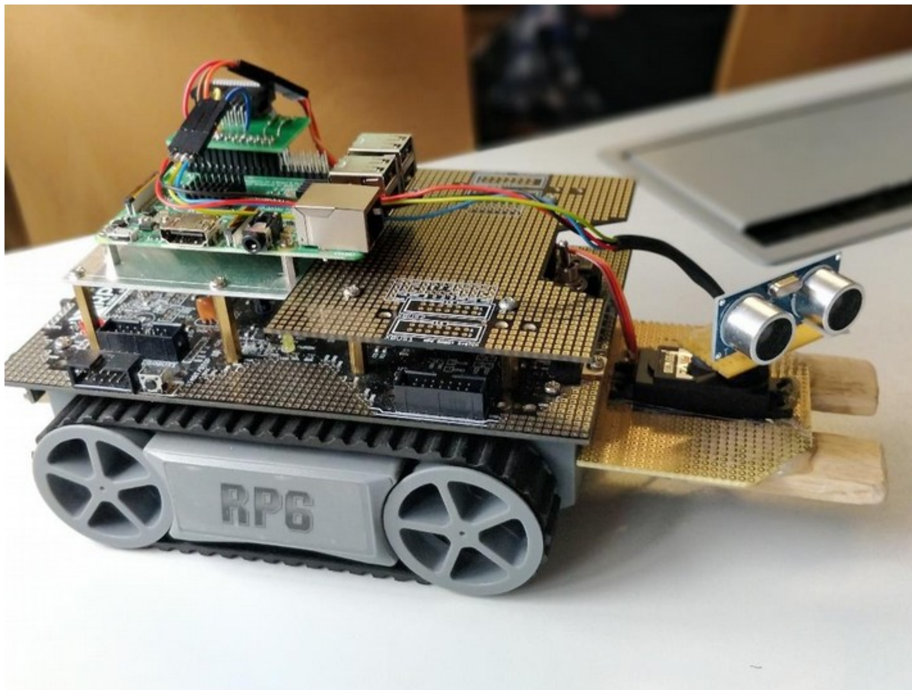
PROJEKTDOKUMENTATION

Ubiquitious Computing Sommersemester 2018 –
MR ROBOTO



Gruppe „Piel Roja“:

-
-
-
-
-

INHALT

PROJEKTPLANUNG 3

FINDUNGSPHASE 3

ZIELSETZUNG 3

BESTELLUNG 4

SYSTEMBESTANDTEILE 5

SYSTEMAUFBAU 5

RASPBERRY PI 5

RP6 BIBLIOTHEK 6

SENSOREN 8

Ultraschallsensor 8

Servo-motor 11

TECHNISCHE BESCHREIBUNG 13

PYTHONSKRIPTE 13

Ultraschallsensor 13

Servor-Motor 16

DATENÜBERTRAGUNG PYTHON ZU JAVA 18

ALGORITHMEN 19

right hand – algorithmus 20

pledge – algorithmus 25

VISUALISIERUNG 31

APPLIKATION ZUM FERNSTEUERN DES ROBOTERS 34

GRUNDLEGENDE IDEE 34

DRAHTLOSE VERBINDUNG 34

KOMMUNIKATION JAVA PROGRAMM UND RP6 36

AUFBAU DER APPLIKATION 37

Startseite 38

Auswahlseite 39

Steuerseite 40

Algorithmenseite 41
FAZIT 42
QUELLENVERZEICHNIS 42

PROJEKTPLANUNG

FINDUNGSPHASE

Nachdem die Gruppenmitglieder feststanden, ging direkt das Brainstorming los, um festzulegen mit welchem Themengebiet sich unser Projekt beschäftigen sollte. Anfänglich schwirrten, durch die verschiedenen Anwendungsfälle, die das Gebiet Ubiquitious Computing einschließt, eine Menge verschiedene Vorschläge durch den Raum.

Schnell kristallisierten sich jedoch zwei, recht unterschiedliche, Ideen heraus:

Entweder wollten wir uns mit einer Spiele-App beschäftigen, die ein Quiz zu Fragen der verschiedenen Module im Studium dargestellt hätte oder einen der, von Professor Knauth zu Verfügung gestellten, RP6-Robotor zu verwenden.

Nach einer Diskussion über die Vor- und Nachteile der beiden Projekte, fiel die Wahl auf eine Projektarbeit mit dem RP6-Roboter. Klar auf der Hand lag, dass wir uns für ein Projekt mit dem RP6 Roboter einfacher in einzelne Projektteams aufteilen und somit gleichzeitig an den verschiedenen Einzelheiten arbeiten konnten.

ZIELSETZUNG

Während der ersten Projektbesprechung legten wir fest welche Aufgaben unser RP6 erfüllen sollte. Zwei Punkte stellen hierbei die Eckpfeiler unseres Projektes dar:

In erster Linie war unsere Idee, dass der RP6 eigenständig, unter Verwendung von Wegfindungsalgorithmen den Ausweg aus einem Labyrinth finden soll.
Hierbei kommen, ein Ultraschallsensor, zum Abmessen der Entfernungen zu den Wänden, und ein Servomotor, zum Einstellen der Blickrichtung, zum Einsatz.

Des Weiteren soll die Möglichkeit bestehen den Roboter mit Hilfe einer Applikation für ein Android Smartphone fernzusteuern und die jeweiligen Algorithmen zu starten.

Als zusätzliche Option schwebte uns eine Visualisierung am Rechner vor. Diese sollte den zurückgelegten Weg des Roboters aufzeichnen und anschließend simpel am Rechner darstellen.

Um einerseits gute Kommunikation innerhalb der Gruppe zu gewährleisten und andererseits einzelne Meilensteine, Aufgaben und Ideen zu verteilen, benutzen wir die web-basierte Projektmanagementsoftware Trello. Diese bietet intuitiv die Möglichkeit verschiedene Aufgaben und Checklisten einzelnen oder auch mehreren Personen zuzuweisen, sodass jedes Gruppenmitglied immer die Möglichkeit hat auch den momentanen Stand des Gesamtprojektverlaufs einzusehen.

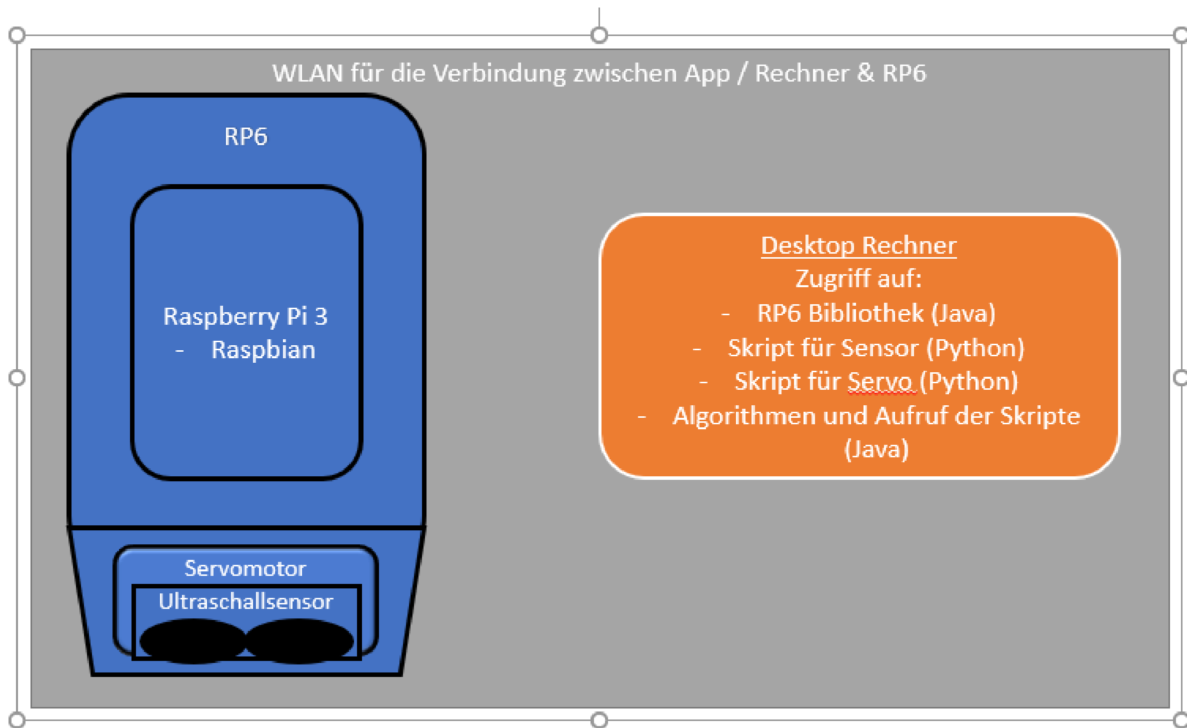
BESTELLUNG

Nachdem wir festgestellt hatten welche Bauteile uns im Labor zu Verfügung stehen und welche fehlten, ging es an die Recherche für die benötigten Teile. Uns fehlten ein Ultraschallsensor und ein Servomotor, um die jeweilige Blickrichtung einzustellen. Kleinteile, wie Schrauben und Abstandshalter zur Montage der beiden Anbauteile, besorgten wir im Fachhandel. Servomotor und Sensor wurden über Prof Knauth bestellt.

