werden in der Klasse **Sensor** gemessen.

```python
def get_distances(self):
    """
    Returns the distances from the last scan
    """
    distances = []

    # Rotate medium_motor to start position
    self.medium_motor.on_to_position(SpeedPercent(10), 0)

    # Rotate the sensor 36 times by 10 degrees
    for _ in range(0, 360, 10):
        self.medium_motor.on_for_degrees(SpeedPercent(10), 10)
        distance = self.ir.proximity
        distances.append(distance)

    # Return medium_motor to start position
    self.medium_motor.on_for_degrees(SpeedPercent(10), -360)

    self.distances = distances

    max_index = distances.index(max(distances)) * 10
    self.angle = max_index

    return distances
```

(e) Methode get_distances in Klasse Robot

Der Sensor hat nur eine Methode, und zwar *get_distances(self)* (siehe Abb. e). Dieser gibt eine Liste mit allen Distanzwerten, die gemessen wurden zurück. Zunächst wird hierfür der Motor, an dem der Sensor befestigt ist, zur Ursprungsposition rotiert. Dann wird in 10° Schritten einmal um den Roboter herum gescannt, und jede gemessene Distanz wird in die Liste aufgenommen. Bevor die Liste jedoch zurück gegeben wird, wird der Index bestimmt, an dem die Distanz in der Liste am größten ist. Dieser Index wird mit 10 multipliziert und in der Umgebungsvariable angle gespeichert. Diesen Winkel kann dann mit einer Instanz der **Sensor**-Klasse abfragen.

Partikelfilter

Der Partikelfilter besteht aus 3 grundlegend Schritten.

- Sampling
  Beim Sampling werden Partikel generiert. Diese Partikel stellen Hypothesen über den Roboter Zustand dar.

- Gewichtung
  Hier wird jedem Partikel ein Gewicht zugeordnet, welche darstellt, wie sehr das Partikel dem Originalzustand entspricht.

- Resampling
  Das Resampling sorgt dafür, dass Partikel mit hohem Gewicht bei Bestehen und dupliziert werden, während Partikel mit einem niedrigen Gewicht aussortiert werden.

Der Partikelfilter, den wir implementiert haben besteht aus verschiedenen Methoden, die diese drei Schritte widerspiegeln. Die genauen Schritte für die Durchführung des Partikelfilter wurden aus Vorlesungsnotizen entnommen, von der CMU School of Computer Science in Pittsburgh Pennsylvania [1]. Die Notizen resultieren aus einer Vorlesung über statistische Techniken in Robotik von Drew Bagnell. Die Partikelgenerierung wird einmal am Anfang durchgeführt. Hierfür werden zufällige Koordinaten und Winkel erzeugt. Jedes Partikel hat initial das Gewicht $1/$Anzahl Partikel. Bei einer Anzahl von 100 Partikeln also 0.01. Danach werden die Partikel mit der Methode *update_particle(self, particle, time_duration, angle, sensor_data, map, noise)* aktualisiert. Zunächst werden die Partikel weiter entwickelt. Hierfür wird mithilfe dem Winkel und der *time_duration* geschätzt, wo sich der Partikel befinden würde, wenn es sich in die Richtung des Winkels bewegt. Danach wird mithilfe der *sensor_data* die Positionen von Objekten in der Umgebung bestimmt (siehe Abb. f).

```python
def calculate_detected_object_positions(self, sensor_data, pose: Pose):
    """
    Generate the positions of objects detected by the sensor based on the sensor data and the pose of the robot.
    """
    measurements = np.array(sensor_data)

    angle_data = np.arange(0, 360, 10)
    object_positions = []
    for i in range(len(measurements)):
        angle = math.radians(angle_data[i] + pose.angle)
        delta_x, delta_y = int(measurements[i] * math.sin(angle)), int(measurements[i] * math.cos(angle))
        object_x, object_y = int(delta_x + pose.x), int(delta_y + pose.y)
        object_positions.append((object_x, object_y))
    return object_positions
```

(f) Methode calculate_detected_object_positions in Klasse ParticleFilter

[1] (Seyfarth, Batts und Bagnell, n.d.)

Hierfür wird durch die Sensor-Daten iteriert und von der Position jedes Partikels berechnet, wo die Objekte relativ zum Partikel sind. Die Partikel Position wird mithilfe von dem Parameter pose mitgegeben. Wir haben zur Repräsentation eines Partikels ein Value Object eingerichtet namens **Particle** (siehe Abb. g).

```
class Particle:

    def __init__(self, x, y, orientation, weight):
        self.pose = Pose(x, y, orientation)
        self.weight = weight

    def get_as_list(self):
        return [self.pose.x, self.pose.y, self.pose.angle, self.weight]
```

