Projektbeschreibung - Laborratte

WiSe 2023/24

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Angewandte Informatik*

Dozent: Prof. Dr. Alexander Huhn

Eingereicht von

Bastian Lehmann (0584512)

Inhaltsverzeichnis

1 Einleitung

Dieses Projekt soll den Lego Mindstorms EV3, auch als Laborratte bekannt, dahingehend umrüsten, sodass er in der Lage ist einen Raum ohne jegliche Vorkenntnisse abzufahren und seine eingeschlagene Route speichert. Die Route soll dann auf einem anderen Gerät visualisiert werden. Der Name ist der Tatsache geschuldet, dass die Maschine im Labor während Ruhezeiten fahren und bei Lehraktivitäten wieder verschwinden soll, ganz gleich einer Ratte die den Menschen meidet. Bei Lärm oder Lehrzeiten soll die Ratte dann zu ihrer Startposition zurückkehren und zu einem anderen Zeitpunkt die Erkundung fortsetzen.
Dieses Dokument beinhaltet sowohl das Resultat dieses Unterfangens, als auch den Weg dorthin und welche Hürden es zu überwinden gab. Dieses Dokument eignet sich außerdem insbesondere dafür, sollten Sie die Laborratte weiterentwickeln wollen. Dafür werden in der Sektion 8 Vorschläge zu möglichen Erweiterungen gemacht. Den Quellcode dieses Projektes finden Sie auf GitHub unter "https://github.com/blackpuschel/Laborratte".
Die Bearbeitung des Projektes erfolgte nicht immer in der selben Reihenfolge der Gliederung dieses Dokumentes. Einige Aspekte wie die Entwicklung des UIs und der Datenübertragung fanden parallel zueinander statt.
Sollten Sie sich für das Aufsetzen der Entwicklungsumgebung interessieren, empfiehlt sich die Vorarbeit von Robin Zschäckel (vgl. Zsc). Hier werden die Grundlagen genauer erläutert.

2 Umbau der Laborratte

Der EV3 Brick inklusive sämtlicher Sensoren und Legoteilen wird in seiner Gesamtheit als Laborratte bezeichnet. Somit ist unter diesem Begriff das gesamte System aus den genannten Komponenten zu verstehen.
Das Ursprüngliche Design der Laborratte von Herrn Zschäckel (vgl. Zsc, 9) wies einige Mängel auf, die dem Projekt im Wege standen. Unter anderem ist der EV3 Brick recht lose an den restlichen Komponenten angeschlossen. Der Sinn davon sollte der einfache Zugang zu den Batterien sein, welcher aber für eine Instabilität sorgte. Des Weiteren ist die Kettenführung entlang der Räder lose. Da die Kette aus einem festen Gummi geschaffen ist und sich nicht an die Räder anschmiegt, greifen die beiden Räder nicht die Kette sonderlich gut. Besonders bemerkbar machte sich das, bei Probefahrten. Die Laborratte erreichte nicht den eingestellten Meter und rotierte nicht um eingestellte Gradzahlen. Hauptgrund für den Umbau war jedoch die Notwendigkeit, den Infrarotsensor seitlich anbringen zu müssen, um Aspekte die näher in Sektion 3 erläutert werden realisieren zu können.
Für den Umbau wurden alle Komponenten anders angeordnet und der Farbsensor entfernt, da dieser für die Laborratte keinen Mehrwert bot. Der Grundsatz bestand trotzdem darin, die Quadratfläche bestmöglich beizubehalten. Diese Form ist essenziell wenn die Laborratte sich nicht bei Rotationen festfahren soll.
Der Antrieb erfolgt nun über zwei vertikal angeordnete Motoren, welche mit der Radaufhängung nach innen gerichtet sind. Dadurch soll ein Heckantrieb anstelle des vorherigen Frontantriebes genutzt werden. Die Entscheidung viel darauf, da unter dem EV3 Brick eine stabile Grundkonstruktion entstehen können sollte und der Platz nicht für die Motoren gereicht hätte. Jedoch resultierte diese Konstruktion darin, dass die Laborratte bei positiven Werten rückwärts fuhr. Somit ist es wichtig bei der Software darauf zu achten, für das vorwärts Fahren und Drehen im Uhrzeigersinn negative Werte zu verwenden.
Die Laborratte besitzt zudem jeweils ein zusätzliches Rad, welches etwas Spannung auf die Kette bringt. Die Spannung darf nicht zu groß sein, da die Legoteile sich biegen könnten. Generell war eine stabile Radaufhängung notwendig. Das Gewicht des EV3 Bricks und die Kettenspannung sorgen für das Biegen von Achsen, welche die Räder halten. Eine leichte Biegung an den Fronträdern besteht noch immer, da die Teile zur Stabilisation nicht aufgebracht werden konnten. Dieser Mangel scheint in der Praxis keine bedeutsamen Auswirkungen zu haben. Wichtig ist auch, dass sich das mittlere und vordere Rad frei drehen lassen, sobald die Kette abgenommen wird. Sollte dies nicht der Fall sein, sollten sie etwas von der inneren Wand weggezogen werden, um die Reibung mit ihr zu verhindern.
Der EV3 Brick ist nun seitlich zur Front befestigt und alle wichtigen Schnittstellen sind erreichbar. Die Abnahme für einen Batterietausch ist dabei sehr simpel. Der Brick ist nur an drei Steckern befestigt, von denen er mit vorsichtigem Ziehen entfernt werden kann. Diese Anordnung kommt mit dem Preis, dass der WLAN-Stick vorneweg die Laborratte überragt. Sollte ein Hindernis einen si-