Leider ließ sich der mit der ersten Bestellung gelieferte Servomotor nicht, wie gewünscht von 0 – 180° bewegen, was zu Komplikationen führte. Zwischenzeitlich wurde dann auf eine Variante mit 3 Ultraschallsensoren ausgewichen. Mit einer zweiten Bestellung konnte dieses Problem aber behoben werden. Nun war es möglich mit dem auf dem Servomotor montierten Ultraschallsensor in alle drei relevanten Richtungen zu blicken.

SYSTEMBESTANDTEILE

SYSTEMAUFBAU



Die einzelnen Komponenten, sowie Anbindung und Schaltpläne dieser, werden im jeweiligen Abschnitt nochmal genauer betrachtet. Grundsätzlich besteht unser System aus drei Teilen:

1. RP6 Roboter mit Raspberry Pi 3 und angeschlossenem Ultraschallsensor
2. Rechner, um im laufenden Betrieb auf den Raspberry Pi bzw. den Roboter zuzugreifen
3. App zum Fernsteuern des Roboters

RASPBERRY PI

Auf dem Gehäuse des RP6 ist, über das mittgelieferte Board, ein Raspberry Pi 3 angeschlossen. Dieser verfügt über ein eingebautes WLAN Modul, welches eine Verbindung über 2,4GHz in den Standards b /g /n aufbauen kann. Als Betriebssystem kommt Raspbian in der Version 8 (Jessie) zum Einsatz. Hierbei handelt es sich um ein extra auf den Raspberry Pi zugeschnittenes Betriebssystem, das auf der Debian Linux-Distribution fußt. Java und Python stehen in Raspbian ab Installation direkt zur Verfügung, somit muss hier nichts zusätzlich installiert werden.

Der Pi bildet das Herzstück unseres Systems, da auf ihm alle zum Betrieb relevanten Skripte, Bibliotheken und Java Klassen liegen, die wir zum Steuern und Starten der Algorithmen benötigen.

RP6 BIBLIOTHEK

Diese Bibliothek wurde von Yeliz Akdas für den RP6 entwickelt. Unter Verwendung eines Interface können somit via I2C Bus Befehle an den RP6 gesendet werden.

Die Bibliothek stellt verschiedene Methoden zur Verfügung mit denen sich dann der RP6 steuern lässt.
Leider wurden nicht alle Methoden getestet, was dann zu Problemen beim einfachen Drehen des Roboters führte. Welche wir aber mit direkter Ansprache der jeweiligen Ports korrigieren konnten.

Kleinere Ungenauigkeiten bei der Steuerung des RP6 (ob per App oder eigenständig im Rahmen der angewendeten Algorithmen) sind nicht direkt auf die Bibliothek zurückzuführen, sondern resultieren aus der unterschiedlichen Umdrehungszahl und Reaktionszeit der eingebauten Motoren, welche über die Zeit unterschiedlich stark verschleißen.

Hier ein kurzer Auszug der enthaltenen Methoden:

```
144⊕      * Distance traveled: Right.
149⊕     public int getDistRigtL() throws IOException {
152
154⊕      * Status register for drive system.
166⊕     public int getMotionStatus() throws IOException {
169
171⊕      * Currently set PWM value: Left.
176⊕     public int getPowerLeft() throws IOException {
179
181⊕      * Currently set PWM value: Right.
186⊕     public int getPowerRight() throws IOException {
189
191⊕      * Currently set LED values.
196⊕     public int getLEDs() throws IOException {
199
200
202⊕      * Encoder Reading Left: Enc. Counts / 200ms.
207⊕     public int getSpeedLeft() throws IOException {
210
212⊕      * Encoder Reading Right: Enc. Counts / 200ms.
217⊕     public int getSpeedRight() throws IOException {
220
222⊕      * Status Register 1 for Bumpers and Obstacle
237⊕     public int getStatus1() throws IOException {
256
258⊕      * Status Register 2 for Bumpers and Obstacle.
271⊕     public int getStatus2() {
290
293⊕      * @param speed target speed.
310⊕     public void move(int speed, int dir, int distance, boolean blocking) throws IOException, InterruptedException {
325
327⊕      * With this function, the target speed of the two motors is set.
344⊕     private void moveAtSpeed(int speedLeft, int speedRight) throws IOException, InterruptedException {
349
351⊕      * Go straight to the function.
358⊕     public void moveAtSpeedDirectly(int speed, int dir) throws IOException, InterruptedException {
363
365⊕      * Speed left.
382⊕     public void moveAtSpeedLeft(int speed, int divider, int dir) throws IOException, InterruptedException {
389
391⊕      * Speed right.
407⊕     public void moveAtSpeedRight(int speed, int divider, int dir) throws IOException, InterruptedException {
414
416⊕      * Disconnection of the power of ACS, the encoder, the motor current sensors
421⊕     public void powerOFF() throws IOException {
424
426⊕      * Before ACS, IRCOMM and motor control is used, it is necessary to called
431⊕     public void powerON() throws IOException {
434
437⊕      * @param speed Targed speed.
443⊕     public void rotate(int speed, int dir, int angle) throws IOException, InterruptedException {
456
458⊕      * Movement of the Robotors for example via a remote control
464⊕     public void sendRC(int adress, int data) throws IOException {
469
```

SENSOREN

ULTRASCHALLSENSOR

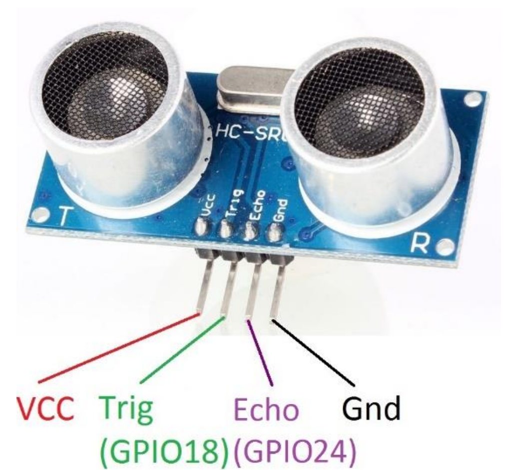
ZUBEHÖR

Für das Projekt ist eine Abstandsmessung nötig um zu erkennen welchen Weg der RP6 Roboter gehen kann. Um eine Abstandsmessung zu ermöglichen, benötigen wir folgendes Zubehör:

1. HC-SR04 Modul
2. Widerstände (330Ω und 470Ω)
3. Jumper Kabel

TECHNISCHE DATEN DES ULTRASCHALLSENSORS HC-SR04

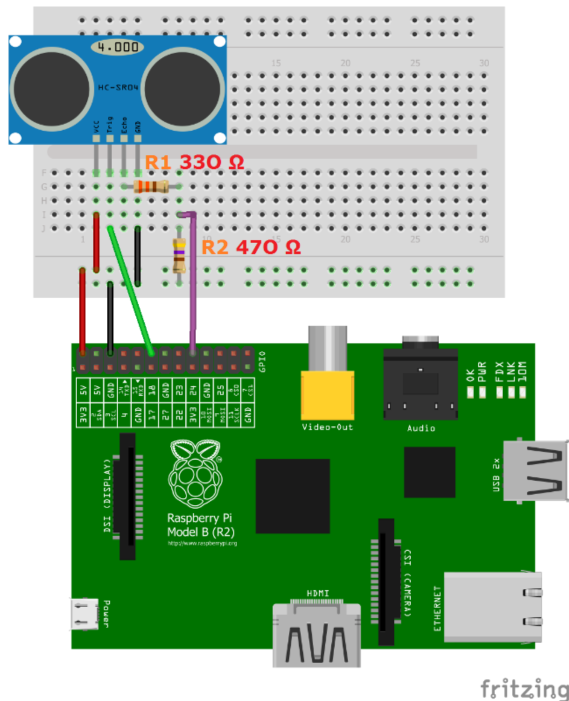
Betriebspannung	5V (+/- 10%)
Strombedarf	ca. 2mA pro Messung
Signal Level	TTL-Pegel
max. messbare Entfernung	ca. 3m
min. messbare Entfernung	ca. 2 cm
Maximalen Messungen pro Sekunde	50
Ultraschallkapseln	zwei (Sender & Empfänger)
Genauigkeit	ca. 3mm
Pinbelegung	Pin 1: VCC Pin 2: Trigger Pin 3: Echo Pin 4: GND



SCHALTPLAN

Auf dem HC-SR04 Modul sind 4 Pins, die mithilfe von einer Lochplatine und Kabeln an den Raspberry Pi angeschlossen bzw. zusammengelötet wird.

1. VCC, Versorgungsspannung 5V
2. Triggereingang, TTL-Pegel (GPIO18)
3. Echo, Ausgang Messergebnis, TTL-Pegel (GPIO24)
4. GND, 0V



Ein 330Ω Widerstand wird an ECHO angeschlossen, dessen Ende zu PIN18 (GPIO24) und über den 470Ω Widerstand eine Verbindung zu PIN6 (GND) besteht.

Die Widerstände sind nötig, da die GPIO Pins nicht mehr als 3.3V vertragen. Auch hat der GPIO Pin eine Verbindung zu GND, damit ein eindeutiges Signal an GPIO24 ankommen kann, sonst wäre bei fehlenden Signalen der Zustand undefiniert. Wird ein Impuls gesendet ist durch die Verbindung mit GND der Zustand des Signals auf 1, bei keinem Impuls wäre der Zustand des Signals 0.