(g) Klasse Particle

Die Methode *get_as_list()* ist dafür da um die Particle-Daten (x-Koordinate, y-Koordinate, Winkel und Gewicht) als ein Listelement zurück zugeben, weil dies für die Durchschnittsberechnung in diesem Format sein muss. Nachdem die Partikel aktualisiert wurden, in dem die neuen Gewichte für die Partikel berechnet werden[1], werden die Partikel normalisiert. Dies ist wichtig für das nun folgende **Resampling**. Wenn die Partikel nicht normalisiert werden, werden Partikel mit größeren Absolutwerten überrepräsentiert, ohne die relative Wahrscheinlichkeit zu betrachten.
Zum resamplen gibt es verschiedene Algorithmen. In unserem Fall arbeitet unser Partikelfilter mit einem **Systematic Resampler**. Diese sind einfach zu implementieren und liefern sehr robuste Ergebnisse und werden empfohlen, wenn die Partikelanzahl statisch ist. Das heißt, bei n Startpartikeln, wird es immer nur n Partikel geben, wie das bei uns der Fall ist. Zunächst muss für Systematic Resampling die Kumulative Verteilung der Gewichte berechnet werden. Hierfür werden die normalisierten Gewichte kumulativ summiert, und die Verteilung ist dann die Wahrscheinlichkeit der Resampling Auswahl von jedem Partikel. Es werden dann gleichmäßig-große Mengen an Gewichten gebildet, die einzeln durchgearbeitet werden. Für alle Startpunkte werden nun die Partikel gesucht, dessen kumulative Gewichtung den Startpunkt überschreitet. Dieser Partikel ist dann Teil der neuen Partikel. So werden die Partikel mit hohem Gewicht mit einer höheren Wahrscheinlichkeit mehrmals gewählt.

[1] (*Particle Filtering 1.1 Particle Filtering Summary*, n.d.)

$xx[] = SR(x, w, N)$
$j = 0,\ accumW = w[j]$
$u = rand()/N$
for $i = 0..(N-1)$
    while $accumW < u$
        $j = j + 1$
        $accumW = accumW + w[j]$
    end
    $xx[i] = x[j]$
    $u = u + 1/N$
end

(h) Systematic Resampler Algorithmus (Pseudo-code)

```python
def SR(x, w, N):
    xx = np.zeros(N)
    j = 0
    accumW = w[j]
    u = np.random.rand() / N
    for i in range(N):
        while accumW < u:
            j = j + 1
            accumW = accumW + w[j]
        xx[i] = x[j]
        u = u + 1/N
    return xx
```

(i) Systematic Resampler Algorithmus (Python)

```python
def resample_particles(self):
    weights = [particle.weight for particle in self.particles]
    cumulative_sum = np.cumsum(weights)
    step = 1.0 / self.num_particles
    uniform_distribution = np.random.uniform(0, step, self.num_particles) + np.arange(self.num_particles) * step
    indexes = np.searchsorted(cumulative_sum, uniform_distribution, side='right')
    return indexes
```

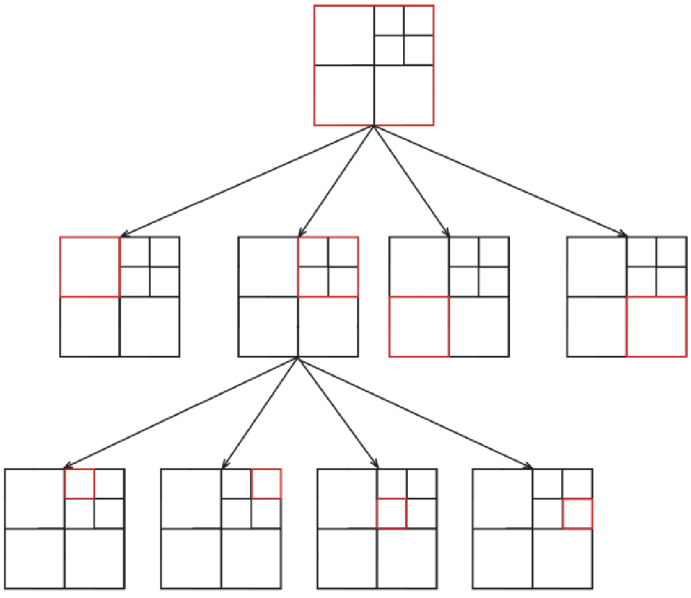
(j) Methode resample_particles in der Klasse ParticleFilter

In Abbildung h[1] sieht man den Algorithmus Schritt für Schritt in Pseudo-Code und und in Abbildung i in Python. Jedoch haben wir diesen Algorithmus abgeändert (siehe Abb. j). Wir nutzen numpy Methoden um die Verteilung zu erstellen und die Indizes zu finden, statt zwei Schleifen. Bei kleinen Listen ist die Geschwindigkeit mit np.random.uniform() ähnlich wie bei den Schleifen. Jedoch ist es schneller bei größeren Datenmengen. Außerdem ist die Generierung von zufälligen Zahlen mit Vektorisierung schneller als mit der normalen random Methode von Python [2]. Die Methode np.searchsorted wird genutzt, um die Indizes zu finden, mit denen die Stellen gefunden werden, an denen Elemente der Gleichverteilung in die kumulative Summe eingefügt werden sollten, um die Reihenfolge beizubehalten.