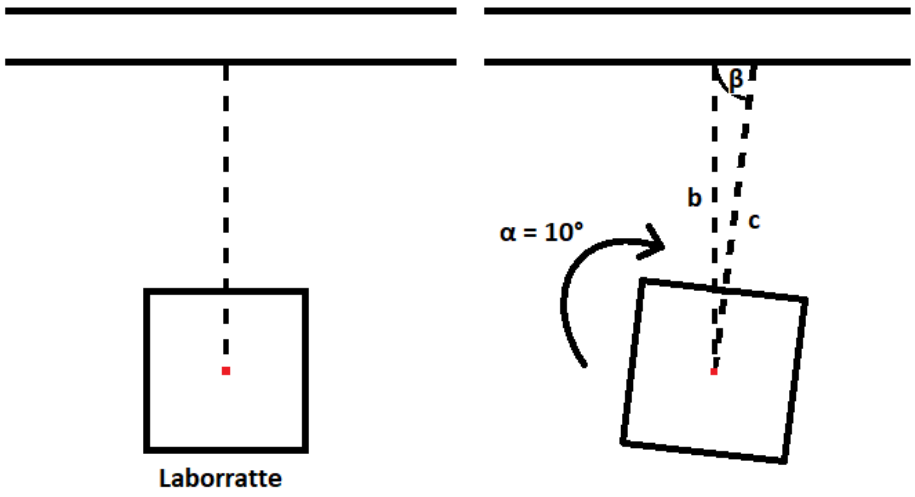
gnifikanten Überhang haben, sodass der Ultraschallsensor ihn nicht wahrnimmt, so könnte es zu einer Kollision mit dem WLAN-Stick und dem Hindernis kommen.
Der Infrarotsensor ist nun an der linken Seite der Laborratte befestigt und kann links-befindliche Wände und Lücken wahrnehmen. Diese Fähigkeit wird für das Abfahren der Wand benötigt.

3 Abfahren des Raumes

Das Abfahren des Raumes wird durch mehrere Algorithmen realisiert, welche sich in der main.py Datei befinden. Wie diese zusammenspielen, um einen Raum abzufahren wird in dieser Sektion erläutert. Zudem befasst sich diese Sektion mit weiteren Ideen, dessen Umsetzung aber nicht realisiert werden konnten.
Grundsätzlich erfolgt das Abfahren wie beim Lösen einen Labyrinthes. Die Laborratte hält sich stets an der linken Wand, bis sie wieder am Ausgangspunkt angelangt ist. Anfänglich fährt die Laborratte in einer geraden Linie, bis sie auf die erste Wand oder Hindernis stößt. Fortan versucht die Laborratte sich stets links von einer Wand zu halten.
Um den Ultraschallsensor mehr Raum für die Hinderniserkennung einzuräumen wurde die ursprüngliche Fahrgeschwindigkeit von 200mm/s auf 50mm/s reduziert.

3.1 Angleichung an Wand durch Berechnung

Wenn die Laborratte auf eine Wand zusteuert, muss sie diese erkennen und sich drehen, bis sie parallel zu ihr ausgerichtet ist. Dafür ermittelt sie mit Hilfe des Ultraschallsensors die Entfernung zum Hindernis, dreht sich um 10° nach rechts und ermittelt erneut die Entfernung. Daraus erhält man zwei Längen b und c, als auch den Winkel zwischen beiden Strecken α. Berechnen möchte man den Winkel β, der die verbleibende Rotation darstellt bis zur Erreichung der Parallelität.



Unter Anwendung des Kosinussatzes lässt sich mit der folgenden Gleichung β berechnen. Sollte die Rotation dafür sorgen, dass die Laborratte an der Wand vorbeischaut (aufgrund eines spitzen Winkels zur Wand) so könnte man um $2 * \alpha$ gegen den Uhrzeigersinn rotieren und die dadurch erhaltenen Werte für die Berechnung nutzen.

Gleichungen:

$$
\begin{aligned}
I \quad & b^2 = a^2 + c^2 - 2accos(\beta) \\
II \quad & a^2 = b^2 + c^2 - 2bccos(\alpha)
\end{aligned}
$$

$$
\begin{aligned}
b^2 &= a^2 + c^2 - 2accos(\beta) && | + 2accos(\beta) \\
b^2 + 2accos(\beta) &= a^2 + c^2 && | - b^2 \\
2accos(\beta) &= a^2 + c^2 - b^2 && | : 2ac \\
cos(\beta) &= \frac{a^2 + c^2 - b^2}{2ac} && | II \ einsetzen \\
cos(\beta) &= \frac{b^2 + c^2 - 2bccos(\alpha) + c^2 - b^2}{2ac} \\
cos(\beta) &= \frac{2c^2 - 2bccos(\alpha)}{2ac} \\
cos(\beta) &= \frac{2c\ (c - bcos(\alpha))}{2ac} \\
cos(\beta) &= \frac{c - bcos(\alpha)}{a} && | \sqrt{II} \ einsetzen \\
cos(\beta) &= \frac{c - bcos(\alpha)}{\sqrt{b^2 + c^2 - 2bccos(\alpha)}} && | arccos \\
\beta &= \arccos \frac{c - bcos(\alpha)}{\sqrt{b^2 + c^2 - 2bccos(\alpha)}}
\end{aligned}
$$

