



SÖDERBLOM GYMNASIUM

PROJEKTDOKUMENTATION

Intelligenter Labyrinth-Roboter

Jan Beckschewe und Jan Reimer

Fachlehrer: Herr Salloch

Verfasser: Jan Reimer

Schuljahr 2016/17

6. Juli 2017

Inhaltsverzeichnis

1 Einleitung	3
2 Theorie	4
2.1 Hardware	4
2.2 Verwendete Libraries	7
3 Praxis	7
3.1 Anleitung zur Benutzung	7
3.2 PD-Regler	8
3.3 Abbiegen	9
3.4 Lösen des Labyrinths	11
3.5 Selbstkalibration	14
3.6 Android App	14
4 Zusammenfassung	15
5 Erklärung der Urheberschaft	15

1 Einleitung

Allgemein Robotik ist ein Feld mit vielen Anwendungsmöglichkeiten. Von Militärrobotern bis zu Robotern die Menschen aus Trümmern helfen, aber auch bei gewöhnlichen Dingen wie im Haushalt spielt die Robotik eine immer größer werdende Rolle. Viele Menschen sind an Robotik interessiert, jedoch gibt es Probleme die im Weg stehen: ungeeignete Umgebung und ungenaue Sensoren erschweren den Einsatz erheblich. Das Labyrinth besteht aus dünnem schwarzen Klebeband das auf einer weißen Tapete angebracht ist. Um das problemlose abfahren des Labyrinths zu ermöglichen müssen mehrere Faktoren beachtet werden: Bewegungskontrolle in Kombination mit Sensorik, Lösungsalgorithmen für das Labyrinth sowie das Design des Fahrzeugs.

Zielsetzung Unser Ziel war es einen intelligenten, kleinen und autonomen Roboter zu bauen, der ein beliebiges Labyrinth lösen kann, indem er den kürzesten/schnellsten Weg zum Ziel findet. Diese Dokumentation wird die notwendige Hardware erklären, angefangen mit dem Mikrocontroller bis zur Auswahl des Akkus.

Was ist ein Mikrocontroller? Ein Mikrocontroller ist ein kleiner Computer in einem einzigen integrierten Schaltkreis (englisch integrated circuit, kurz IC). In der modernen Fachsprache werden Mikrocontroller als 'system on a chip', Ein-Chip-System oder auch kurz als SoC bezeichnet. Ein Mikrocontroller beinhaltet einen oder mehrere CPUs (Prozessoren) sowie Speicher und programmierbare I/O Schnittstellen. Außerdem ist oft Programmspeicher, RAM und komplexe Peripherie wie USB- (Universal Serial Bus), I²C- (Inter-Integrated Circuit) Schnittstellen verbaut. Microcontroller sind für den Einsatz in 'embedded applications' (deutsch: eingebettete Systeme) entworfen, im Gegensatz zu Mikroprozessoren die in PCs verwendet werden.

Mikrocontroller werden in automatisch kontrollierten Geräten wie in Motor-Kontrollsystemen, medizinische Implantaten, Fernbedienungen, Elektrowerkzeug, Spielzeug und in vielen anderen embedded systems benutzt. Die ständige Reduktion der Größe und Kosten von Mikrocontrollern macht es technisch und wirtschaftlich möglich immer mehr Geräte und Prozesse digital zu steuern. Mixed signal Microcontroller sind weitverbreitet, um analoge Komponenten zu integrieren, das ist notwendig um nicht digitale elektronische Systeme zu steuern.

Manche Mikrocontroller benutzen lediglich 4-bit Binärwörter und laufen auf sehr geringen Frequenzen, um den Stromverbrauch zu senken (einstellige Milliwatt oder Microwatt). Der Stromverbrauch im 'sleep' Zustand (CPU clock und der Großteil der Peripherie ist abgeschaltet) liegt bei wenigen Nanowatt das macht sie perfekt für Anwendungen im Batteriebetrieb.

2 Theorie

2.1 Hardware

Arduino Uno Rev.3 Der Arduino Uno ist ein Mikrocontroller Board das auf dem 8-bit ATmega328P basiert, er hat 14 digitale I/O pins, 6 analoge Eingänge, eine USB Schnittstelle, einen Stromanschluss, einen ICSP header und einen reset Knopf. Sobald man eine Stromquelle über USB, mit AC zu DC Adapter an die Steckdose oder einfach eine Batterie anschließt, kann es los gehen. Man kann sich mit dem Uno austoben ohne viel Angst zu haben etwas falsch zu machen, im schlimmsten Fall tauscht man den Chip für wenige Euro aus.