[1] (Lim, Yeong, Su, Shithil, Chik, Duan und Chin, 2019)
[2] (Turner-Trauring, 2022)

Mapping

Der Code, welcher genutzt wird um die Karte der Umgebung zu erzeugen, wird im Ordner map gespeichert. Dort befindet sich die Datei *map.py*, die den Hauptcode der Map hat. Zunächst haben wir uns entschieden die Map mit einer Matrix darzustellen. Hierfür wird bei der Objektinitialisierung die *grid_width* mitgeteilt. Hier kommt auch der erste Grund, weshalb wir am Ende nicht bei der 2x2 Matrix geblieben sind. Wenn die Matrix zu klein wurde, musste diese erweitert werden. Dies konnte mit wenig Code erzielt werden, aber problematisch waren die leeren Elemente. Die Bereiche, die nicht mit einem Objekt besetzt waren, mussten ebenfalls gespeichert werden, wodurch man viel Speicher für die Grids aufwenden musste. Bei der Erweiterung dieser Elemente, konnte man mithilfe von *np.pad()* ein Padding hinzufügen, sodass um die ursprüngliche Matrix neue Elemente hinzugefügt werden. Nichtsdestotrotz hat man eine hohe Redundanz und Speicherverbrauch. Um dieses Problem zu lösen, haben wir recherchiert und sind auf Quadtrees gestoßen. QuadTrees bieten eine Möglichkeit um ein Grid als ein Baum darzustellen. Hierfür wird für eine Map ein Root definiert. Dieser Root hat 4 Unterknoten. Diese 4 Knoten stellen die vier aktuell sichtbaren Quadranten dar. Jedes dieser Quadranten, kann ebenfalls in Quadranten geteilt werden (siehe Abb. k).



(k) QuadTree Prinzip

Die Map wird aktualisiert, in dem es mithilfe der Distanzdaten des Sensors und der aktuellen Roboterposition, Objektpositionen berechnet (siehe Abb. l). Wenn dieser Punkt schon im

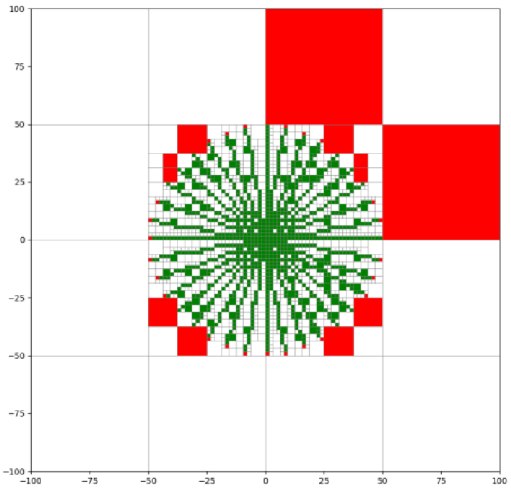
QuadTree vorhanden ist, wird es auf Occupied gesetzt und es werden die Punkte, die zwischen dem Roboter und dem gerade erkannten Hindernis sind, als *free-points* zum Baum hinzugefügt. Wenn der Punkt nicht im Baum ist, muss der Baum um den Punkt erweitert werden. Ein Punkt, oder eher eine Position im QuadTree hat 3 Zustände. Entweder ist sie FREE, OCCUPIED oder UNDISCOVERED.

```
def update_map(self, distance_data, robot_position):

    datapoints = len(distance_data)
    for i in range(datapoints):
        angle = math.radians(360 * i / datapoints + robot_position.angle)
        delta_x = int(distance_data[i] * math.sin(angle))
        delta_y = int(distance_data[i] * math.cos(angle))

        object_x = int(round(delta_x + robot_position.x))
        object_y = int(round(delta_y + robot_position.y))
        if object_x >= 0 and object_x < self.grid_width and object_y >= 0 and object_y < self.grid_width:
            self.map[object_x, object_y] = 1

        if self.tree.contains_point((object_x, object_y)):
            self.tree.insert((object_x, object_y), data=Observation.OCCUPIED)
            self._add_free_points(object_x, object_y, robot_position)
            continue

        half_width = self.grid_width / 2

        if object_x > self.tree.center.x + half_width and object_y > self.tree.center.y + half_width:
            self.extend_map(QuadPosition.LOWER_LEFT)
        elif object_x > self.tree.center.x + half_width and object_y < self.tree.center.y + half_width:
            self.extend_map(QuadPosition.UPPER_LEFT)
        elif object_x < self.tree.center.x + half_width and object_y > self.tree.center.y + half_width:
            self.extend_map(QuadPosition.LOWER_RIGHT)
        elif object_x < self.tree.center.x + half_width and object_y < self.tree.center.y + half_width:
            self.extend_map(QuadPosition.UPPER_RIGHT)

        self.tree.insert((object_x, object_y), data=Observation.OCCUPIED)
        self._add_free_points(object_x, object_y, robot_position)

    print("Done updating map")
```