In der Theorie wäre dies eine gute Möglichkeit, die Laborratte parallel zur Wand auszurichten. Die Praxis zeigt jedoch, dass dieser Ansatz nicht umsetzbar ist. Die Dokumentation der *nxtdevices* gibt zum Ultraschallsensor an, dass seine Messungen in mm vorliegen (vgl. LEGa). Dies stimmt auch wenn man sich die Werte ausgeben lässt, jedoch misst der Sensor nur in cm (vgl. ev3) und das auch nicht sehr präzise: "It is able to measure distances from 0 to 2.5 meters with a precision of +/-3 cm." (LEGb, 29). Nach eigenen Erfahrungswerten ist diese Abweichung noch ungenauer.
Hinzu kommt, dass der Ultraschallsensor einige Objekte nur auf kurzer Reichweite oder garnicht erkennen kann. Dies sind gerundete Körper wie Kugeln oder Zylinder, aber auch weiche Oberflächen zählen dazu (vgl. LEGb, 29). Es gibt vielerlei externer Störeinflüsse die den Sensor beeinflussen weswegen er sich für präzise Messungen nicht eignet. Auch bei Probefahrten kam es vor, dass die Laborratte geradewegs auf eine konkave Oberfläche zusteuerte und sie bis zum Ende nicht erkennen konnte und mit ihr kollidierte. Selbst nach physischen Kontakt fuhr sie noch in dieses Hindernis. Somit ist diese Form der Wandangleichung praktisch nicht umsetzbar.

3.2 Angleichung an Wand durch Probieren

Eine weitere Methode um die Laborratte richtig auszurichten ist das Drehen, bis keine Wand mehr in Sichtweite ist. Dies funktioniert in der Praxis nicht, da die Laborratte sich dreht, bis sie die Wand zwar nicht mehr sieht, ihre Ausrichtung aber ein spitzer Winkel zur Wand ist. Es folgt, dass die Laborratte schräg in die Wand fährt und die nahende Wand nicht erkennt.
Dieser Ansatz kann aber erweitert werden mit Hilfe des Infrarotsensors. Zur Erinnerung, dieser ist auf er linken Seite der Laborratte angebracht und wie sich herausstellt verlässlicher als der Ultraschallsensor. Die erhaltenen Messdaten scheinen wesentlich seltener unsinnige Werte zu haben und reagieren schon auf leichte Distanzänderungen. Zu beachten ist, dass die Messwerte zwischen 0 und 100 liegen. Hierbei bedeutet 0, dass ein Objekt direkt vor dem Sensor ist.
Der neue Algorithmus funktioniert wie folgt. Die Laborratte dreht sich bis der Ultraschallsensor keine Wand mehr vor sich hat. Eine Drehung erfolgt immer im Uhrzeigersinn mit jeweils 5°. Anschließend beginnt der Infrarotsensor seine Messungen. Dabei dreht er sich ebenfalls immer um 5° im Uhrzeigersinn, misst aber bei jedem Stopp die Entfernung zur Wand. Dabei wird versucht den kleinsten Wert zu finden. Dieser Wert ist die Stelle, an der der Infrarotsensor am nächsten an der Wand war. Diesen Wert nennen wir d_0. Die Laborratte dreht sich mindestens 3 mal um je 5°. Wird dabei eine neue kleinste Distanz d gefunden so wird d_0 auf d gesetzt und es erfolgen drei weitere Drehungen. Wird keine neue kleinste Distanz gefunden, so dreht sich die Laborratte gegen den Uhrzeigersinn zurück, bis der Infrarotsensor die kleinste Distanz erneut findet.
Die Ausrichtung der Laborratte ist nicht genau parallel zur Wand, reicht aber für einen ersten guten Ansatz aus. Die Implementierung dieses Ablaufes sieht folgendermaßen aus:

```
def align_to_wall():
    while take_measurements(5) < 255:
        robot.turn(-5)

    lowest_measurement = 100
    higher_measurements = 0
    while higher_measurements < 3:
        current_measurement = infraredSensor.distance()
        if current_measurement < lowest_measurement:
            lowest_measurement = current_measurement
            higher_measurements = 0
        else:
            higher_measurements += 1

        robot.turn(-5)

    while infraredSensor.distance() > lowest_measurement + IFRS_TOLERANCE:
        robot.turn(5)
```