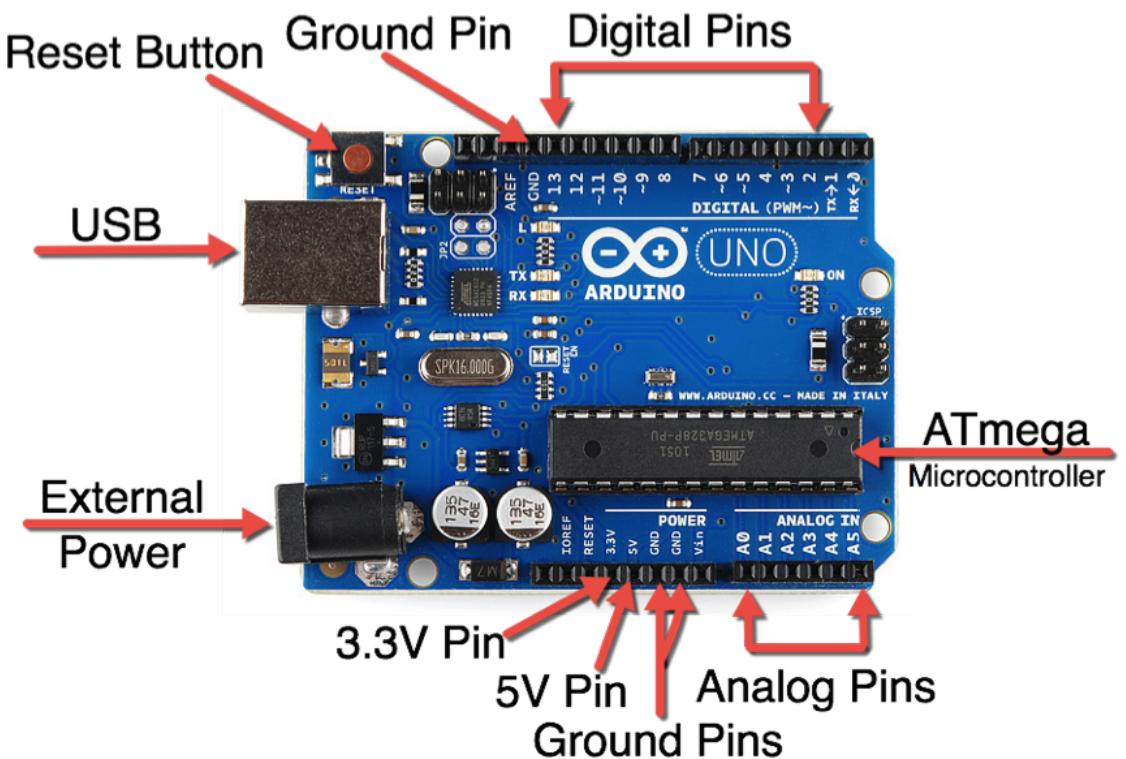


Abbildung 1: Arduino Uno [5]

Tabelle 1: Spezifikationen des Arduino Uno

Mikrocontroller	ATmega328p
Betriebsspannung	5V
Eingangsspannung	6-20V
Digitale I/O Pins	14
Analoge Eingangs Pins	6
DC Strom pro I/O Pin	20 mA
DC Strom pro 3.3V Pin	50 mA
Flash Memory	32 KB
Clock Speed	16 MHz

Arduino Motor Shield Rev.3 Der Arduino Motor Shield basiert auf dem L298 Dual-Vollbrücken-Motortreiber und ist entworfen, um mit induktiven Ladungen umzugehen, die beispielsweise von unseren Motoren erzeugt werden. Der Motor Shield ermöglicht es die Geschwindigkeit und Richtung von beiden DC Motoren eigenständig zu regulieren.

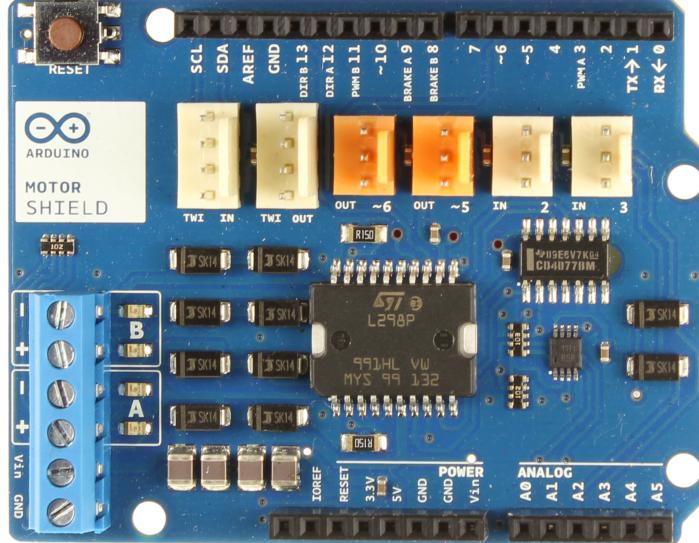


Abbildung 2: Arduino Motor Shield [4]

Mini Roboter Rover Chassis Kit Wir haben uns entschieden einen Bausatz zu benutzen, um uns das anfängliche Hardware Chaos zu ersparen. Das Kit enthält: 2 Räder, 2 DC Zahnrad Motoren, ein Stützrad, ein Metall Chassis und eine Metallplatte mit Befestigungen.

Da der in der Betriebsanleitung empfohlene Aufbau nicht für unsere Zwecke geeignet ist, haben wir uns entschlossen die Metallplatte an der Unterseite des Autos zu befestigen, um die Sensoren möglichst nah an der Drehachse zu verbauen. Zudem haben wir das Stützrad weiter vorne angebracht, um die Stabilität des Fahrzeugs zu verbessern. Die DC Plastik Zahnrad Motoren sind nicht besonders stark und scheitern am kleinsten Hindernis, dennoch sind sie für unsere Einsatzzwecke brauchbar. Das Fahrzeug wird durch eine Powerbank mit Strom versorgt.