(l) Methode update_map in Klasse Map

Man kann in *update_map* erkennen, dass dort auch eine Matrix-Map gepflegt wird. Das liegt daran, dass der Sprung zum QuadTree etwas später in der Projektlaufzeit durchgeführt wurde, und zur Sicherheit als eine Erweiterung implementiert wurde, statt einer Ersetzung.

Die Map und den QuadTree kann man mit den Methoden *save_map(self)* und *save_tree(self)* speichern. Die Visualisierung des Baums wird mithilfe der visualize Methode realisiert (siehe Abb. m).



(m) QuadTree Visualisierung aus MapTester

Die Ausgabe in Abbildung m wird durch die visualize Methode erzeugt. Das ist ein Resultat aus einer Datei namens *map_tester*, in der man den QuadTree und auch die Matrix selbst ausgeben kann, ohne den Roboter nutzen zu müssen. Es muss nur eine Map erstellt werden, und mit Sensor-Daten aktualisiert die Klasse Map die Matrix und den Baum.

SLAM-Main-Loop

Sensor- und Roboter-Interface, Map und Partikelfilter wurden nun implementiert. Das Einzige, was fehlt, ist eine Main-Loop, welche den Roboter ansteuert. Diese Main-Loop ist in der Datei *main.py* (siehe Abb. n). Darin befindet sich eine Klasse namens **SLAM**. Bei der Erstellung einer Instanz werden Instanzen zur Roboter-Klasse, Sensor-Klasse, Partikelfilter-Klasse und Map-Klasse erstellt. Daraufhin wird die Karte mit Initial-Sensorwerten aktualisiert.

```
def mainloop(self):

    """ Phase 1: Get Sensor Data"""
    sensor_data = self.sensor.get_distances()

    if self.timeStamp is None:
        self.timeStamp = time.time()

    """ Phase 2: Move to closest wall"""
    self.execute_phase_two()


    """ Phase 3: Rotate Robot left"""
    self.robot.rotate_robot(-90)


    """Phase 4: Move Robot around the room"""
    self.execute_phase_four()


    """Phase 5: Save Map and Tree"""
    self.map.save_map()
    self.map.save_tree("tree.pickle")
```

(n) Methode mainloop in Klasse SLAM

Der Main Loop besteht aus insgesamt fünf Phasen. In der ersten Phase wird die Umgebung gescannt, in der man die Methode *get_distances()* im Sensor aufruft.

Dann startet die zweite Phase, in welcher der Roboter sich zur nächsten Wand dreht, dann zu dieser hinfährt. Mit dem Schätzen des Standortes und den Informationen vom Sensor wird regelmässig überprüft, wie weit sich der Roboter von der Wand entfernt befindet. Ist er genug nahe an der Wand, ist Phase 2 abgeschlossen.

```
def execute_phase_two(self):
    self.robot.rotate_robot(self.sensor.angle)
    sensor_data = self.sensor.get_distances()

    while self.sensor.ir.proximity > 15:

        # Get Sensor Data
        sensor_data = self.sensor.get_distances()

        # Estimate Position
        timeDifference = time.time() - self.timeStamp
        pose = self.particle_filter.particle_filter_process(sensor_data,
        timeDifference, self.map, self.sensor.angle)
        self.timeStamp = time.time()

        # Update Map
        self.map.update_map(sensor_data, pose)

        # Move Robot Forward
        self.robot.move_forward()
```

(o) Methode execute_phase_two in Klasse SLAM

In Phase 3 dreht sich der Roboter nach links und ist in Startposition.

In Phase 4 fährt der Roboter den Raum dann linksherum ab. Dazu überprüft der Sensor den Abstand nach vorne und nach rechts. Ist recht keine Wand mehr genug nahe, fährt er rechtsherum, um den gesamten Raum abzufahren. Fährt er vorne zu nahe an eine Wand, dreht er linksherum, um weiter auf Kurs zu bleiben. Phase 4 ist abgeschlossen, wenn ein gewisses Zeitlimit erreicht ist.

```
def execute_phase_four(self):
    while True:

        # Get Sensor Data
        sensor_data = self.sensor.get_distances()

        # Estimate Position
        timeDifference = time.time() - self.timeStamp
        pose = self.particle_filter.particle_filter_process(sensor_data,
        timeDifference, self.map, self.sensor.angle)
        self.timeStamp = time.time()

        # Update Map
        self.map.update_map(sensor_data, pose)
        self.map.save_map()

        # Move Robot
        if self.sensor.ir.proximity < 15: # turn left
            self.robot.rotate_robot(-45)

        elif sensor_data[9] > 15: # turn right
            self.robot.rotate_robot(45)
            self.robot.move_forward()

        else: # move forward
            self.robot.move_forward()

        # Check Time Limit
        if self.robot.time_limit_exceeded(): # check time limit
            print('SLAM completed!')
            break
```

(p) Methode execute_phase_four in Klasse SLAM

Die fünfte und letzte Phase speichert die neu erstellte Karte.