Die Toleranz kann benutzt werden, wenn der Infrarotsensor zu starke Abweichungen aufweisen sollte, was in der Praxis nicht er Fall war. Die Funktion *take_measurements()* ist die Messung mit dem Ultraschallsensor. Da dieser manchmal einen Wert von 255 aufweist, obwohl ein Hindernis vor ihm ist kann man einen Parameter angeben, welcher die Anzahl an Messungen angibt. Von diesen Messungen kann dann der Mittelwert gebildet werdem. Sollten dabei nur wenige 255 Messungen im Verhältnis zu den anderen auftreten werden diese bei der Berechnung ignoriert. Der Code für diese Funktion sieht so aus:

```
def take_measurements(amount):
    total_distance = 0
    out_of_ranges = 0
    for _ in range(amount):
        distance = ultrasonsicSensor.distance()
        total_distance += distance
        if distance == 255:
            out_of_ranges += 1

    ratio = out_of_ranges / amount
    if ratio < 0.5:
        total_distance -= out_of_ranges * 255
        amount -= out_of_ranges

    return total_distance / amount
```

3.3 Orientieren an der Wand

Wie bereits im vorherigen Abschnitt erwähnt ist die Laborratte nicht genau parallel zur Wand ausgerichtet. Unabhängig davon kann nicht garantiert werden, dass sie exakt geradeaus fährt. Um trotzdem entlang der Wand fahren zu können wird erneut der Infrarotsensor benutzt.
Es kann eine anfängliche Messung gemacht werden, die den Abstand zur Wand festhält. Sollte sich die Laborratte der Wand nähern oder sich von ihr entfernen, so kann dementsprechend dagegen gelenkt werden. Die Funktion *drive()* der DriveBase Klasse ermöglicht das andauernde fahren, bis dieser Befehl gestoppt wird. Es ist auch möglich den momentanen Kurs zu ändern, indem *drive()* erneut aufgerufen wird (vgl. LEGc).
Um nun den Kurs anzupassen wird *drive()* entweder mit einem Grad nach links oder rechts aufgerufen, solange bis die initiale Distanz wieder erreicht ist. Dieses Vorgehen hält die Laborratte zuverlässig an der Wand. Der Programmabschnitt sieht folgendermaßen aus:

```
def adjust_to_wall():
    global initial_distance

    ...

    delta = infraredSensor.distance() - initial_distance

    ...

    sways_right = delta > 0
    sways_left = delta < 0
    if sways_right:
        robot.drive(SPEED, 1)
    elif sways_left:
        robot.drive(SPEED, -1)
    else:
        robot.drive(SPEED, 0)

    return True
```

Dieser Codeabschnitt ist noch nicht vollständig, angedeutet durch die drei Punkte am Anfang der Funktion. Das liegt daran, dass die Funktion noch eine weitere Funktionalität übernimmt. Hier könnte man zukünftig mit seperation of concerns einiges optimieren. Wofür genau der restliche Teil zuständig ist, wird in dem nächsten Abschnitt erläutert.

3.4 Wegfallen der Wand

Die Laborratte soll sich immer an der linken Wand bewegen, doch zurzeit dreht sie sich nur, wenn eine Wand vor ihr erscheint. Sie muss aber auch auf das Wegfallen einer linken Wand reagieren, also ein Abzweig nach links. Erneut kann der Infrarotsensor für diese Aufgabe genutzt werden. Dieses Problem erscheint recht simpel zu sein, erwies sich aber doch als ein Problem.
Grundsätzlich muss der Infrarotsensor nur einem überaus großem *delta* schauen, jedoch reicht dies nicht vollständig aus. Es ist durchaus üblich, dass an der Wand platzierte Gegenstände wie eine Mauer für die Laborratte wirken. Doch zwischen den Gegenständen können sich kleine Lücken auftun. Damit die Laborratte nicht glaubt, eine Lücke wäre ein Wegfallen einer Wand, wird erst gezählt wie häufig solch eine Messung auftritt. Mit jeder Messung, die die Lücke bestätigt, steigt die Sicherheit, dass es sich um ein Wegfallen der Wand handelt. Dieser Wert ist zurzeit nach Gefühl gesetzt. Konkrete Messungen wären hier jedoch angebracht.
Der Code aus dem vorherigen Abschnitt ist nun um diese Funktionalität erweitert abgebildet:

```
def adjust_to_wall():
    global initial_distance
    global turn_assurance

    ...

    delta = infraredSensor.distance() - initial_distance

    wall_is_gone = delta > 10

    if wall_is_gone:
        turn_assurance += 1
    else:
        turn_assurance = 0

    if turn_assurance >= 10:
        turn_assurance = 0
        initial_distance = -1
        return False

    sways_right = delta > 0
    sways_left = delta < 0
    if sways_right:
        robot.drive(SPEED, 1)
    elif sways_left:
        robot.drive(SPEED, -1)
    else:
        robot.drive(SPEED, 0)
```

```
    return True
```