QTR-8A Reflexions-Sensoren-Array Der QTR-8A ist zwar als Liniensorer entworfen und wird auch so von uns genutzt, jedoch kann man ihn auch als Näherungssensor oder auch einfach als Reflexionssensor. Auf dem Modul befinden sich acht Infrarotstrahler/Infrarotempfänger (Phototransistoren) Paare in gleichmäßigen Intervallen. Jeder Phototransistor ist mit einem 'pull-up' Widerstand verbunden, um die Spannung zu verteilen. Jeder angeschlossene Sensor gibt eigenständig eine analoge Spannung zurück die

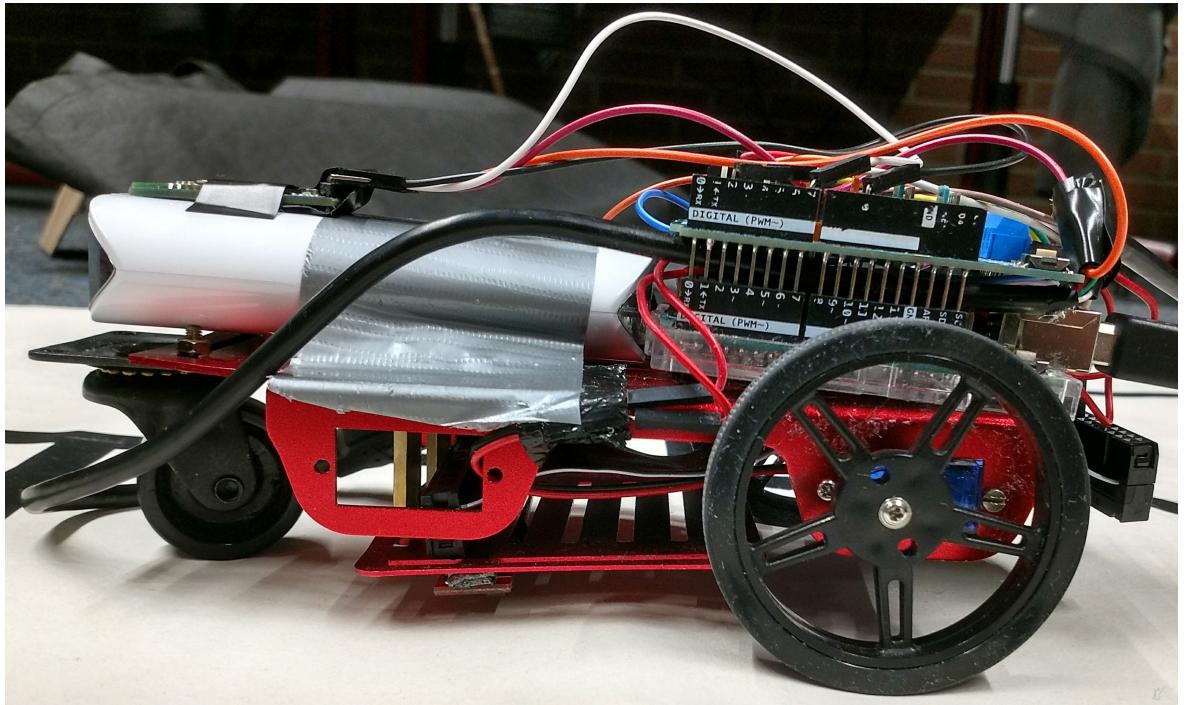


Abbildung 3: Aufbau des Roboters

zwischen 0 Volt und 5 Volt liegt. Geringere Spannung deutet auf eine höhere Reflexion hin.

Der Sensor kann sehr gut zwischen schwarz und weiß unterscheiden, weshalb er sich gut für den Roboter eignet. Im Nachhinein wäre ein Sensor mit I²C Anschluss besser gewesen, da man sich so einige Kabel sparen kann.



Abbildung 4: QTR-8A Sensor[6]

SH-HC-08 Bluetooth 4.0 BLE Modul Der HC-08 ist ein Bluetooth 4.0 Modul. Das Modul benutzt einen Texas Instruments CC2541 Chip der AT Kommandos unterstützt. Man kann die 'baud rate' des Serial ports, den Anzeigenamen des Geräts und das Passwort ändern. Das Bluetooth Modul wird zur Kommunikation zwischen Android Smartphone und Roboter benutzt. Das Modul hat zunächst nicht funktioniert, da der Spannungsabfall bei den Pins des Arduino schon zu groß war, deshalb haben wir kurzerhand das USB-Kabel durchgetrennt und das Modul direkt an die Stromversorgung angeschlossen.

2.2 Verwendete Libraries

Pololu QTR Reflectance Sensors [1] Um die Library benutzen zu können muss zunächst ein QTRSensorsAnalog Objekt initialisiert werden. Das Objekt gibt einem die Möglichkeit auf die Funktionen der Library zuzugreifen, die das Einlesen von Sensordaten ermöglichen. Die Library gibt einem Zugriff auf die rohen Sensordaten sowie auf 'high level' Funktionen wie das kalibrieren und das Einlesen einer Linie.