4.3 Herausforderungen bei der Implementierung

Die Implementierung des Projektes lief nicht immer sehr flüssig. Nicht selten sind wir auf Szenarios gestoßen, die uns für eine längere Zeit beschäftigt haben. Manche dieser Probleme konnten wir lösen, aber einige auch nicht.

Hardwarebeschränkungen

Aufgrund dessen, dass wir nur Lego Mindstorms Brick zur Verfügung hatten, hatten wir für das gesamte Projekt wenig Ressourcen zur SLAM-Implementierung. Dies führte dazu, dass wir uns bei der Auswahl der Partikel auf eine niedrige Anzahl einlassen mussten da wir sonst mehrere Minuten pro Schleifendurchlauf warten mussten bis etwas passierte. 100 Partikel haben eine Minute benötigt und bei 50 Partikeln dauert es “nur"15 Sekunden. Besonders beim Updaten und Speichern der Karten wird viel Zeit verbraucht. Im Code wird die Karte pro Schleifeniteration einmal gespeichert, damit bei Abstürzen wir trotzdem in der Lage waren, die Karte im Nachhinein anzuschauen.

Fehlerhafte Messungen des Infrarot-Sensors bei Plastikoberflächen und Rotationsproblematik bei Ultraschall Sensor

Um die Software praktisch zu testen, haben wir kleine Umgebungen gebaut und den Roboter darin gestartet. Beim Aufbauen der Umgebung ist uns aufgefallen, dass bei Plastikoberflächen der Infrarot-Sensor falsche Messwerte erzeugt. Somit konnten wir keine Hindernisse mit Plastikobjekten repräsentieren. Diese Fehler sind während der Nutzung des Infrarot-Sensors vorgekommen. Eine Mögliche Lösung und unser initialer Plan war es, einen Ultraschall-Sensor zu nutzen. Hierbei hatten wir jedoch eine andere Besonderheit, und zwar dass der Rotor, auf dem der Sensor befestigt war nicht mehr akkurat gedreht hat. Für dieses Problem ist die Ursache bisher unbekannt. Dies könnte an einem Hardware-Problem legen, aber wir haben uns entschieden am Ende nur mit dem Infrarot-Sensor zu arbeiten, und bei dem Aufbau einer Umgebung Plastikmaterialien zu vermeiden.

Parallele Motoransteuerung

Bei unseren ersten Tests des EV3Interfaces, haben wir mit dem Package pybricks gearbeitet, um mit dem Pybrick zu kommunizieren und mit den Ports die angeschlossenen Motoren und Sensoren zu steuern. Wir sind recht früh auf ein Problem gestoßen, als es um die Fortbewegung ging. Unser Plan war es, die zwei Räder, oder die Motoren, mit Objekten im Code zu repräsentieren und zu steuern. Hierfür gibt es die Klasse Motor aus pybricks. Wir mussten jedoch eine Lösung finden, um die Motoren parallel anzusteuern, denn nacheinander einzuschalten, würde für ungenaue Pfade sorgen, denn während ein Motor bereits aktiviert ist, muss der andere Motor noch aktiviert werden. Dies sorgt für Artefakte bei der Fahrt. Dieses Problem haben wir mit der Klasse MoveTank gelöst, welche durch das Package ev3dev importiert werden kann. Die MoveTank Klasse ermöglicht es uns mehrere Motoren gleichzeitig zu steuern. Dies hat unser Problem der Parallelität gelöst, und das Package pybricks, wurde in der Implementierung des Algorithmus mit ev3dev ersetzt.

Pip in MicroPython

MicroPython ist ein vollständiger Python Compiler, welcher für Microcontroller geeignet ist. In dem MicroPython welches auf dem Lego Brick installiert war, gab es teilweise nicht die Packages, die wir benötigten oder zumindest veraltete Versionen der Packages. Es war kein PIP vorhanden, weshalb wir die Packages nicht mit einem pip install command installieren konnten, und wir wussten, dass wir nicht die Zeitaufwenden können, um jedes Package als zip-Datei in den Brick zu speichern. Eine temporäre Lösung war, dass wir in das Projekt, die Dateien der benötigten Packages in einen Ordner ziehen, und die benötigten Klassen direkt aus diesem Ordner importieren. Später sind wir auf ein Video von dem Youtube Kanal „The Code City“[1] gestoßen, in dem erklärt wird, wie man mit einem Python-Skript pip installieren konnten. Jedoch war der Brick aufgrund eines veralteten WLAN-Dongles nicht in der Lage auf das Internet zuzugreifen. Deshalb haben wir die Pakete welche wir benötigt haben, wie zum Beispiel quads, welches für QuadTrees genutzt wurde, manuell auf den Brick verschoben.