Wie man sieht muss eine *turn_assurance* von über 10 erreicht werden, bevor eine Drehung ausgelöst wird. Das reicht aber noch nicht aus, um immer an der Wand entlang fahren zu können. Es kann passieren, dass ein Hindernis zwar auf der Höhe des Ultraschallsensors, jedoch nicht auf der Höhe des Infrarotsensors liegt. Dies ist ein Designfehler der Laborratte, dessen Behebung folgendes redundant werden lassen würde.
Wenn ein Objekt zu tief liegt, um vom Infrarotsensor wahrgenommen zu werden, geschehen unerwartete Dinge. Der Infrarotsensor berechnet die initiale Distanz und legt sie auf einen Wert fest, der nicht die linke Wand repräsentiert. Das fahren zu einer tatsächlichen Wand sorgt dann dafür, dass die Funktion *adjust_to_wall()* diesen Unterschied auszugleichen versucht, wodurch die Laborratte effektiv von der Wand wegfährt. Um dieses Problem zu lösen müsste man entscheiden welchen Sensor man glauben schenken mag. In diesem Fall wird ein Kompromiss ausgehandelt. Wenn die initiale Distanz zur Wand zu groß ausfällt, versuche eine neue initiale Distanz zu bestimmen. Solange keine initiale Distanz festgelegt ist fährt die Laborratte ohne Orientierung. Das bestimmen eines neuen Wertes wird solange fortgesetzt, bis entweder ein Wert gefunden werden konnte oder eine bestimmte Zeit erreicht wurde.
Konnte keine initiale Distanz bestimmt werden, so beginnt ein Timer zu laufen. Nach Ende des Timers gilt die Vermutung, dass die Laborratte sich zu weit von einer Wand entfernt hat, also nicht mehr an einer linken Wand fahren kann. Daraufhin wird eine Rotation gegen den Uhrzeigersinn um 90° initiiert. Momentan würde der Timer bei nicht-auffinden der Wand erneut starten, was potenziell zu einer Endlosschleife des im Kreis Drehens führt. Hier könnte man auf ein weiteres Starten des Timers verzichten. Jedoch löst dies das Problem des wegfahrens von der linken Wand. Sollte sich der Infrarotsensor irren und neben der Laborratte ist doch eine (tiefe) Wand, so wird der Ultraschallsensor dies bemerken und sich wieder weg drehen. Dies ist natürlich nur der Best Case. Wie bereits erwähnt kann eine Wandangleichung durch Probieren, ohne verwertbare Messwerte des Infrarotsensors, scheitern und zu einer Kollision führen.
Der vollständige Code sieht dementsprechend aus:

```
def adjust_to_wall():
    global initial_distance
    global turn_assurance
    global timer_running
    global start_time

    if initial_distance < 0:
        initial_distance = infraredSensor.distance()
        if initial_distance > 40:
            initial_distance = -1
            if not timer_running:
                timer_running = True
```

```
                start_time = default_timer()

            duration = default_timer() - start_time
            if duration > 5:
                timer_running = False
                return False

        return True

    delta = infraredSensor.distance() - initial_distance

    wall_is_gone = delta > 10

    if wall_is_gone:
        turn_assurance += 1
    else:
        turn_assurance = 0

    if turn_assurance >= 10:
        turn_assurance = 0
        initial_distance = -1
        return False

    sways_right = delta > 0
    sways_left = delta < 0
    if sways_right:
        robot.drive(SPEED, 1)
    elif sways_left:
        robot.drive(SPEED, -1)
    else:
        robot.drive(SPEED, 0)

    return True
```

Das setzen der *initial_distance* auf eine negative Zahl (hier -1) symbolisiert, dass der Wert noch nicht gesetzt wurde.

3.5 Positionsbestimmung

Die DriveBase Klasse bietet noch einige weitere hilfreiche Funktionen an. die Funktion *distance()* ermöglicht es eine Schätzung der gefahrenen Distanz zu erhalten. Mit *angle()* erhält man eine Schätzung des derzeitigen Winkels relativ zur Startposition und mit *reset()* können diese Werte wieder zurückgesetzt werden (vgl. LEGc). Dadurch lassen sich Koordinaten in einem 2-dimensionalen kartesisches Koordinatensystem darstellen. Die dafür verantwortliche Funktion *get_position()* berechnet die x- und y-Koordinate über die Steigung und über den Pythagoras. Die Steigung erhält man aus der gemessenen Distanz und dem gemessenen Winkel. Da die Koordinaten relativ zur vorherigen Position sind, werden sie auf die letzte Position addiert, um schließlich einen neuen Punkt zu erhalten. Punkte sind hierbei Nodes, ein Tupel aus x- und y-Koordinaten. Sollte dieses Projekt weiter vorangeschritten sein kann man mit den Nodes auch Wegfindung implementieren. Dazu aber mehr in Sektion 8.
Die Funktion ist dadurch recht kompakt:

```
def get_position():
    distance = robot.distance()
    angle = robot.angle()
    robot.reset()

    y = distance * math.sin(angle)
    x = math.sqrt(distance**2 - y**2)

    last_pos = nodes[-1]
    return (last_pos[0] + x, last_pos[1] + y)
```

Es ist anzumerken, dass die Nodes nur Näherungsweise die tatsächliche Position der Laborratte bestimmen können. Viele äußere Faktoren und Hardwarelimitierungen sorgen für ungenaue Ergebnisse. So kann alleinig das kurze Anheben der Laborratte alle zukünftigen Daten invalide werden lassen.