Listing 1: Initialisierung des Sensors

```
QTRSensorsAnalog qtra(sensorPins, sizeof(sensorPins), 4, 2);
unsigned int sensorValues[sizeof(sensorPins)];
```

SoftwareSerial Library [3] Die Arduino unterstützt native serielle Kommunikation auf den digitalen Pins 0 und 1. Diese native serielle Unterstützung wird durch den eingebauten 'Universal asynchronous receiver/transmitter' ermöglicht. Die SoftwareSerial Library emuliert die serielle Kommunikation an anderen digitalen Pins des Arduino. Wir haben uns das zunutze gemacht um das Bluetooth Modul an die digitalen Pins 7 und 10 anzuschließen.

3 Praxis

3.1 Anleitung zur Benutzung

Zur Benutzung benötigen Sie ein 2D-Labyrinth das aus schwarzen Linien auf weißen Untergrund besteht. Achten Sie darauf:

- Das der Roboter genug Platz zum Wenden hat
- Das Labyrinth keine Schleifen beinhaltet
- Der Roboter das Labyrinth nicht verlassen kann
- Sie das Ziel wie in Abbildung 5 mit einem ausreichend großen schwarzen Kasten markieren

Installieren Sie die App aus der Repository[2] und stellen Sie sicher das der Akku des Roboters geladen ist. Stellen Sie den Roboter auf irgendeine Linie und stecken Sie das Kabel in die Powerbank. Der Roboter kalibriert sich nun selbst. Öffnen Sie die App und warten Sie bis der Roboter sich verbindet. Drücken Sie auf Start und der Roboter fängt an zu fahren, nachdem der Roboter das Ziel erreicht hat stellen Sie ihn wieder auf den Ausgangspunkt. Der Roboter wird dann den schnellsten/kürzesten Weg zum Ziel fahren. Dieser Vorgang kann beliebig oft wiederholt werden, indem man den Roboter nach erreichen des Ziels erneut an den Ausgangspunkt stellt. Will man den Ausgangspunkt

ändern stellt man den Roboter auf einen anderen Teil des Labyrinths und drückt die Reset-Taste. Der Roboter wird nun die obigen Schritte erneut durchlaufen.