[1] (City, 2023)

Mühsames Testen aufgrund Modellanzahl

Wir haben für dieses Projekt zwei Roboter gestellt bekommen. Wir sind jedoch 5 Mitglieder und zwischen den Mitgliedern ist teilweise eine Distanz von mehreren Stunden Autofahrt. Deswegen waren wir sehr darauf angewiesen, dass einer der Mitglieder mit den Robotern den Code durchtestet. So wurde die Gesamtimplementierungszeit verlängert, weil die Entwickler Code geschrieben haben, und teilweise keine Möglichkeit hatten, den Code zu testen. Um diesem entgegenzuwirken, haben wir eine Roboter Simulierung mit Lidar entwickelt (siehe Kapitel 4.1), um die Grundgedanken daran testen zu können. Aber die Simulation hatte keinen Einfluss von Unsicherheiten und war deterministisch, also mussten wir auch nach der Nutzung der Simulation sehr viel anpassen. Eine sehr realistische Simulation hätte uns hier eventuell viel Arbeit sparen können, aber nichtsdestotrotz waren unsere Leichtgewichtigen Simulationen hilfreich, Code zu testen.

Debugging mit EV3 Brick

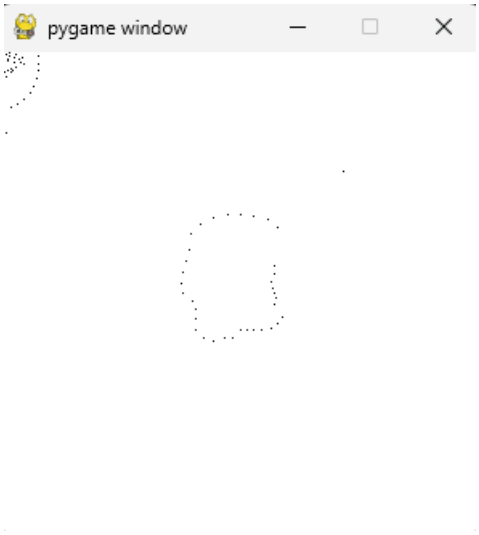
Eine weitere Herausforderung war das Debugging. Die Erstellung der Map Points war schwer während der Laufzeit zu prüfen. Man hätte nach jeder Karten aktualisierung eine Ausgabe erzeugen könne, aber das Problem ist, dass wir keine Möglichkeit hatten diese während der Laufzeit anzusehen. Plots werden nämlich nicht auf dem Desktop angezeigt, sondern technisch gesehen im Lego Brick. Daher dass der Lego Brick keine visuelle Ausgabe für solche Plots anzeigen würde, mussten wir nach jedem Durchlauf die Karte in einer Datei speichern. Diese Datei konnte man dann später ansehen, in dem man die Map-Tester verwendet. Dadurch wurde das Debugging sehr Zeitaufwändig und mühsam.

5 Ergebnisse

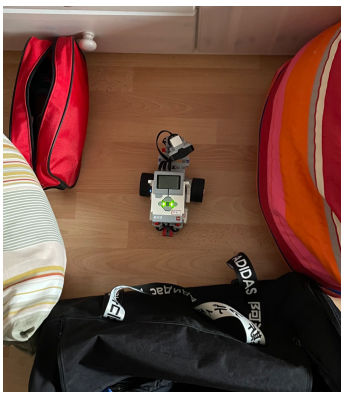
Nach der Implementierung des SLAM-Algorithmus, kommt es nun zum Ergebnis. Daher, dass eine Kontrolle der Karte während der Laufzeit nicht möglich war, haben wir den Roboter durchlaufen lassen. Am Ende wird dann die Karte als .npy und .pickle, also als Matrix und als QuadTree gespeichert.

Um nicht zu lange auf die Testergebnisse des Roboters warten zu müssen, haben wir die Umgebung angepasst. Der Roboter wurde in einem kleineren Bereich platziert, welcher durch Wände gekennzeichnet wird. Innerhalb der Wände wurden jedoch keine Hindernisse platziert, weil der Roboter nicht genügend Freiraum hatte um sich in der Umgebung mit Hindernissen zu navigieren und wir leider manche Objekte nicht als Hindernis nehmen konnten, weil der Infrarot-Sensor diese Hindernisse nicht erkannt hatte, und der Roboter daraufhin rein gefahren ist.