4 User Interface

Ziel ist es gewesen den gegangenen Weg der Laborratte grafisch darzustellen. Dies stellt einige Herausforderungen dar. Zunächst muss geklärt werden, wie Nodes dargestellt werden sollen und wie man die Reihenfolge der angesteuerten Nodes visualisiert. Die anschließenden Sektionen sollen diesen Prozess erläutern und die genutzten Hilfsmittel beleuchten. Das UI befindet sich in der ui.py Datei.

4.1 Allgemeine Darstellung

Die gesamte Visualisierung wurde mit Hilfe des *pygame* Modules realisiert (vgl. Shi). Diese Modul ist ein mächtiges Tool und dient vor allem der Spieleentwicklung im kleineren Stil. Die Wahl fiel auf *pygame* da ich bereits Erfahrung mit diesem Modul besitze und die grafische Darstellung sehr leicht umzusetzen ist.
Grundsätzlich besteht die Oberfläche aus Nodes die von der Laborratte übertragen werden. Diese Nodes sind als grüne Kreise dargestellt. Nodes sind über grüne Linien verbunden. Wenn die Nodes weit genug auseinander sind werden die Linien gestrichelt und bewegt. Die Bewegungsrichtung gibt den Fahrtverlauf an. Dieser wird alleinig aus der Reihenfolge der aufeinander folgenden Nodes gewonnen. Die Reihenfolge der Nodes ist also zu wahren.
Um die Darstellung anzupassen müssen nur die Parameter innerhalb der *draw_nodes()* Funktion angepasst werden. Beispielsweise kann der Funktion *draw_dashed_line()* Argumente für Farbe, Breite des Rings, Margin und weitere mitgegeben werden. Phase stellt dabei eine Verschiebung der gestrichelten Linie dar. Dieser Wert kann und wird für die Animation dieser Linie genutzt.

4.2 Transformation von Koordinaten

Auf dem Canvas bzw. Bildschirm den *pygame* bereitstellt kann gezeichnet werden. Dafür wird die Position in Form eines Tupels mit Höhe und Breite benötigt. Man kann sich das also wie ein Koordinatensystem denken, nur dass ausschließlich der 1. Quadrant vorhanden ist und an der x-Achse gespiegelt wurde. Die Position (0, 0) befindet sich oben links.
Eingehende Daten der Laborratte bestehen sowohl aus negativen Zahlen, als auch aus Werten die zu klein oder zu groß für den derzeitigen Bildschirm sind. Somit müssen diese Werte vor ihrer Visualisierung transformiert und skaliert werden. Der gesamte Algorithmus soll hier nicht im Detail erläutert werden, aber seine grundsätzliche Funktionsweise.
Zunächst müssen alle Nodes in den 1. Quadranten verschoben werden. Dafür ist *shift_nodes* zuständig. Wenn das problemlos in die momentane Verschiebung eingearbeitet werden kann, kann *shift* aus vorherigen Transformationen genutzt werden. Ist dies nicht möglich, weil der neue Node wesentlich weiter unten oder links liegt als alle bisherigen Nodes, so müssen auch die alten Nodes neu *geshifted* werden.
Anschließend müssen die Nodes so skaliert werden, dass sie alle auf den Bildschirm passen. Zuständig ist die Funktion *scale_nodes*. Reicht die bisherige Ska-

lierung aus um die neuen Nodes abbilden zu können, so kann diese angewendet werden. Wären die neuen Nodes jedoch außerhalb des sichtbaren Bereiches, so muss eine neue Skalierung berechnet und auf alle Nodes angewendet werden. Immer wenn alle Nodes betroffen sind, müssen erst vorherige Modifikation rückgängig und anschließend wieder angewandt werden. Ich sehe daher in den bestehenden Funktionen viel Verbesserungspotenzial im Bezug auf deren Laufzeiten.
Wenn Sie den dazugehörigen Code betrachten fällt Ihnen unter Umständen diese Zeile Code auf:

```
scaled_y = abs(scaled_y - SCREEN_HEIGHT)
```

Sie spiegelt die Nodes an der x-Achse, sodass Nodes wie (0,0) nicht oben links, sondern unten links angezeigt werden.