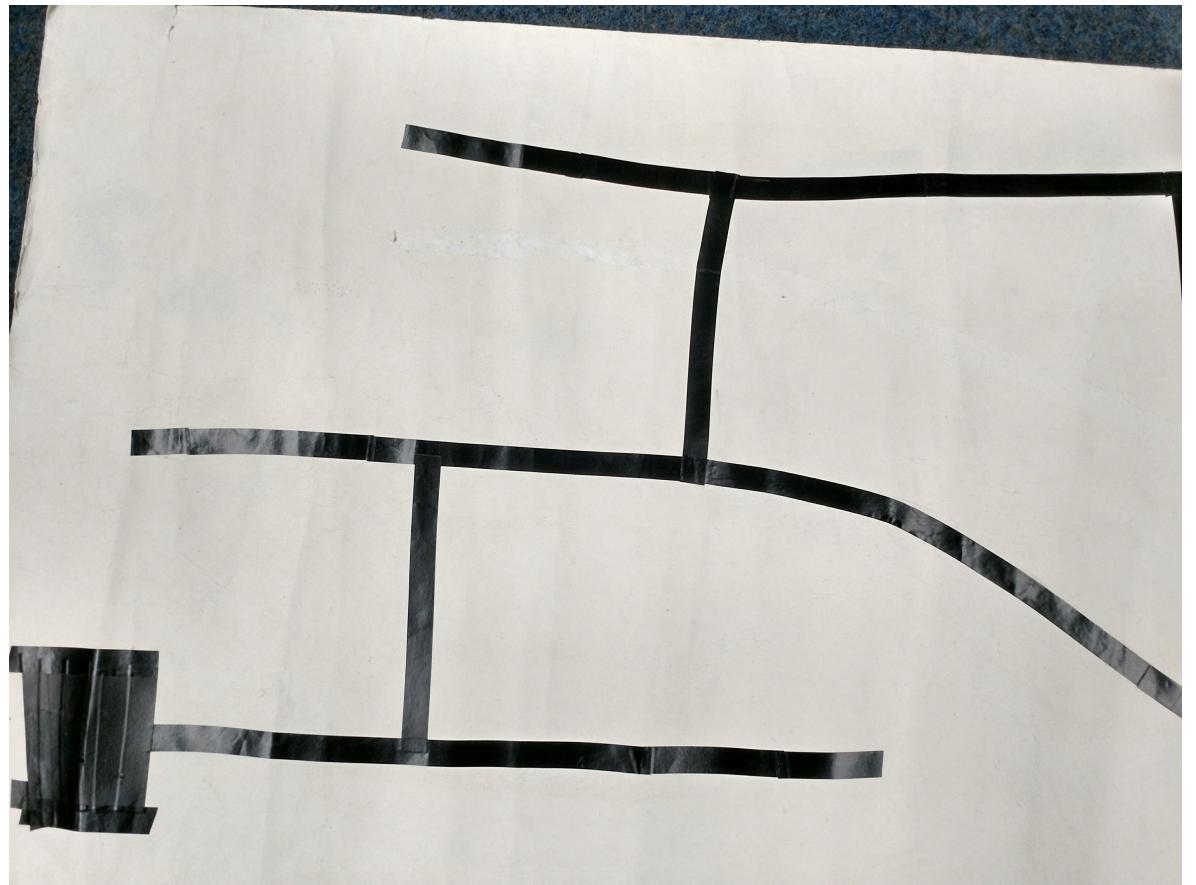


Abbildung 5: Beispiel Labyrinth

3.2 PD-Regler

Anfängliche Probleme Eine der ersten Aufgaben der wir uns stellen mussten war das Abfahren einer Linie, wobei das Zusammenspiel zwischen Sensorik und Motoren besonders wichtig ist. Doch bevor wir uns diesem ersten Problem widmen konnten, hatten wir mehrere Probleme: zwei Sensoren gaben keine Werte zurück, die Sensoren waren zu weit vom Boden entfernt und das Klebeband, was wir als erstes verwendet haben, war ungeeignet. Das Klebeband war schnell ausgewechselt und nach einem Umbau des Roboters war auch die Entfernung der Sensoren kein Problem mehr. Die zwei nicht funktionierenden Sensoren beschäftigten uns jedoch für eine längere Zeit. Nachdem wir nach geraumer Zeit falschen Code und defekte Hardware ausschließen konnten, haben wir festgestellt das es Pins gibt die immer vom Shield verwendet werden, um Spezialfunktionen des Shield zu benutzen, wie Kontrolle der Richtung und Geschwindigkeit. Da die Funktion (Strommessung) der beiden Pins für unser Projekt nicht zu gebrauchen ist, bogen wir die Verbindungsstellen des Shields ab und verbanden die beiden Sensoren direkt mit dem Arduino.

Umsetzung Ungenauigkeiten im Labyrinth und Unebenheiten verhindern es das man den Roboter nur geradeaus fahren lassen kann, da er sonst schnell die Spur verliert und man eventuell auch Kurven fahren möchte. Deshalb muss ein Regler implementiert werden, der selbständig die Geschwindigkeiten der Motoren anpasst, um den Roboter möglichst mittig auf der Strecke zu halten.

Der Roboter kann mithilfe der Infrarotsensoren bestimmen wie er im Labyrinth steht. Die `readLine()` Methode der Library gibt bei 6 Sensoren einen Wert zwischen 0 und 5000 zurück. Mithilfe dieser Methode haben wir einen PD Regler implementiert, der anhand der jetzigen Position (proportional) und der letzten Position (derivative) seine Motorgeschwindigkeiten anpasst. PD Regler kommen zum Beispiel in Klimaanlagen verbaut, damit wird verhindert das die Klimaanlage den Raum nicht immer periodisch unterkühlt und sorgt dafür das die Wunschtemperatur im Raum herrscht.

Listing 2: PD Regler Implementation

```
// PD loop constants
const float proportionalConst = 0.2f;
const float derivateConst = 1.0f;

posPropotionalToMid = sensorPosition - 2500;

motorSpeed = proportionalConst * posPropotionalToMid + derivateConst *
    (posPropotionalToMid - lastError);
lastError = posPropotionalToMid;

moveBothMotors(maxMotorSpeed - motorSpeed, forward, maxMotorSpeed +
    motorSpeed, forward);
```

Wie man im Code sieht werden zwei Konstanten initialisiert die den Einfluss der jetzigen und vorherigen Position auf die Motorlenkung bestimmen. Die Konstanten müssen dem Gewicht des Roboters und der Leistungsfähigkeit der Motoren angepasst werden. Hat man die richtigen Werte gefunden, folgt der Roboter dem Pfad perfekt.

3.3 Abbiegen

Das nächste Problem mit dem wir uns beschäftigt haben ist das Abbiegen. Es gibt beim Abbiegen mehrere Probleme:

- Wie muss man abbiegen um das Labyrinth gründlich und effizient zu durchfahren?
- Wie erkennt man eine Kreuzung?

Durchfahren des Labyrinths Die erste Frage war theoretisch schnell gelöst, der Roboter muss die Links-Hand-Regel beachten, das heißt er hält sich immer an der linken Wand. Daraus schlussfolgert man folgende Priorität für die Abbiegungen: Wenn du

nach links abbiegen kannst biege nach links ab, ist das nicht möglich fahre geradeaus, ist das auch nicht möglich biege rechts ab.

Listing 3: Linke-Hand-Regel Implementation

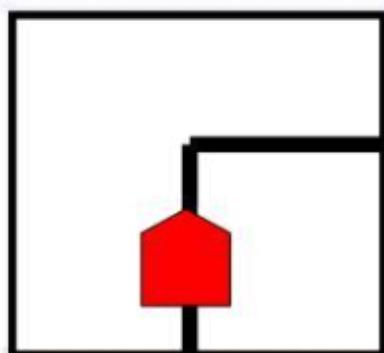
```
if (isEachDiversionOnCrossing[left])
{
    direction = left;
}
else if (isEachDiversionOnCrossing[forward])
{
    direction = forward;
}
else
{
    direction = right;
}
```

Erkennung der Kreuzung Die Sackgasse ist die einfachste 'Kreuzung', denn es ist die einzige Situation wo die Sensorwerte sich von '001100' zu '000000' verändern (wobei 1 für eine Linie unter dem Sensor steht und 0 für weißen Untergrund). Wird eine Sackgasse erkannt dreht sich der Roboter um.