Die Karte welche nach Ablauf gespeichert wurde, wird mithilfe der MapTester Klasse ausgewertet. Nach ungefähr 10 Minuten Laufzeit wird die Map aus Abbildung X erstellt. Die Umgebung, die der Roboter durchlaufen ist, ist in Abbildung b zu sehen.

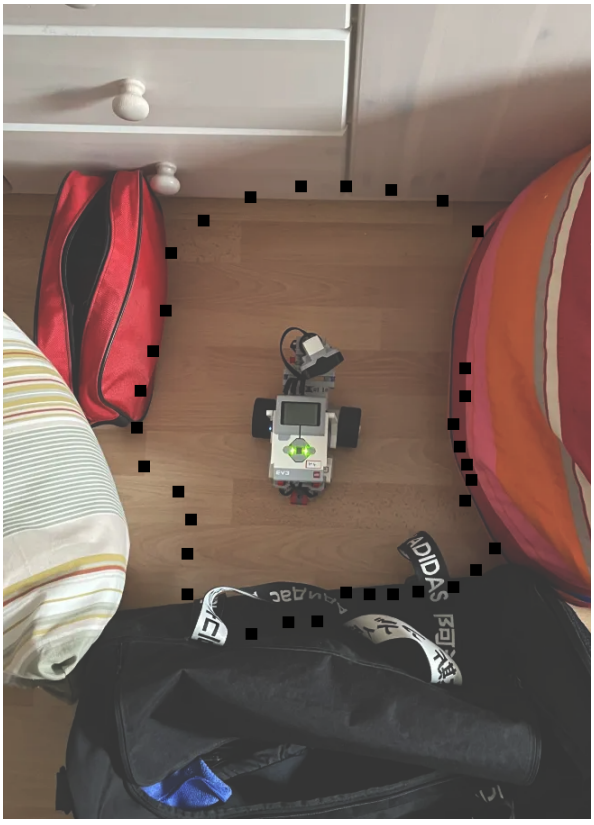


(a) Ergebnis Map



(b) Umgebung

Wenn man nun die Ergebnis Karte (siehe Abb. a) über die Umgebung (siehe Abb.b), dann bekommt dieses Bild.



(c) Ergebnis Map über Umgebung

Es ist deutlich zu erkennen, dass die grundlegenden Hindernisse (also Wände) erkannt wurden und auf der Karte auch so gekennzeichnet wurden. Das Resultat in dem es jetzt gerade ist, war auch so zu erwarten, weil in unserer Implementierung kein Loop-Closure durchgeführt wird, weshalb die Hindernisse etwas verschoben sind. Nichtsdestotrotz ist dies ein positives Ergebnis, mit dem wir im Großen und Ganzen sehr zufrieden sind.

Nach genauer Betrachtung des Ergebnisses, ist uns eine Sammlung an Punkten aufgefallen, welche wir nicht direkt zuordnen konnten. In der Ausgabe sind oben links in der Ecke viele Punkte angesammelt. Wir vermuten, dass diese durch Messfehler resultierten oder durch Rechenfehler in unserem Partikelfilter. Aufgrund der mühsamen Debugging Situation mit dem Roboter, haben wir dieses Problem bisher auch nicht lösen können.

6 Ausblick

Das Ergebnisse welches aus unserer Arbeit resultiert ist, dienst als ein Grundstein für die zukünftige Entwicklung, auf die wir in diesem Kapitel eingehen werden. Wir haben einen leichtgewichtigen, SLAM-fähigen Roboter implementiert, welches Partikelfilter für die Zustandsschätzung nutzt und die Karte in einer Matrix und in einem Quadtree pflegt. Durch diese Eigenschaften, wird eine Basis für weiterführende Forschung geboten.
Im weiteren Verlauf dieses Kapitels werden wir uns mit potenziellen Verbesserungen beschäftigen, die für ein besseres Ergebnis dieses Projektes sorgen würden.

Hardwareressourcen

Die Hardware zu verbessern, könnte zu erheblichen Verbesserungen führen. Ein zum Beispiel besserer Prozessor könnte es dem Roboter ermöglichen, komplexere Berechnungen in Echtzeit durchzuführen. Hierzu könnten bessere Sensoren genauere Messwerte erzeugen, wodurch man die Performance des Roboters verbessern kann. Verbesserte Hardwarekomponenten würden für mehr Möglichkeiten an Algorithmen und Techniken sorgen, weil die Hardware die Implementierungsentscheidungen weniger begrenzen würde.

Sensor Fusion

Sensor Fusion ist das kombinieren der Daten von mehreren Sensoren um die Genauigkeit und Zuverlässigkeit der Informationen zu verbessern. Wenn man Daten von verschiedenen Sensortypen kombinieren würde, könnte man die Unsicherheiten der Roboter Position und Orientierung verringern. Dies würde zu einer genaueren Lokalisierung der Roboters und zu einer genaueren Karte führen.