5 Datenübertragung

Das UI kann Nodes akkurat darstellen und die Laborratte ermittelt eine Approximation ihres aktuellen Standortes. Nun müssen diese Daten übermittelt werden.
Die Laborratte besitzt einen WLAN-Stick, mit dessen Hilfe sie sich in einem Netzwerk registrieren kann. Die Anleitung dafür finden Sie bei Bedarf in der Vorarbeit von Herrn Zschäckel (vgl. Zsc, 13).
Intern wird für die Kommunikation das "socket" Modul von Python genutzt. Dabei stellt das UI den Server dar und die Laborratte den Client. Die Wahl für diese Konstellation viel darauf zurück, dass man einen Server, am Besten mit einem festen Domainnamen, irgendwo laufen lassen kann und die Laborratte sich automatisch mit diesem verbindet. Dafür besitzt die Laborratte neben der main.py auch eine JSON-Datei mit ihr bekannten Servern. Vor dem Start der Laborratte kann diese Datei leicht bearbeitet werden. Die Laborratte liest dann zur Laufzeit diese Datei ein und versucht sich mit allen angegebenen Servern zu verbinden, bis sie eine Verbindung herstellen kann.
Die gesamte Kommunikation zwischen UI und Laborratte erfolgt in einem anderen Thread, um den Main-Thread nicht zu unterbrechen. Dabei ist der UI-Thread als Daemon-Thread angelegt, was bei der Laborratte nicht möglich war. Vermutlich liegt dies an einer veralteten micropython Version oder fehlender Unterstützung dieses Befehls. Trotz alledem beeinflusst das Fehlen dieser Funktion die Threads nicht negativ.
Nachdem eine Verbindung zwischen UI und Client besteht, beginnt der Datenaustausch. Die Laborratte behält dafür einen Zähler der festhält, welche Nodes bereits übertragen wurden und welche eine Übertragung erwarten. Dadurch müssen nicht immer alle Nodes übertragen werden, sondern stets die neuesten. Die zu übertragenden Nodes werden mit der *struct.pack()* Funktion des *struct* Modules in ein bytearray geschrieben und anschließend über die Socket gesendet. Hier ein Ausschnitt dieser Funktion:

```
def get_data_bytes():
    global sent_nodes

    byte_values = bytearray()
    unsent_nodes = nodes[sent_nodes:]

    sent_nodes = len(nodes)

    for node in unsent_nodes:
        byte_values.extend(struct.pack('dd', node[0], node[1]))
    return byte_values
```

Das erste Argument der *struct.pack()* Funktion weist darauf hin, dass die folgenden Argumente von 64-Bit floats zu Bytes konvertiert werden sollen. Ein "'d'" steht für einen 64-Bit float, "'dd'" für zwei 64-Bit floats. Würde man stattdessen die Nodes lieber als 32-Bit floats übertragen wollen, müsse man ein "'d'" durch

“’f’“ ersetzen.
Auf der Seite des UIs wird dieser Prozess mit der *struct.unpack()* Funktion rückgängig gemacht. Auch hier muss der Datentypen-String mit “’dd’“ übereinstimmen:

```
def byte_to_nodes(bytes: bytes):
    result = []
    for i in range(0, len(bytes), 16):
        node = struct.unpack('dd', bytes[i:i+16])
        result.append(node)
    return result
```

Beachte jedoch, dass die for-Schleife bei anderen Datentypen auch um einen anderen Wert inkrementiert werden könnte. Da hier zwei 64-Bit floats ausgelesen werden sollen, also insgesamt 16 Byte, muss die Zählvariable auch immer um 16 erhöht werden. Beim Auslesen zweier 32-Bit floats sollten Sie also die Variable nur um 8 bei jedem Durchlauf erhöhen.
Die erhaltenen Nodes werden wie im vorherigen Kapitel transformiert und an die Liste der bestehenden Nodes gehängt. Die korrekte Reihenfolge ist dabei einzuhalten.

6 Resultate

Die Laborratte hatte anfänglich viele Ziele zu erfüllen, von denen einige aus zeit-technischen Gründen gestrichen werden mussten. Solcherlei Features sind unter 8 aufgeführt. Zu den dennoch adressierten Problemstellungen ist das Resultat eines, das unvollständig aber weiter entwickelbar ist. Die Laborratte kann bei guten Raumbedingungen diesen zuverlässig abfahren. In der Realität gelten diese Bedingungen nicht und die Laborratte fährt sich an vielen Hindernissen fest.
Es ist der Laborratte auch noch nicht möglich zu erkennen, wann sie erfolgreich einen Raum abgefahren hat. Momentan fährt sie unaufhörlich. Allerdings sind die grundlegendsten Mechanismen vorhanden. Sie kann auf Wände und Hindernisse mit collision avoidance reagieren und sich derartig drehen, dass sie annähernd parallel zu diesen weiterfahren kann. Auch kann sie erkennen, wenn sie nach links abzubiegen hat, um den Raum vollständig zu erfassen. Dazu werden die Stellen des Abbiegens als Nodes intern gespeichert. Somit ist die Laborratte in der Lage ihre Position zu bestimmen.
Auch ein UI zur grafischen Darstellung des genommenen Weges ist vorhanden und auch funktionsfähig. Es stellt die Nodes zuverlässig und leicht verständlich dar.
Auch die Kommunikation zwischen UI und Laborratte erfolgt problemlos. Ebenso ist ein Austausch von Servern sehr simpel zu handhaben.
Die Laborratte ist im jetzigen Zustand nicht in der Lage ihren Weg zum Startpunkt zu finden. Auch reagiert sie nicht auf ihr Umfeld. Damit gemeint sind Lautstärke und Lehrzeiten.