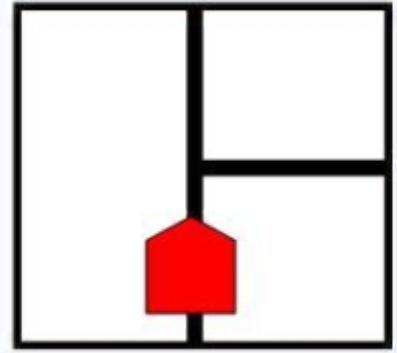
Listing 4: Erkennen von Sackgassen

```
// dead end
else if (getNumberOfCurrentlyWhiteSensors() == sizeof(sensorPins))
{
    direction = backward;
    storeTurnToPath();
}
```

Der nächste Fall ist die Unterscheidung zwischen einer Kreuzung wo man nur rechts abbiegen kann und einer Kreuzung wo man gerade und rechts fahren kann. Beide zeigen zu Anfang das Muster '000111' aber wie weiß man welcher Fall vorliegt.



Nur Rechts



Geraudeaus und Rechts

Die Lösung ist einfach man fährt einfach ein wenig weiter und liest die Sensoren erneut aus. Sind die Sensorwerte nach erneutem einlesen '000000' ist nur eine rechte Abbiegung vorhanden. Sind die Sensorwerte '001100' kann man auch geradeaus fahren und das gleiche gilt auch anders herum für linke Abbiegungen.

Listing 5: Erkennen von Abbiegungen

```
void checkForDiversions()
{
    if (sensorValues[sizeof(sensorPins) - 1] > threshold)
    {
        isEachDiversionOnCrossing[right] = true;
    }
    if (sensorValues[0] > threshold)
    {
        isEachDiversionOnCrossing[left] = true;
    }
}
```

Listing 6: Erkennung von geraden Strecken durch vorwärts fahren

```
checkForDiversions();
if (isEachDiversionOnCrossing[left] || isEachDiversionOnCrossing[right])
{
    direction = diversionChecking;
    startFurtherDiversionCheckingTime();
}
```

Der letzte zu beachtende Fall ist das Erreichen von dem Ende des Labyrinths, wenn alle sechs Sensoren einen schwarzen Untergrund nach dem Vorwärtfahren erkennen, hält der Roboter an.

Listing 7: Erkennung des Ziels

```
if (getNumberOfCurrentlyWhiteSensors() == 0)
{
    direction = none;
}
```