Pfadplanung

Pfadplanung ist sehr wichtig beim autonomen Fahren, weil dadurch Hindernisse vermieden werden können und die Effizienz des Roboters teilweise vom Pfad abhängig ist. Die Pfadplanung zwischen zwei Punkten wird üblicherweise nach der Kartenerstellung erstellt, aber die Reaktion von Änderungen in der Umgebung wäre hilfreich für das Ausweichen von Hindernissen. Dies wird reaktive Pfadplanung oder reaktive Bewegungsplanung genannt, und könnte in Zukunft für dieses Projekt implementiert werden.

Parallele Bewegung und Scanning

Aktuell ist der Vorgang bei unserer Implementierung, dass der Roboter die Umgebung scannt, sich den Winkel mit der größten Distanz aussucht und dann diesem Winkel entlang fährt. Wenn ein Hindernis im Weg ist und zu nah ist, stoppt der Roboter und wiederholt den Vorgang. Diesen Prozess könnte man so erweitern, dass der Roboter fährt, und gleichzeitig die Umgebung scannt. Die nächsten Aktionen werden dann in Echtzeit berechnet. Hierfür würde man auch mehrere Sensoren benötigen um eine effiziente Berechnung zu ermöglichen.

Generelle Verbesserungen des Quellcodes

Wie in den Ergebnissen verdeutlicht wurde, gibt es noch viel Potenzial bezüglich des Codes. Die ersten Dinge die man machen könnte, wären die Kontrolle des Partikelfilter Algorithmus, um sicherzustellen, dass alles Richtig funktioniert. Die falsch platzierten Punkte aus dem vorherigen Kapitel könnten so eventuell verhindert werden. Außerdem könnte man mithilfe von Loop-Closure die Karte präziser machen, in dem man die verschobenen Wände anpasst, sodass diese mit der tatsächlichen Umgebung übereinstimmen.

Literaturverzeichnis

Ahmed (2023), 'The types of slam algorithms', *Medium* .
URL: *https://medium.com/@nahmed3536/the-types-of-slam-algorithms-356196937e3d*

City, T. C. (2023), 'How to install pip in python 3.10 | pip install in python (easy method)'.
URL: *https://www.youtube.com/watch?v=fJKdIf11GcI*

Deterministic algorithm (2021).
URL: *https://en.wikipedia.org/wiki/Deterministic*$_a lgorithm$

Lim, T., Yeong, C. F., Su, E. L. M., Shithil, S., Chik, S., Duan, F. und Chin, P. (2019), 'Enhanced localization with adaptive normal distribution transform monte carlo localization for map based navigation robot', *ELEKTRIKA- Journal of Electrical Engineering* **18**, 17–24.

Pao, C. (2021), 'Using dead reckoning to solve navigation challenges'.
URL: *https://www.ceva-ip.com/ourblog/using-dead-reckoning-to-solve-navigation-challenges*

Particle Filtering 1.1 Particle Filtering Summary (n.d.).
URL: *https://www.cs.cmu.edu/ 15381-s19/recitations/rec12/Recitation*$_1 2_S ol.pdf$

Seyfarth, G., Batts, Z. und Bagnell, D. (n.d.), 'Good, bad, and ugly of particle filters'.
URL: *https://www.cs.cmu.edu/ 16831-f14/notes/F14/16831*$_l ecture 05_g seyfarth_z batts.pdf$

SLAM - Grid-based FastSLAM Introduction to Mobile Robotics (n.d.).
URL: *http://ais.informatik.uni-freiburg.de/teaching/ss13/robotics/slides/15-slam-gridrbpf.pdf*

Stachniss, C. (2013), 'Slam course - 12 - fastslam (2013/14; cyrill stachniss)'.
URL: *https://youtu.be/Tz3pg3d1TIo*

Stachniss, C. (2020*a*), 'Bayes filter - 5 minutes with cyrill'.
URL: *https://www.youtube.com/watch?v=oUq0a8jHSQg*

Stachniss, C. (2020*b*), 'Kalman filter - 5 minutes with cyrill'.
URL: *https://youtu.be/o_HW6GnLqvg*

Stachniss, C. (2020*c*), 'Particle filter and monte carlo localization (cyrill stachniss)'.
URL: *https://www.youtube.com/watch?v=MsYlueVDLI0*

Stachniss, C. (2022), 'Lidar odometry - 5 minutes with cyrill'.
URL: *https://www.youtube.com/watch?v=9FhKgAEQTOg*

Thrun, S., Burgard, W. und Fox, D. (2005), *Probabilistic Robotics*, MIT Press.
URL: *https://docs.ufpr.br/ danielsantos/ProbabilisticRobotics.pdf*

Turner-Trauring, I. (2022), 'How vectorization speeds up your python code'.
URL: *https://pythonspeed.com/articles/vectorization-python*

Nicht zitierte Quellen