7 Limitationen

Der Umbau der Laborratte ist nur dann möglich, wenn man Zugriff auf benötigte Klemmbausteine hat. Andernfalls muss mit Schwächen in der Konstruktion gerechnet werden.
Die verwendeten Klemmbausteine von Lego bestehen aus einem festen, dennoch sehr biegsamen Plastik, welches unter dem Gewicht des EV3 Bricks und der Kettenspannung nachgibt. Es kommt zu kleineren Biegungen von Achsen. Außerdem ist das Gerüst, an dem der EV3 Brick befestigt ist nicht sehr stabil, da die Klemmkraft dieser Steine nachzugeben scheint.
Auch ist die Hardware in vielen belangen Fehleranfällig und unpräzise. Der Ultraschallsensor übersieht eine Vielzahl von Hindernissen. Dazu gehören insbesondere konkave, stoffartige und raue Objekte (vgl. Wes17). Dies hat etwas mit den physikalischen Eigenschaften der Wellenausbreitung und deren Absorption zu tun.
Das Fahrverhalten ist nicht genau. Es können keine exakten Drehungen und Distanzen gefahren werden. Grund hierfür ist zum einen die Hardware und zum anderen die Werte die die *DriveBase* Klasse benötigt. Hierfür können nur ungefähre Werte genutzt werden, welche durch ausprobieren ermittelt wurden.
Der Ultraschallsensor gibt gelegentlich Werte von 255 an, signalisierend, dass kein Hindernis in Reichweite ist, obwohl Messungen davor und danach kleiner als dieser Wert ausfallen. Dieses Verhalten scheint zufällig aufzutreten und kann Messergebnisse verfälschen.

8 Erweiterbarkeit

Die Laborratte ist zurzeit unvollständig und kann dementsprechend vielseitig erweitert werden.
Es ist möglich das Raumabfahren effizienter und präziser zu gestalten. Dazu gehört zum einen die bestehenden Probleme mit Lücken oder nur schwer erkennbaren Hindernissen zu Lösen. Man könnte weitere Ultraschallsensoren nutzen oder den Infrarotsensor anders positionieren. Dabei sei aber zu beachten, dass sich mehrere Ultraschallsensoren mit ihren Signalen behindern könnten.
Ein weiteres großes Problem stellen Stühle dar, welche zwar von dem Ultraschallsensor erkannt werden, jedoch nicht von dem höher hängendem Infrarotsensor. Dieses Problem könnte mit einem Umbau beseitigt werden.
Ebenso kann man dafür sorgen, dass nach Abfahren der Raumgrenzen, die Laborratte den gesamten Raum erkundet. Dabei könnte die Ratte auf den Lärm und bestehende Lehrzeiten des Raumes achten und darauf reagieren. Die Reaktion könnte entweder mit dem Zurückfahren zur Startposition erfolgen oder mit dem gezielten Verstecken in einer dunklen Ecke. Möglicherweise eignet sich dafür der Farbsensor.
Der Farbsensor könnte ebenfalls dafür genutzt werden, eine bestimmte Folge von Farben zu finden, welche das Ziel der Reise darstellt. Damit könnte man in Kombination mit dem Abfahren des Raumes das Verstecken und Suchen eines Gegenstandes realisieren. Die Ratte könnte versuchen während Ruhezeiten einen farbigen Zettel im Raum zu finden.
Da die Laborratte mit einem UI kommunizieren kann, welches Punkte visualisieren kann, wäre es theoretisch möglich auf dem UI Punkte anzugeben, die die Ratte ansteuern soll. Man könnte dafür auch über bereits bestehende Punkte fahren und Algorithmen wie Dijkstra oder A* anwenden. Natürlich sind auch viele andere Algorithmen für diese Wegfindung denkbar.
Schließlich wäre eine weitere Erweiterungsmöglichkeit, mit Hilfe von anderen Gerätschaften und der Trilateration genauere Positionsdaten für die Laborratte zu bestimmen. Die damit erhaltenen Daten würden der Laborratte eine wesentlich präzisere Fahrt ermöglichen. Insbesondere würden Kollision und das Aufheben der Ratte nicht zu inkonsistenten Positionsdaten führen.

9 Quellen

Literatur

[ev3] EV3DEV.ORG (Hrsg.): *Sensor Data*. ev3dev.org. – http://docs.ev3dev.org/projects/lego-linux-drivers/en/ev3dev-stretch/sensor_data.html#lego-nxt-us [Aufgerufen am 05.01.2024].

[LEGa] THE LEGO GROUP (Hrsg.): *NXT Devices*. The LEGO Group. – https://pybricks.com/ev3-micropython/nxtdevices.html [Aufgerufen am 05.01.2024].

[LEGb] THE LEGO GROUP (Hrsg.): *NXT User Guide*. The LEGO Group. – https://www.generationrobots.com/media/Lego-Mindstorms-NXT-Education-Kit.pdf [Aufgerufen am 05.01.2024].

[LEGc] THE LEGO GROUP (Hrsg.): *Robotics*. The LEGO Group. – https://pybricks.com/ev3-micropython/robotics.html [Aufgerufen am 06.01.2024].

[Shi] SHINNERS, Pete: *Pygame Front Page*. – https://www.pygame.org/docs/ [Aufgerufen am 06.01.2024].

[Wes17] WESTHOFF, Mike: *Objekterkennung mit Ultraschall mit Matlab/Simulink und EV3*. 2017. – https://wiki.hshl.de/wiki/index.php/Objekterkennung_mit_Ultraschall_mit_Matlab/Simulink_und_EV3 [Aufgerufen am 07.01.2024].

[Zsc] ZSCHÄCKEL, Robin: *LEGO Mindstorms EV3 mit obstacle avoidance fahren lassen und Daten der Sensoren an einen Computer übertragen und darstellen*