MESSVORGANG

Über den Trigger Anschluss wird die Messung gestartet. Mit einer fallenden Flanke wird ein Messvorgang ausgelöst. Das vorhergehende High-Signal muss dabei eine Mindestzeit von 10 Mikrosekunden anliegen.

Das Modul sendet darauf nach ca. 250µs ein 40 kHz Burst-Signal für die Dauer von 200µs Nachdem das geschehen ist, so geht der Ausgang „Echo" sofort auf den H-Pegel und der Empfang des akustischen Echos wird vom Ultraschallsensor erwartet. Sobald das Echo erkannt wird, so fällt der Ausgang „Echo“ auf den Low-Pegel. Die nächste Messung kann dann nach 20 ms erfolgen.

BERECHNUNG DER ENTFERNUNG

Der Schall legt in der Sekunde 330 Meter zurück. Wenn man die Zeit misst wie lange der Ultraschallsensor braucht sein eigenes Echo zu empfangen, so kann man mit dieser gemessenen Zeit die Entfernung berechnen. Diese errechnete Entfernung muss man dann allerdings noch durch 2 teilen, um nur die einmalige Wegstrecke berechnen zu wollen und nicht den gesamten Hin- und Rückweg.

Die Schallgeschwindigkeit nicht ganz genau 330 Meter pro Sekunde groß, denn die Ausbreitungsgeschwindigkeit ist auch abhängig von der Temperatur. Die genauere Berechnung der Ausbreitungsgeschwindigkeit würde so aussehen:

Ausbreitungsgeschwindigkeit (in Luft) =331,5 + (0,6 * Temp°)

Bei einer Raumtemperatur von 20° ergibt sich somit: 331,5+ (0,6 * 20) = 343,5 m/s Pro gemessener Mikrosekunde wäre der Schall also 0,03434 cm unterwegs gewesen.

Um die gemessene Zeit des Echo-Signals nun in cm umzurechnen, muss die gemessene Zeit zunächst durch 2 geteilt werden (für die einmalige Strecke). Danach muss die Zeit (in uS) nur noch mit 0,03434 cm multipliziert werden.

SERVO-MOTOR

ZUBEHÖR

- Servo Motor
- Jumper Kabel

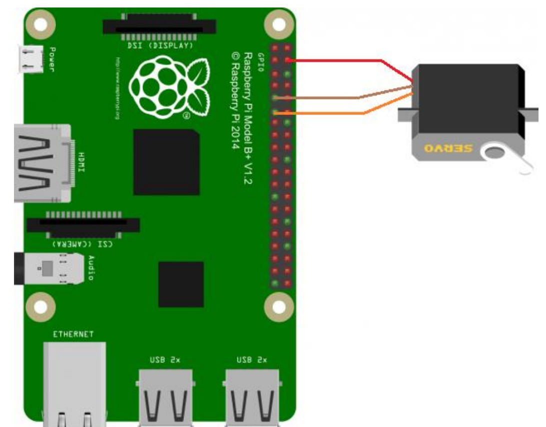
ANSCHLUSS UND INBETRIEBNAHME

In vielen Fällen wird der Servomotor wie folgt angeschlossen:

Rot – kommt an 5V (Pin 4) vom Pi

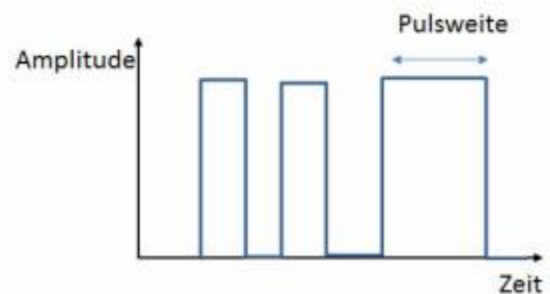
Braun – kommt an GND (Pin 9) vom Pi

Gelb/Orange – an einen freien GPIO Pin (z.B. GPIO17, Pin 11)



FUNKTIONSWEISE

Für den Betrieb bzw. Steuerung des Servomotors wird eine PWM (Puls-Weiten-Modulation) benötigt. die Rotation über die Länge des Impulses gesteuert. Der Winkel des Motors wird über die Länge des Impulses gesetzt, daher bietet sich PWM besonders an, womit wiederholende Signale in gleichmäßigen Abständen geschickt werden (Die Raspberry Pi Python Bibliothek muss installiert sein).



STEUERUNG DES SERVOMOTORS

Bei der Steuerungssoftware werden folgende Methoden benötigt:

Die Methode PWM besitzt zwei Parameter. Einmal den Pin, am dem der Servomotor angeschlossen ist (GPIO 18, Pin 12) und die Frequenz für die Pulsweitenmodulation. Die meisten Servomotoren arbeiten mit 50 Hertz, was auch hier eingestellt wird.

Mit der Methode start wird die Ausgabe des PWM-Signals gestartet. Der Anfangswert des Tastverhältnisses wird als Parameter übergeben.

Mit der Methode ChangeDutyCycle kann das Tastverhältnis und damit die Ausrichtung des Servoarms neu gesetzt werden. Werte zwischen 2,5 und 12,5 können dabei gewählt werden. Dabei steht 2,5 für 0 Grad, 12,5 für 180 Grad. Diese Werte können aber um einige Grad abweichen.

Um auf die Werte zu berechnen, wird folgenden Formel benötigt: DC = Länge/Periodendauer. Der Servomotor verwendet eine Periodendauer von 20ms.

Beispielrechnung für 0 Grad: 0 Grad (0.5 ms): DC = 0.5/20*100 = 2.5%

TECHNISCHE BESCHREIBUNG

PYTHONSKRIPTE

ULTRASCHALLSENSOR

Zur Implementierung der Algorithmen mussten wir die Abstände, die der Roboter zur Wand hat genau berechnen bzw. messen lassen.

Hierfür haben wir ein Skript in der Programmiersprache Python verwendet, welches einen Ultraschallsensor anspricht und uns als Rückgabewert den jeweiligen Abstand vom Sensor zur Wand des Labyrinths liefert.

Allgemeine Erklärung des Skripts:

Um den Sensor über das Skript richtig anzusprechen, müssen wir zuvor die Pins zuweisen und definieren welche als Input und welche als Output fungieren.

Daher auch der Name **GPIO-Pin's - auch General Purpose Input Output genannt.**

```
#GPIO Pins zuweisen
GPIO_TRIGGER = 18
GPIO_ECHO = 24

#Richtung der GPIO-Pins festlegen (IN / OUT)
GPIO.setup(GPIO_TRIGGER, GPIO.OUT)
GPIO.setup(GPIO_ECHO, GPIO.IN)
```

Alle folgenden Aufrufe wurden als Definition distanz() gespeichert:

```
def distanz():
```

Trigger steuert das anschalten des Sensors:

```
GPIO.output(GPIO_TRIGGER, True)
```

Schaltet ihn nach einer Wartezeit von 0,01 ms mit setzen des Werts auf False wieder aus:

```
time.sleep(0.00001)
GPIO.output(GPIO_TRIGGER, False)
```

Echo steuert bzw erkennt mit dem Befehl: **while GPIO.input(GPIO_ECHO) == 0** das noch keine Messung durchgeführt wurde und speichert mit **StartZeit = time.time()** die Startzeit.

Sobald **GPIO.input(GPIO_ECHO)==1** gilt die Messung, durch setzen des Triggers auf Low, als beendet und es wird nochmals die Zeit abgespeichert:

```
while GPIO.input(GPIO_ECHO) == 1:
    StopZeit = time.time()
```

Die Entfernung in cm wird mit Hilfe dieser Formel berechnet:

```
# Zeit Differenz zwischen Start und Ankunft
TimeElapsed = StopZeit - StartZeit
# mit der Schallgeschwindigkeit (34300 cm/s) multiplizieren
# und durch 2 teilen, da hin und zurueck
distanz = (TimeElapsed * 34300) / 2
```

Das ursprüngliche Skript hat vorgesehen die Messungen solange durchzuführen bis diese durch eine Tasteneingabe unterbrochen wird:

```
if __name__ == '__main__':
    try:
        while True:
            abstand = distanz()
        os.system('python hello.py ' +str(abstand))
            print ("Gemessene Entfernung = %.1f cm" % abstand)
            time.sleep(1)

        # Beim Abbruch durch STRG+C resetten
    except KeyboardInterrupt:
        print("Messung vom User gestoppt")
        GPIO.cleanup()
```

Da wir lediglich eine Messung beim Überprüfen der Distanz brauchen nachdem der Roboter gefahren ist, haben wir auf die while Schleife verzichtet und lediglich den Abstand mittels print als Wert zurückgeben lassen:

```
if __name__ == '__main__':

        abstand= distanz()
            print (abstand)
            GPIO.cleanup()
```

Um eventuelle Ungenauigkeiten bei der Messung zu kompensieren, nahmen wir statt der oberen Main-Methode diese um den Median aus fünf Messungen darzustellen

```
if __name__ == '__main__':

            list = [0,0,0,0,0]

            for i in range(len(list)):
                list[i] = distanz()

            list.sort()
            print (list[2])

            GPIO.cleanup()
```