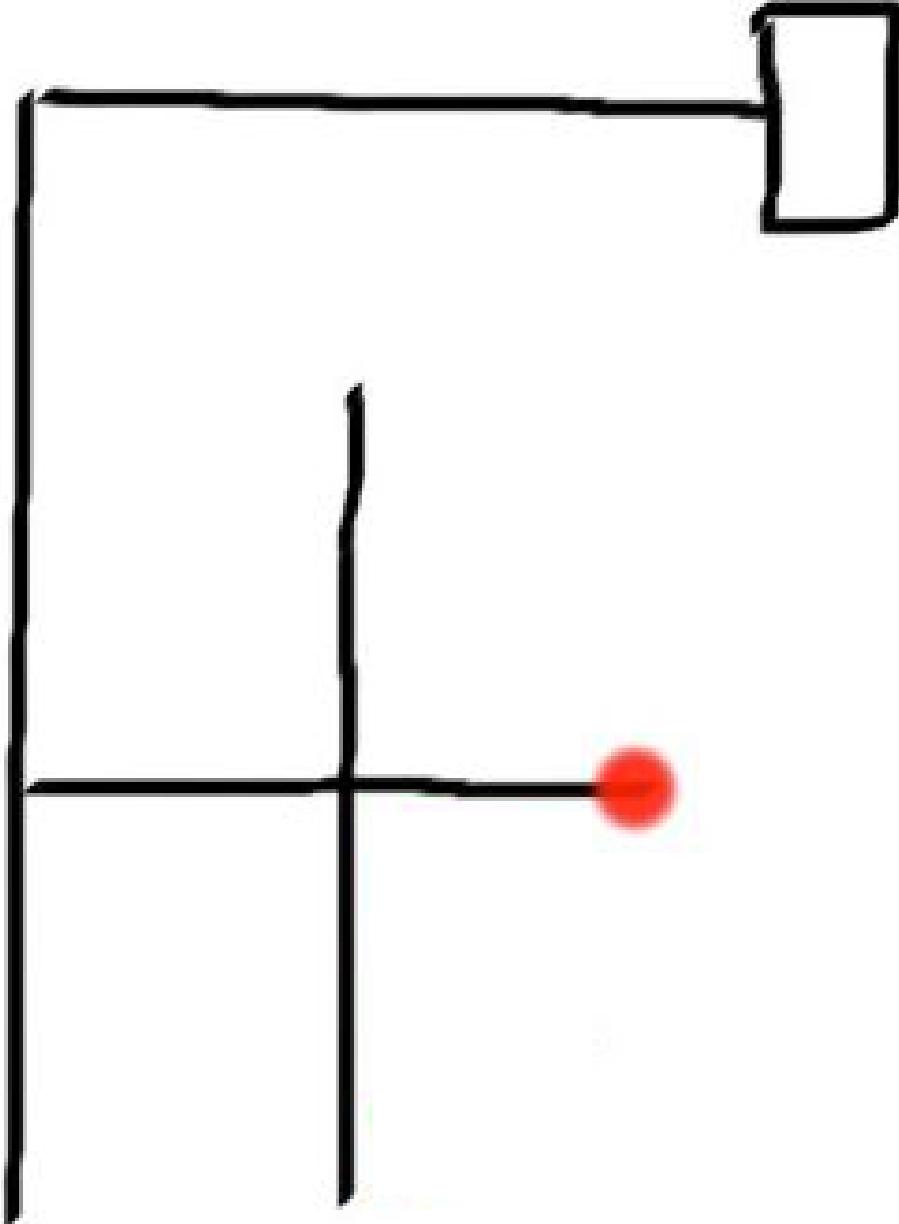
3.4 Lösen des Labyrinths

Welche Schritte müssen zum Lösen des Labyrinths durchgeführt werden?
Es gibt im wesentlichen 2 Schritte. Der erste ist es das Labyrinth zu durchfahren und das Ende zu finden. Der zweite ist es den Pfad so zu optimieren das der Roboter den schnellsten Weg durch das Labyrinth findet

Das Lösen Um das Lösen des Labyrinth zu erläutern werde ich ein einfaches Beispieldabyrinth lösen. Der rote Punkt steht für den Startpunkt des Roboters, der schwarze

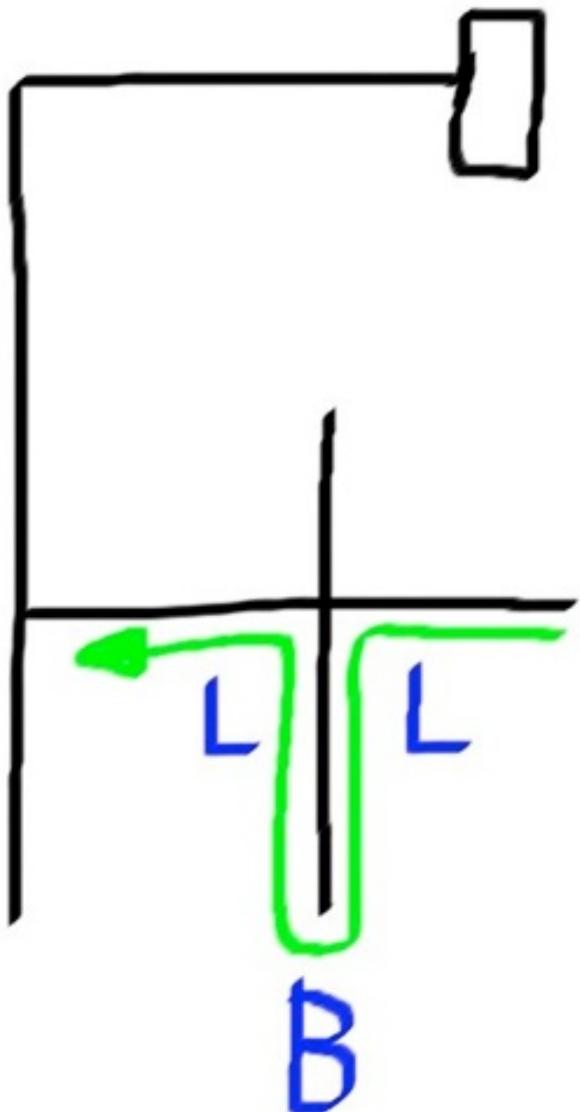
Kasten zeigt das Ende an. Wenn das Labyrinth mit der Links-Hand-Regel abgefahren wird ist der Pfad 'LBLLBSR'.

L = Links R = Rechts S = Geradeaus B = Umdrehen



Wie kann man jetzt diesen Pfad zum schnellsten Weg 'SRR' umwandeln. Für den Anfang stellen wir erst mal fest wo wir falsch abgebogen sind. Ein 'B' signalisiert das der Roboter sich gedreht hat, was bedeutet das er auf den falschen Pfad gefahren ist. Um den Pfad zu optimieren müssen wir das B durch Substitution entfernen.

Zuerst schauen wir uns die ersten drei Abbiegungen im Pfad 'LSBLLBSR'. Diese Bewegungen sieht man im Bild.



Statt links abzubiegen dann umzukehren und nochmal links abzubiegen kann man auch einfach geradeaus fahren. Also ist $LBL = S$. Das ist ein Beispiel einer Optimierung, aber hier ist die ganze Liste:

$LBR = B \quad LBS = R \quad RBL = B \quad SBL = R \quad SBS = B \quad LBL = S$

Lasst uns jetzt den Pfad optimieren: 'LBLLSBR' wir wissen das $LBL=S$ also ist unser neuer Pfad: 'SLBR'. Wir wissen auch das $LBS=R$ also ist unser neuer Pfad: 'SRR'. Wie man sieht ist das der schnellste Weg.