SERVOR-MOTOR

Um den Servomotor wie gewünscht zu steuern müssen erst einige Bibliotheken geladen werden, um zum Beispiel den GPIO Pin anzusteuern, Wartezeiten einstellen und Übergabeparameter abzufangen.

```
1   # GPIO initialisieren
2   #!/usr/bin/env python
3
4   import RPi.GPIO as gpio
5   import time
6   import os
7
8   from sys import argv
```

Da wir den GPIO an PIN 12 am PI angeschlossen haben und für die Steuerung Pulsweitenmodulation brauchen, die eine Frequenz von 50Hz benötigt, wird im folgenden Codeabschnitt gezeigt wie dieser GPIO Pin konfiguriert werden muss. Mithilfe einer neuen Variable namens „servo“ mit der man die GPIO-Methoden aufrufen kann, kann der Servomotor nun angesprochen werden

```
9
10  # Servo-GPIO (PWM-GPIO 18, Pin 12)
11  servopin = 17
12
13  gpio.setmode(gpio.BCM)
14  gpio.setup(servopin, gpio.OUT)
15
16  # PWM-Frequenz auf 50 Hz setzen
17  servo = gpio.PWM(servopin, 50)
```

Als erstes wird der Servomotor in die Startposition gebracht, dies geschieht mit folgender Methode:

```
18
19  # PWM starten, Servo auf 0 Grad
20  servo.start(2.75)
```

Da der Servomotor vom Ultraschallsensorskript aufgerufen wird, muss der Übergabeparameter abgefangen und in eine Variable („n“) zugewiesen werden. Dieser Parameter bestimmt in welche Position sich der Servomotor befinden soll.

```
21
22  try:
23      n = int(argv[1])
24  except IndexError:
25      print('missing argument')
26  except ValueError:
27      print('argument must be an integer')
```

Als nächstes wird überprüft welcher Wert der Variable „n“ zugewiesen ist. Hier gibt es 3 Möglichkeiten bzw. Richtungen. Um den Servomotor nach rechts zu positionieren wird der Wert 0, für geradeaus 90 und für links 180 benötigt. Sobald eines dieser Bedingungen erfüllt ist, wird die Methode servo.ChangeDutyCycle(x) aufgerufen um die Richtung des Motors zu ändern.
Zum Schluss wird eine Wartezeit von 700ms verwendet, da der Servomotor eine gewisse Zeit braucht um die Richtung zu wechseln. Diese wird benötigt um den GPIO nicht zu beenden während der Motor sich dreht, da sonst der Servomotor falsch positioniert wird. Zum Schluss wird dann noch das Ultraschallsensorskript aufgerufen um die Entfernung zu den Wänden zu messen.

```
28  else:
29      # 90 Grad
30      if n==90:
31       servo.ChangeDutyCycle(7.25)
32
33      # 180 Grad
34      if n==180:
35       servo.ChangeDutyCycle(11.3)
36
37      # 0 Grad
38      if n==0:
39       servo.ChangeDutyCycle(2.75)
40
41  time.sleep(0.7)
42  gpio.cleanup()
43  os.system("python ultraschallsensor_entfernung.py")
```

DATENÜBERTRAGUNG PYTHON ZU JAVA

Da die Steuerung des Roboters über die RP6 Bibliothek bzw. die von uns implementierten Java Klassen für die Algorithmen funktioniert, mussten wir uns eine Möglichkeit überlegen wie die Werte aus den beiden Python – Skripten (für Ultraschallsensor und Servomotor) in Java übernommen werden konnten. Die einfachste Lösung bot sich über den, in Java integrierten, Prozessaufruf. Hierbei erstellt man sich ein Prozess Objekt über welches sich dann per „Runtime.getRunTime().exec(„python *dateiname*")"das gewünschte Skript aufrufen lässt. Unter Verwendung eines BufferedReader bzw. eines InputStreamReader kann man dann den Rückgabewert des Skriptes einlesen und über eine Variable an die nötigen Stellen verteilen. Es ist darauf zu achten, dass nach Abruf der Werte aus dem Skript sowohl Reader als auch der Prozess wieder geschlossen werden, da es sonst zu Problemen im Ablauf der Algorithmen kommen kann.

Zusätzlich kommt es beim Aufrufen der Python Skripte leider zu leichten Verzögerungen, was dazu führt, dass die Werte des Ultraschallsensors nicht richtig bzw. zeitnah übernommen werden. Wir konnten dieses Problem, unter zu Hilfenahme eines Sleep-Timers in Java, in den Griff bekommen, nichtsdestotrotz kommt es bei manchen Manövern des Roboters zu kleinen Ungenauigkeiten.

Beispiel Code der Methode zum Übertragen der gemessenen Entfernung des Ultraschallsensors:

```
3  import java.io.BufferedReader;
4
5  import java.io.IOException;
6  import java.io.InputStreamReader;
7
8  public class InputStreamPython {
9      public static float distance_value;
10
11     public static float distance(double servo) throws IOException {
12
13         Process p = Runtime.getRuntime().exec("python ultraschallsensor_entfernung.py");
14
15         BufferedReader reader = new BufferedReader(new InputStreamReader(p.getInputStream()));
16
17         distance_value = Float.valueOf(reader.readLine());
18
19         reader.close();
20         p.destroy();
21         return distance_value;
22     }
23
```

ALGORITHMEN

Bei unserer Recherche zur Wegfindung stießen wir auf verschiede Algorithmen, die mit unterschiedlichen Ansätzen und daraus resultierender Effektivität unserem Roboter ermöglichen den Weg aus einem Labyrinth zu finden.

Fünf der am häufigsten genannten Algorithmen sind uns besonders ins Auge gestochen. Wir entschieden uns zwei der Algorithmen zu implementieren (Right Hand & Pledge) - diese sind im Anschluss an diesen Absatz ausführlicher beschrieben:
(Die folgenden drei kurzen Erklärungen sind frei von Wikipedia übernommen)[1]

1. Maus – Algorithmus:
   Ein eher lächerlicher Ansatz, da es sich hierbei um eine rein zufällige Wegwahl handelt. Es wird schlicht und ergreifend so lange geradeaus gelaufen bis man auf eine Kreuzung trifft, dort wird zufällig entschieden in welche Richtung man seinen Weg fortsetzt.
   So wird zwar immer der Ausgang gefunden, aber es kann mitunter sehr lange dauern.

2. Algorithmus von Gaston Tarry:
   Durch Markieren der benutzten Wege, kann mit diesem Algorithmus auch ein Irrgarten von außen betreten werden, um beispielsweise ein Ziel im Inneren zu finden.
   I. Wenn ein Gang betreten wird, markiert man den Eingang mit dem Wort *Stopp* – Es darf niemals ein Gang betreten werden, der mit *Stopp* markiert ist.
   II. Wird eine Kreuzung zum ersten Mal betreten (Gang hat keine Markierung), markiert man den Gang, aus dem man kommt mit *zuletzt*.
   III. Kommt man an eine Kreuzung, die markierungslos ist, kann man frei wählen welchen Gang man als nächstes betritt. Wenn es keine unmarkierten Gänge mehr gibt, wird der weg eingeschlagen, den man mit *zuletzt* markiert hatte.

[1] https://de.wikipedia.org/wiki/L%C3%B6sungsalgorithmen_f%C3%BCr_Irrg%C3%A4rten (letzter Zugriff: 11.06.2018)

3. Trémaux- Algorithmus (Spezialfall des Algorithmus von Tarry):
   Auch hier werden Markierungen benutzt, um den Weg aus dem Irrgarten zu finden.
   Für einen Weg gibt es drei Möglichkeiten: unbesucht, einfach oder zweifach markiert.
   I. Freie Wahl der Startrichtung, sofern überhaupt möglich
   II. Jeder benutze Gang wird von Kreuzung zu Kreuzung mit einem Strich am Boden markiert. An jeder nichtmarkierten Kreuzung freie Wahl des nächsten Ganges, markieren nicht vergessen.
   III. Trifft man auf eine markierte Kreuzung und der Gang, aus dem man kommt ist nur einmal markiert, geht man genau auf diesem Weg zurück und markiert ihn ein zweites Mal. Sollte dies nicht der Fall sein, wählt man den Gang, der am wenigsten Markierungen hat und markiert ihn, wie gehabt.

   Wenn man am Ziel ankommt, kann man den direkten Weg zum Startpunkt wiederfinden, indem man den „ein-Strich-Markierungen“ folgt.
   Falls der Irrgarten keinen Ausgang hat, erreicht man letztlich den Starpunkt und alle Gänge sind mit genau zwei Strichen markiert.

RIGHT HAND – ALGORITHMUS

Damit unser Roboter eigenständig durch ein Labyrinth fahren kann und auch den richtigen Weg bis zum Ausgang findet, brauchen wir dazu Algorithmen, die den Weg aus einem Labyrinth rausfinden. Der erste Algorithmus wird der Right-Hand Algorithmus sein. Dieser Algorithmus ist nicht perfekt, da er mit einigen Problemen, die in einem Labyrinth auftauchen könnten, nicht umgehen kann. Eines der Probleme wäre, dass wenn sich eine Säule in dem Labyrinth befindet, der Roboter sich die ganze Zeit an der Säule halten würde und sich sozusagen die ganze Zeit im Kreis drehen würde. Das liegt daran, dass man sich den Right-Hand am besten vorstellt, indem man seine rechte Hand ausstreckt und permanent mit der Hand in Kontakt zu Wand bleibt. Ein weiteres Problem bzw. Voraussetzung die gegeben sein muss, ist das der Eingang und der Ausgang mit einander verbunden seine müssen, da beim Right-Hand ein permanenter Kontakt zur rechten Wand herrschen muss, kann auch nur dementsprechend ein Labyrinth gelöst werden, bei dem der Eingang und Ausgang mit einander verbunden ist. Trotz den zwei Nachteilen haben wir uns für den Right-Hand-Algorithmus entschieden, da der Algorithmus einfach in der Umsetzung mit unseren Sensoren ist. Bei Erfüllung der Voraussetzungen wird der Lösungsweg immer gefunden.

Den Right-Hand Algorithmus mussten wir dann speziell für den RP6-Roboter anpassen. Die Anpassungen bezog sich auf den minimalen und den maximalen Abstand den der Roboter zu den Wänden, an der er sich orientiert, einhalten muss. Die minimalen Abstände sind spezifisch je nach Richtung (links, gerade aus, rechts) abgestimmt. Die einzelnen Abstände werden weiter unten an Hand des Quellcodes genauer erklärt. Der maximale Abstand liegt bei 4000, dieser Wert kommt dadurch zu Stande, dass der Ultraschallsensor manchmal Fehlwerte schickt und sich dieser Wert bei zwischen 3300-3400 befindet.

Als nächstes müssten wir noch das geradeaus fahren definieren, also wie weit bzw. wie lange der Roboter geradeaus fahren soll. Die Definition bzw. die Dauer wie lange der Roboter gerade ausfahren soll, haben wir mit TimeUnit.MILLISECONDS.sleep(1200) gesetzt. Dadurch wird das Programm für 1,2

Sekunden „schlafen gesetzt“ und führt danach weiter den Code aus. Eigentlich ist so etwas nicht gut für den Programmfluss, aber das Problem, dass wir mit der Rp6Lib haben ist der, das die Kommunikation zwischen den Python Skripten und den Java Methoden für den Roboter zu langsam sind. Dadurch entstand oft das Problem, dass der Roboter weiter geradeaus fuhr obwohl vorm ihm eine Wand war. Das lag daran, dass die neuen Werte noch nicht zum Verarbeiten verfügbar waren. Das kurze Codestück macht genau die beschriebenen Schritte, also als erstes wird die Methode moveAtSpeedDirctly() des RP6 aufgerufen. Durch diesen Methodenaufruf fährt der Roboter solange geradeaus bis eine neue Methode zum Bewegen des Roboters aufgerufen wird oder die Methode stop() aufgerufen wird. Durch das, dass zwischen diesen zwei Methoden ein „schlafen“ des Programms von 1,2 Sekunden erzwingt wird, fährt der Roboter jedes Mal einheitlich 1,2 Sekunden gerade aus.

```
rp6.moveAtSpeedDirectly(100, 1);
TimeUnit.MILLISECONDS.sleep(1200);
rp6.stop();
```

Auf diesen Wert sind wir gekommen, da der Roboter mit 1,2 Sekunden ungefähr eine Roboter Länge geradeaus fährt.

Desweitern enthält der Right-Hand Algorithmus Codezeilen, die auch wichtig für die einfache Visualisierung, des gefahren Weges zuständig sind. Dazu wird weiter unten zum Thema Visualisierung mehr erklärt.

Im nächsten Abschnitt wir Schritt für Schritt der Quellcode des Right-Hand Algorithmus beschrieben und erklärt.

Als erstes kommen wir zu Main Methode. Die Main Methode erzeugt als aller erstes ein Objekt von RP6RobotI2C, dadurch erhalten wir den Zugriff auf alle Methoden, die für den RP6 Roboter vorgegeben waren. Mit der Methode powerOn() starten wir den RP6, damit wir weiter fortfahren können. Damit wir den Roboter auch jederzeit stoppen können, haben wir auch einen „keyBoardListener“ implementiert, dieser funktioniert genauso wie der „UDP-Listener“ und dieser wurde schon genauer erklärt und deswegen werden wir hier dazu nicht weiter eingehen. Der Hauptteil läuft in der while-Schleife, die wir als THEWHILE deklariert haben. Die Bedingungen der while-Schleife beziehen sich auf die Distanz der jeweiligen Richtung. Die Bedingung ist erfüllt, so lange die Messwerte für die jeweilige Richtung sich unter 100 Zentimeter befinden. Wenn alle Werte gleich 100 oder darüber liegen bzw. der Roboter das Labyrinth verlasen hat wird die while-Schleife beendet. Jedoch kommt es auch hier zu Problemen mit der Umgebung bzw. falschen Werten des Sensors und es ist nicht immer ein Verlass auf das Abbruchkriterium. Deswegen haben wir auch den KeyBoardListener der abhört, ob die Taste \`q\` gedrückt wurde und wenn dies der Fall ist, wird die THEWHILE beendet. In der while-Schleife wird in jedem Durchgang die Methode checkDirection() aufgerufen. Diese Methode beinhaltet den Right-Hand Algorithmus und wird im nächsten Abschnitt genau erklärt.

```
rp6 = new RP6RobotI2C(0x05, I2CBus.BUS_1);
rp6.powerON();
```

```java
Thread keyBoardListener = keyboardReader(in, keys);
char input = 'n';
THEWHILE:
while (!(distanceLeft >= 100 && distanceStraight >= 100 &&
distanceRight >= 100)) {
    checkDirection();
    if (in.contains(QUIT)) {
        input = in.take();
        if (input == QUIT) {
            System.out.println("Console: " + input);
            break THEWHILE;
        }
    }
}
```

Die Methode checkDirection() enthält eine Große if-else Anweisungen die wiederum weitere if-else Prüfungen beinhaltet. Vor der äußersten if-Bedienung wird über den InputStreamPython, dessen Funktion schon erklärt wurde, die Distanz zur rechten Wand geholt. Wenn dieser Messwert Größer als die minDistanceRight, die bei 30 Zentimeter liegt, ist und unter der errorDistance liegt, dann ist die rechte Seite frei. Der Abstand von 30 Zentimeter kommt dadurch zu Stande, dass wenn der Roboter nach links rotiert, es zu keinem Anschlag des Hecks an der Wand kommt und der Roboter sich dann nicht richtig drehen kann. Bevor der Roboter nach rechts rotieren kann muss dieser ein Stück geradeaus fahren damit es beim Rotieren nach rechts zur keiner Kollision mit der Wand kommt. Auch hier wird wieder mit der Methode moveAtSpeedDirectly() und TimeUnit.MILLISECONDES.sleep() gearbeitet. Danach werden die Motoren mit rotate() angesprochen, sodass die eine Seite der Motoren gerade aus drehen und die andere Seite sich rückwärts dreht. Der letzte Parameter ist die Angabe des Winkels, also wie weit sich der Roboter drehen soll. Durch das, dass die Motoren nicht zu hundert Prozent genau ansprechbar sind mussten wir den Roboter um einen 100 Grad Winkel rotieren lassen. Gerade beim Rotieren kommt es immer wieder zu Problemen.

```java
distanceRight = InputStreamPython.distance(0);
System.out.println("Rechts: " + distanceRight);
System.out.println("------------------------");
/*Right Site Check*/
if (distanceRight >= minDistanceRight && distanceRight < errorDistance) {
    /*Before rotate to the right, you have to drive straight out a little
bit.*/
    rp6.moveAtSpeedDirectly(100, 1);
    TimeUnit.MILLISECONDS.sleep(800);
    rp6.stop();

    /*Rotation to the right site*/
```

```
    rp6.rotate(100, RP6RobotI2C.I2C_REG_POWER_LEFT, 100);
```

Nach dem Rechtsrotieren fährt der Roboter geradeaus, jedoch wird davor immer geprüft ob geradeaus frei ist. Dafür wird das Servomotor Skript mit dem Wert 90 Grad aufgerufen.

```
distanceStraight = InputStreamPython.distance(90);
if (distanceStraight >= minDistanceStraight && distanceStraight <
errorDistance) {
    /*drive straight and check the distance to the wall and correct the
    direction*/
    System.out.println("Gerade aus: " + distanceStraight);
    System.out.println("------------------------");
    rp6.moveAtSpeedDirectly(100, 1);
    TimeUnit.MILLISECONDS.sleep(1200);
    rp6.stop();
```