Der Roboter optimiert das Labyrinth während des Fahrens. Jedes mal wenn eine Abbiegung gespeichert wird, überprüft es ob die letzte Abbiegung ein B war, wenn das zutrifft wird der Pfad optimiert. Man muss mindesten 3 Bewegungen kennen, um den Pfad zu optimieren (siehe Listing 9 im Anhang).

3.5 Selbstkalibration

Ein kleines Feature was wir eingebaut haben ist die Selbstkalibration der Sensoren. In der Kalibrationsphase müssen alle Sensoren auf die Extremwerte der Reflexion eingestellt sein, um sich so an das Labyrinth anzupassen. Wir lassen das Fahrzeug also ein paar Mal während der Kalibrationsphase hin und her schwenken so das jeder Sensor auf die Reflexion des Labyrinths eingestellt ist, anschließend dreht sich der Roboter dann soweit das er wieder mittig auf der Linie steht.

Listing 8: Implementation der Selbstkalibration

```
void calibrate()
{
    // make half-turns to have values for black and white without
    // holding it
    for (byte i = 0; i <= 100; i++)
    {
        if (i == 0 || i == 60)
        {
            moveBothMotors(150, backward, 150, forward);
        }

        else if (i == 20 || i == 100)
        {
            moveBothMotors(150, forward, 150, backward);
        }

        qtra.calibrate();
    }
}
```

3.6 Android App

In der Endphase des Projekts wollten wir noch eine Android App entwickeln die sich mit dem Bluetooth Modul des Arduino verbindet und dann das Labyrinth an die App sendet, um das Labyrinth und den schnellsten Weg durch auf dem Smartphone darzustellen. Die Verbindung ist zustande gekommen und die Abbiegungen werden auch mithilfe eines dafür entwickelten Protokolls gesendet, die Darstellung des Labyrinths ist jedoch mangelhaft. Man kann den Roboter mit der App starten und stoppen.

Tabelle 2: Protokoll zum senden des Labyrinths

byte	value
starting	255
the index	0 to 245
the direction	250 to 253
left	250
forward	251
right	252
backward	253
the time in 50 ms	0 to 246 (0 to 12.25 seconds)
finishing and requesting response	254
phone saying received	249
phone request start driving	247
phone request stop driving	248
Arduino request clear the maze view on Android	246

4 Zusammenfassung

In diesem Projekt wurde das Primärziel erreicht. Der Roboter ist in der Lage ein Labyrinth abzufahren und zu lösen. Jedoch ist kein abfahren eines Labiryinths mit Schleifen vollkommen möglich, da wenn er im schlechtesten Fall dort nur im Kreis fährt. Das Darstellen des Labiryinths auf einem Smartphone funktioniert bei Projektabgabe jedoch noch nicht. Um das Projekt zu vervollständigen müsste die App funktionieren. Weitere Features wären die Implementation eines anderen Lösungsalgorithmen, sowie das zurück finden vom Ziel zum Ausgangspunkt. Abschließend lässt sich sagen das die Projektarbeit gelungen ist und ich mit dem Ergebnis zufrieden bin.

5 Erklärung der Urheberschaft

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Ort, Datum

Unterschrift

Literatur

- [1] *Arduino library for the Pololu QTR reflectance sensors.* <https://github.com/pololu/qtr-sensors-arduino>. Abgerufen am: 6. Juli 2017.
- [2] *Repository von unserem Projekt.* <https://github.com/JanBakunin/maze-solving>.
- [3] *SoftwareSerial Library.* <https://github.com/arduino/Arduino/tree/master/hardware/arduino/avr/libraries/SoftwareSerial>. Abgerufen am: 6. Juli 2017.
- [4] *Vorderseite des Arduino Motor Shield.* https://www.arduino.cc/de/uploads/Main/MotorShield_R3_Front.jpg.jpg. Abgerufen am: 6. Juli 2017.
- [5] *Vorderseite des Arduino Uno.* https://wiki.eprolabs.com/index.php?title=Arduino_UNO. Abgerufen am: 6. Juli 2017.
- [6] *Vorderseite des QTR-8A Sensors.* <https://a.pololu-files.com/picture/0J643.1200.jpg?c2f11f34cc015dde43b09d87f0eb85e5>. Abgerufen am: 6. Juli 2017.

Listing 9: Implementierung der Vereinfachung

```
void simplifyMaze()
{
    if (pathLength < 3 || simplePath[pathLength - 2].direction != backward)
    {
        return;
    }

    int totalAngle = 0;

    for (byte i = 1; i <= 3; i++)
    {
        switch (simplePath[pathLength - i].direction)
        {
            case right:
                totalAngle += 90;
                break;
            case left:
                totalAngle += 270;
                break;
            case backward:
                totalAngle += 180;
                break;
        }
    }
    totalAngle = totalAngle % 360;
    switch (totalAngle)
    {
        case 0:
            simplePath[pathLength - 3].direction = forward;
            break;
        case 90:
            simplePath[pathLength - 3].direction = right;
            break;
        case 180:
            simplePath[pathLength - 3].direction = backward;
            break;
        case 270:
            simplePath[pathLength - 3].direction = left;
            break;
    }

    simplePath[pathLength - 1].direction = none;
    simplePath[pathLength - 2].direction = none;
    pathLength -= 2;
}
```