Im nächsten Abschnitt kommt eine weitere äußere if-Bedienung, in der geprüft wird, ob die rechte Seite geschlossen ist bzw. ob distanceRight kleiner als die minDistanceRight ist. Ist das der Fall wird wieder geradeaus geprüft und der Roboter fährt geradeaus. Durch diese if-Bedienung kontrolliert der Roboter abwechselnd die rechte Seite und geradeaus und fährt dadurch solange, wie die rechte Seite geschlossen ist oder vor dem Roboter kein Hindernis auftaucht. Desweitern wird auch der Abstand zur rechten Wand bei jedem Durchgang kontrolliert und bei einem zu großen bzw. geringen Abstand, korrigiert. Das machen wir, indem wir den Roboter in die gewünschte Richtung rotieren und dann ein kleines Stück geradeausfahren lassen und im Anschluss wird der Roboter wieder zurück rotiert damit dieser wieder parallel zur Wand steht.

```
else if (distanceRight < minDistanceRight) {
    /*get and check straight out*/
    distanceStraight = InputStreamPython.distance(90);
    if (distanceStraight >= minDistanceStraight && distanceStraight <
            errorDistance) {
        /*drive straight and check the distance to the wall and correct the
             direction*/
        if (distanceRight <= 9) {
            rp6.rotate(100, 2, 80);
            rp6.moveAtSpeedDirectly(100, 1);
            TimeUnit.MILLISECONDS.sleep(800);
            rp6.rotate(100, RP6RobotI2C.I2C_REG_POWER_LEFT, 80);
        }
```

```
    if (distanceRight >= 26 && distanceRight < 30) {
        rp6.rotate(100, RP6RobotI2C.I2C_REG_POWER_LEFT, 80);
        rp6.moveAtSpeedDirectly(100, 1);
        TimeUnit.MILLISECONDS.sleep(600);
        rp6.rotate(100, 2, 80);
    }
    System.out.println("Gerade aus: " + distanceStraight);
    System.out.println("------------------------");
    rp6.moveAtSpeedDirectly(100, 1);
    TimeUnit.MILLISECONDS.sleep(1200);
    rp6.stop();
```

Der letzte Abschnitt der Right-Hand Implantierung ist der else Fall. Dieser Fall tritt ein, wenn rechts und geradeaus zu ist. Wenn das der Fall ist, dann wird der Servomotor nach links gedreht und der Abstand links eingeholt. Liegt dieser Wert zwischen der minimalen Distanz zur Linke Seite und der errorDistance, dann rotiert der Roboter nach links. Im Anschluss beginnt die Methode „checkDirection“ von neuem.

```
else {
    /*get left distance and check and if it's true then rotate left*/
    distanceLeft = InputStreamPython.distance(180);
    if (distanceLeft >= minDistanceLeft && distanceLeft < errorDistance) {
        System.out.println("Links :" + distanceLeft);
        System.out.println("------------------------");
        rp6.rotate(100, 2, 90);
```

Der Right-Hand Algorithmus enthält auch für das Visualisieren ein paar Zeilen Code auf die wir später im Thema Visualisierung eingehen werden.

PLEDGE – ALGORITHMUS

VORWORT:

Start des Roboters außerhalb des Labyrinths:
„Falls alle Mauern zusammenhängen oder mit der Außenseite verbunden sind – das heißt das Labyrinth ist „einfach zusammenhängend“ – garantiert der Right-Hand Algorithmus, dass man entweder einen anderen Ausgang erreicht, oder wieder zum Eingang zurückkehrt.“

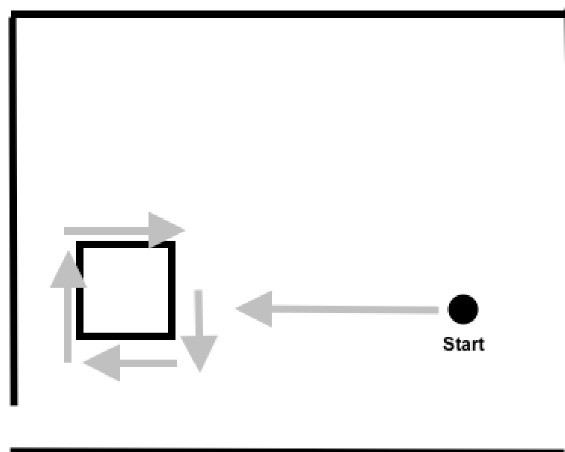
Der Pledge-Algorithmus ist eine Erweiterung des Right-Hand Algorithmus und wurde damals vom 12-jährigen John Pledge konzipiert, um Hindernisse zu umrunden.

Er dient dazu das Problem „eines nicht einfach zusammenhängenden Labyrinths“ auf Grundlage des Right-Hand Algorithmus zu lösen.

PROBLEMSTELLUNG:

Geht man davon aus, dass der Roboter nicht am Eingang/Ausgang des Labyrinths startet, sondern irgendwo in das Labyrinth gesetzt wird, kann es sein, dass der Roboter als erstes auf eine Wand trifft, die nicht mit dem Ausgang zusammenhängt. Hier würde der ursprüngliche Right-Hand dazu führen, dass der Roboter in eine Endlosschleife gerät, da er sich nie von der Säule lösen würde.

Beispiel:



Hier kommt die Erweiterung des Pledge-Algorithmus ins Spiel, der wie folgt definiert ist:

- Zum Start muss eine beliebige Zielrichtung vorgegeben werden
- Die Anzahl der Drehungen bei der man sich beim Vorwärtsfahren vom Ziel wegdreht bzw. darauf zu muss Null sein:
  - Bei einer Drehung gegen Den Uhrzeigersinn wird mit -1 berechnet
  - Bei einer Drehung mit dem Uhrzeigersinn wird mit +1 berechnet

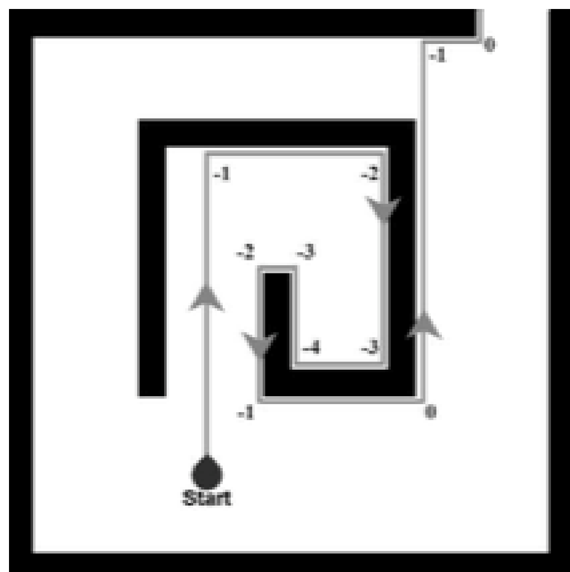
Ist man wieder in Zielrichtung ausgerichtet und ist die Summe der gemachten Drehungen gleich Null, löst man die Hand vom Hindernis und geht wieder geradeaus in Zielrichtung bis man auf das nächste Hindernis trifft.

PSEUDOCODE-BEISPIEL PLEDGE:

```
while(Ausgang nicht erreicht)
      if(Wandkontakt und Winkelzähler != 0)
            folge Wand;

      else
            Geradeaus bewegen;
```

Die Hand wird nur dann von der Wand des Irrgartens genommen, wenn BEIDE Bedingungen erfüllt sind: „Summe der gemachten Drehungen gleich Null“ und „aktuelle Ausrichtung gleich Zielrichtung“. Dadurch vermeidet der Algorithmus, in Fallen zu tappen, die die Form des Großbuchstaben „G“ besitzen. **Siehe Abbildung:**



Startet man nun wie gezeigt in der Grafik und würde lediglich die Zielrichtung beachten würde auch dieser Algorithmus sich, in einer Endlosschleife verfangen.

Sobald man beim Drehzählerwert „-4“ angekommen ist, würde man sich von der Wand lösen und wieder auf die Anfangswand treffen.

IMPLEMENTIERUNG:

Als ersten Gedanken wollten wir aufgrund der zum Teil vielen Informationen im Internet zu diesem Algorithmus, nicht auf dem „Right-Hand“ aufbauen, sondern eine eigene Implementierung hierfür schaffen.

Im weiteren Verlauf ist aber klar geworden, dass eine völlig neue Implementierung widersprüchlich ist, da zur Implementierung des Pledge-Algorithmus lediglich einige Anpassungen des schon implementierten Right-Hand Algorithmus nötig sind.

Anpassungen die am Right-Hand Algorithmus gemacht wurden, um zu Pledge zu erweitern:

- Wir haben eine Variable für die Zahl der gemachten Drehungen eingeführt:

„private static int degree“

- Jede Linksdrehung wird mit -1 gezählt, jede Drehung nach rechts mit +1

- Setter/Getter für diese Variable:

```
public static int getDegree() {
    return degree;
}

public static void setDegree(int degree) {
    Pledge.degree += degree;
}
```

- Als erstes wird abgefragt ob die Zahl der Drehungen null ist und geradeaus frei, solange das der Fall ist soll geradeaus gefahren werden:

```
distanceStraight = InputStreamPython.distance(90);
// 1. Abfrage Ob Winkel 0 und geradeaus frei
while(distanceStraight >= minDistanceStraight && distanceStraight < errorDistance && getDegree() == 0) {
    rp6.moveAtSpeedDirectly(100, 1);
    TimeUnit.MILLISECONDS.sleep(1200);
    rp6.stop();
    setCoordinates();
    labPoints.add(new Point(x, y));
    System.out.println("1. While -geradeaus frei und Winkel=0");
    distanceStraight = InputStreamPython.distance(90);
```

- Wenn geradeaus nicht mehr frei, aber der Winkel(Drehungen) noch null, soll zuerst versucht werden nach links zu drehen (um rechte Hand an die Wand zu bekommen):

  Falls der Fall zutrifft, soll der Winkelzähler -1 „dazu zählen"

```
if (InputStreamPython.distance(180) >= minDistanceLeft && InputStreamPython.distance(180) < errorDistance
        && getDegree() == 0) {
    System.out.println("1. IF (Links Check)");
    rp6.rotate(100, 2, 90);
    setdirectionStatus('l');
    setDegree(-1);
    }
```

- Wenn Drehung gemacht wurde (Winkel ungleich null) soll wie im Right-Hand geprüft werden ob vor ihm frei ist. Direkt im Anschluss wird geprüft ob rechts frei ist:

```
if (InputStreamPython.distance(90) >= minDistanceStraight && distanceStraight < errorDistance
        && getDegree() != 0) {
    // roll straight
    System.out.println("2. IF (Geradeaus wenn Winkel gesetzt)");
    rp6.moveAtSpeedDirectly(100, 1);
    TimeUnit.MILLISECONDS.sleep(1200);
    rp6.stop();
    setCoordinates();
    labPoints.add(new Point(x, y));
    }

     if (InputStreamPython.distance(0) >= minDistanceRight && InputStreamPython.distance(0) < errorDistance
            && getDegree() != 0) {
         System.out.println("4. IF Rechts offen");
        // rotate right davor muss ein kleines stueck vorwaerts gefahren
        // werden
        rp6.moveAtSpeedDirectly(100, 1);
        TimeUnit.MILLISECONDS.sleep(800);
        rp6.stop();
        // rp6.rotate(100, RP6RobotI2C.I2C_REG_POWER_RIGHT, 30);
        rp6.rotate(100, RP6RobotI2C.I2C_REG_POWER_LEFT, 110);
        setdirectionStatus('r');
        setDegree(1);
```

- Letzter Fall, wenn geradeaus und rechts zu aber Winkel ist ungleich Null:

```
else if(InputStreamPython.distance(0) < minDistanceRight && distanceRight < errorDistance
&& InputStreamPython.distance(90)< minDistanceStraight && InputStreamPython.distance(90)< errorDistance
&& getDegree() != 0)
{
    rp6.rotate(100, 2, 90);
       setdirectionStatus('l');
       setDegree(-1);

}
```

Diese Abfragen sollen solange wiederholt werden bis die Abbruchbedingung eintritt.

Abbruchbedingung:

- Tastenabbruch durch App/Konsoleneingabe:
    - Um Raumvisualisierung abzuschließen auch ohne erfolgreiches aus dem Labyrinth finden.
- Messungen der Abstände in alle 3 Richtungen:
    - Falls festgelegter Wert überschritten wird, gilt das Labyrinth als verlassen und das Programm wird beendet.

Diese Bedingung wurde in der Main-Methode mithilfe einer while-Schleife realisiert.

Die Idee war, solange die Abstandsmessungen nach links, rechts oder geradeaus einen gewissen Wert nicht überschritten haben, ist das Labyrinth nicht verlassen. Sobald ALLE 3 Werte größer sind als dem von uns vorgegebenen Wert gilt das Labyrinth als verlassen:

```
THEWHILE: while (!(distanceLeft >= 100 && distanceStraight >= 100 && distanceRight >= 100)) {
    checkDirection();
    input = in.take();
    System.out.println("Console: " + input);
    if (input == QUIT) {
        break THEWHILE;
    }
}
```

Um das Labyrinth nach diesem Prinzip als Verlassen zu markieren muss jedoch davor die Größe des Labyrinths bekannt sein. In unserem Beispiel war dies noch zu realisieren da wir die Maße kennen, wenn dies jedoch nicht mehr bekannt ist sollten andere Hilfsmittel wie bspw. eine visuelle Markierung als Abbruchbedingung genommen werden.

Zu Testzwecken unter anderen im freien Raum, indem die Grenzwerte schnell überschritten werden, haben wir auf die selbstständige Abbruchbedingung verzichtet und beenden den Algorithmus mittels Tastenabbruch:

```
while(true){
        checkDirection();
        input= in.take();
        System.out.println("Console:  " + input);
        if(input==QUIT){
                break;
        }
}
```

KOMPILIEREN & STARTEN DER ALGORITHMEN

KOMPILIEREN DER ALGORTIHMEN

Sobald man jegliche Änderungen an den Algorithmen bzw. dem Sourcecode vorgenommen haben, so muss nach der Bearbeitung diese erst nochmal kompilieret werden. In unserem Fall müssen wir erst mal in das Verzeichnis wechseln, indem das Projekt bzw. sich die Sourcecodes befinden.

Das Projektverzeichnis erreicht man vom HOME-Verzeichnis aus mit dem Befehl „cd MR_ROBOTO/RP6Lib/" und in dem Projektverzeichnis sind folgende Dateien vorhanden:

```
pi@rp6pi1:~ $ cd MR_ROBOTO/RP6PiLib/
pi@rp6pi1:~/MR_ROBOTO/RP6PiLib $ ls
bin  lib  runPledge.sh  runRightHand.sh  SaveArrayList.ser  src
```

Die Sourcecodes befinden sind im Ordner src. Mit dem Befehl: „cd src/jpi2c" wechselst man zum Verzeichnis indem sich die Algorithmen und für die Steuerung der Hardware die entsprechenden Sourcecode Dateien befinden.

```
pi@rp6pi1:~/MR_ROBOTO/RP6PiLib $ cd src/jpi2c/
pi@rp6pi1:~/MR_ROBOTO/RP6PiLib/src/jpi2c $ ls
compilePledge.sh         Pledge.java              RP6RobotI2C.java
compileRightHand.sh      RegisterInterface.java   servo.py
InputStreamPython.java   RightHand.java           ultraschallsensor_entfernung.py
```

Sobald man Änderungen an diesen Dateien vorgenommen hat, so muss das entsprechen neu kompiliert werden. Für das Kompilieren haben sind Bash-Skripte vorbereitet, die sich im gleichen Ordner befinden, um das Kompilieren einfacher zu machen. Für das Kompilieren muss einfach von diesem Verzeichnis aus den Befehl „./compilePledge.sh" oder „./compileRightHand.sh" ausgeführt werden, je nach dem welcher Algorithmus verändert wurde.

STARTEN DER ALGORITHMEN

Im Projektverzeichnis sind 2 Bash-Skripte zu finden, zum einen den „runPledge.sh" zum anderen den „runRightHand.sh". Je nach dem welchen Algorithmus man ausführen will. So gibt man in der Kommandozeile den Befehl: „./runPledge.sh" für die Ausführung den Pledge Algorithmus oder auch den Befehl „./runRightHand.sh" für die Ausführung den Right-Hand Algorithmus ein.

```
pi@rp6pi1:~ $ cd MR_ROBOTO/RP6PiLib/
pi@rp6pi1:~/MR_ROBOTO/RP6PiLib $ ls
bin  lib  runPledge.sh  runRightHand.sh  SaveArrayList.ser  src
```

VISUALISIERUNG

Die Visualisierung des Weges, den der Roboter abfährt, hatten wir als Extrafeature geplant. Deswegen haben wir die Visualisierung nur in einer einfachen Form umgesetzt. Die Visualisierung startet mit dem Algorithmus zum Lösen des Labyrinths. Als erstes wird die Grundidee hinter der Visualisierung beschrieben und danach wird anhand des Quellcodes die Visualisierung genauer erklärt.

Die Grundidee besteht darin, dass für jede geradeaus Bewegung, den der Roboter macht, ein statischer x-Wert bzw. y-Wert hochgezählt wird. Mit den x- und y-Werten werden dann anschließend „Point“ Objekte erstellt, die wiederum in eine Arrayliste hinzugefügt wird. Dadurch bekommen wir am Ende des Labyrinths eine Arraylist mit Koordinatenpunkten, die für den abgefahrenen Weg stehen. Jedoch war die Umsetzung nicht so einfach, wie der erste Grundgedanke. Das Problem war, dass wir uns die Bewegung des Roboters in einem Koordinatensystem vorgestellt haben und es in einem Koordinatensystem Bewegungen auf der X-Achse und Y-Achse gibt. Deswegen war auch die Problemstellung, wann wir welchen Wert hochzählen müssen. So kamen wir auf die erste Umsetzungsidee, dass wenn der Roboter geradeaus fährt, der y-Wert um eine Einheit hochgezählt wird. Kommt es zur einer rechts bzw. links Rotation wird der x-Wert um eine Einheit hoch oder runter gezählt.

Anhand des Codes werden wir auf alle Schritte, Probleme und Umsetzungen vom Anfang an bis hin zur Darstellung alles genau erklären. Wie schon genannt hängt die Visualisierung mit dem Right-Hand-Algorithmus zusammen und fängt deswegen auch damit an.

Wie schon erwähnt brauchen wir eine Arrayliste die Point-Objekte sammelt. Desweiteren brauchen wir die statischen int Werten x und y, die auch gleich als Startposition eingesetzt werden. Ein weiterer Parameter ist der „directionStatus“, der für die aktuelle Lage des Roboters steht. Es gibt die Werte 0,1,2,3, die jeweils für die Gradstellung des Roboters steht. Mit der Startposition liegt der Wert von directionStatus bei 0. rotiert der Robert nach rechts, wird directionStatus um 1 addiert und bei einer links Rotation wird um 1 subtrahiert. Der Wert 0 steht für 0 Grad, der Wert 1 für 90 Grad, der Wert 2 für 180 Grad und der Wert 3 für 270 Grad. Danach fängt es wieder bei 0 an. Was diese Werte mit der Visualisierung zu tun haben wird weiter unten genauer erklärt.

```
private static ArrayList<Point> labPoints = new ArrayList<Point>();
private static int x = 300;
private static int y = 400;
/* 0 = 0Grad || 1 = 90Grad || 2 = 180Grad || 3 = 270 Grad ... */
private static int directionStatus = 0;
```

Der Wert „directionStatus“ wird durch die Methode „setdiretionStatus“ gesetzt. Dabei muss man der Methode einen char (`r` oder `l`) Parameter mitgeben. Mit dem Parameter wird mitgeteilt, ob es sich um eine links oder rechts Rotation handelt. Bei einem char ´r´ wird der int-Wert „directionStatus“ bis 3 hochgezählt und danach wieder bei 0 weitergezählt. Bei dem char ´l´ wird dementsprechend heruntergezählt und nach der 0 fängt es wieder bei 3 an.

```java
private static void setdirectionStatus(char rl) {
    if (directionStatus == 3) {
        if (rl == 'r') {
            directionStatus = 0;
        } else {
            directionStatus = 2;
        }
    } else if (directionStatus == 0) {
        if (rl == 'r') {
            directionStatus = 1;
        } else {
            directionStatus = 3;
        }
    } else {
        if (rl == 'r') {
            directionStatus++;
        } else {
            directionStatus--;
        }}}
```

Nun fragen wir uns, warum der Wert „directionStatus“ so eine wichtige Rolle spielt. Durch diesen Wert beschreiben wir die momentane Lage des Roboters, also ob sich der Roboter auf der vertikalen bzw. horizontalen Linie fortbewegt und dementsprechend wird der x oder y-Wert hoch- oder runtergezählt. Ein kleines Beispiel, wird das alles noch einmal ersichtlicher machen. Wir Starten bei dem Punkt (300|400) und der Wert „directionStatus“ liegt bei 0. Der Roboter bewegt sich um zwei Einheiten geradeaus, der aktuelle Punkt liegt nun bei (300|380). Warum hierbei jeweils eine Einheit (10) abgezogen wird, liegt daran, dass wir fürs Darstellen der Punkte ein JFrame verwenden und bei eine JFrame liegt der Punkt (0|0) ganz oben links. Als nächstes rotiert der Roboter nach rechts, dabei wird dann die Methode setdirectionStatus(´r`) aufgerufen und der Wert „directionStatus“ auf 1 gesetzt. Fährt der Roboter nun wieder gerade aus wird die Methode „setCoordinates“ aufgerufen und weil der directionStatus bei 1 liegt, wird nun der x-Wert hochgezählt, aktuelle Punkt z.B. bei (320|380).

```java
private static void setCoordinates() {
    if (directionStatus == 0) {
        y = y - 10;
    } else if (directionStatus == 1) {
        x = x + 10;
    } else if (directionStatus == 2) {
        y = y + 10;
```

```
        } else if (directionStatus == 3) {
            x = x - 10;
        }
    }
```

Die Methode „setdirectionStatus“ wird immer bei einer Rotation im Algorithmus aufgerufen und die Methode „setCoordinates“ nach einer gerade aus Bewegung. Nach dem die Methode „setCoordinates“ aufgerufen wird, wird dann noch in die Arrayliste der aktuelle Punkt hinzugefügt.

```
/*Rotation to the right site*/
rp6.rotate(100, RP6RobotI2C.I2C_REG_POWER_LEFT, 100);
setdirectionStatus('r');

System.out.println("Gerade aus: " + distanceStraight);
System.out.println("------------------------");
rp6.moveAtSpeedDirectly(100, 1);
TimeUnit.MILLISECONDS.sleep(1200);
rp6.stop();
setCoordinates();
labPoints.add(new Point(x, y));
```

Am Schluss wird die Arrayliste mit den Punkten serialisiert, damit wir mit den Punkten weiterarbeiten können.

Die Serialize Datei wird in das Java Projekt „MR_ROBOTO_Visualierung“ eingebunden und deserialiesiert. Die deserialisiert Arrayliste wird dann durch iteriert und mit den jeweiligen Punkten wird ein Graphics2D Objekt befüllt und am Schluss wird dieses Objekt in einem JFrame dargestellt. Der Code zum Visualisieren ist größtenteils selbsterklärend, deswegen werden wir auch nicht mehr genauer auf die einzelnen Zeilen Code eingehen.

APPLIKATION ZUM FERNSTEUERN DES ROBOTERS

GRUNDLEGENDE IDEE

Die Grundidee war es eine, auf Android basierte, App, für die Fernsteuerung des RP6-Roboters, zu entwickeln. Mit der App sollte es möglich sein den Roboter fernzusteuern, um dieses Ziel zu erreichen wird eine drahtlose Verbindung zwischen der App und dem, mit dem RP6-Roboter verbundenem, Raspberry Pi aufgebaut und einzelne Steuerbefehle an ein, auf dem Pi laufenden, Java-Programm gesendet. Das Java-Programm wandelt die empfangenen Befehle, mithilfe der bereits vorhandenen RP6PiLib, in, für den RP6-Roboter ausführbare Befehle, um und ermöglicht so den Roboter direkt zu steuern.

DRAHTLOSE VERBINDUNG

Ursprünglich war es geplant, die Verbindung zwischen der App und dem Pi über eine SSH-Verbindung zu realisieren. Für die Implementierung der SSH-Verbindung wird die JSCH-Library[2] von JCraft, hierbei handelt es sich um eine Java Implementierung des SSH2 Protokolls, welche es ermöglicht eine SSH-Verbindung zu einem SSHD Server aufzubauen und mittels port forwarding Daten an den Server zu senden, in unserem Fall wird die Rolle des Servers von unserem Raspberry Pi übernommen. Da JCraft nur schlecht verständliche Beispiele liefert, haben wir unsere Implementierung der SSH Verbindung auf ein Beispiel[3] von Stackoverflow aufgebaut. Hierbei wird zunächst ein neues JSCH Objekt erstellt, über welches dann, mithilfe der Ip-Adresse, dem Benutzernamen, einem Passwort, und dem zu sendenden Befehl eine Session aufgebaut und der eingetragene Befehl an den Port 22 des Servers weitergeleitet wird. Um Feedback vom Server zu erhalten, wird der InputStream der Session in ein Byte Array weitergeleitet und dieses Byte Array letztlich in einen String gecasted. Sobald der Befehl an den Server weitergeleitet wurde, wird die Verbindung zum Server wieder getrennt.

Da bei der Android Programmierung blockierende Aufgaben, wie beispielsweise der Verbindungsaufbau einer SSH-Verbindung, nicht im Maintask ausgeführt werden dürfen, da diese sonst die komplette App blockieren, werden die oben beschriebenen Schritte in einer extra Klasse namens SSHConnection, welche von der Klasse AsyncTask abgleitet wurde, in der doInBackground() Methode ausgeführt. Das Beispiel von Stackoverflow hält die Session solange aktiv, bis der InputStream der Session keine weiteren Bytes mehr liefert. Um die Wartezeit so gering wie möglich zu halten wird bei uns die Session beendet, sobald der Server Bescheid gibt das die Verbindung erfolgreich aufgebaut wurde, um dieses Ereignis zu überprüfen wird geprüft ob der, aus dem Byte Array gecastete String folgenden String „Connected to Device OK!“ enthält. Ist dies der Fall wird die Session beendet.

[2] http://www.jcraft.com/jsch/ (letzter Zugriff: 05.06.2018)
[3] https://stackoverflow.com/questions/4194439/sending-commands-to-server-via-jsch-shell-channel (letzter Zugriff: 05.06.2018)

Da eine SSH Verbindung stetigen Kontakt zwischen der App und dem Server benötigt und immer wieder Probleme wie Verbindungsabbrüche, welche relativ kompliziert zu behandeln sind, auftreten können, haben wir uns dagegen entschieden die komplette Kommunikation zwischen App und Pi über eine SSH-Verbindung stattfinden zu lassen. Die SSH-Verbindung wird nur verwendet um das gewünschte Java-Programm auf dem Raspberry Pi zu starten, nach dem Programmstart wieder beendet, da dieser Vorgang auf jeden Fall ausgeführt werden muss, um den Roboter zu steuern, ist eine SSH-Verbindung ideal, da über die SSH-Verbindung sichergestellt werden kann, dass der Befehl ausgeführt wurde. Hierfür senden wir als Befehl beim Verbindungsaufbau einen Ausführungsbefehl, mit zugehörigem Pfad, welcher ein Shellskript startet. Das Shellskript bindet die benötigten Ordner in einen Java Classpath und startet danach das gewünschte Programm.

Für das Versenden der einzelnen Steuerbefehle von der App an das Programm, haben wir uns für eine Kommunikation über UDP entschieden. Das UDP Protokoll bietet sich in diesem Szenario an, da keine stetige Verbindung zwischen App und Pi notwendig ist, wodurch einige Probleme entfallen, und es akzeptabel ist falls einzelne Befehle verloren gehen. Da es sich auch beim Versenden von Daten über UDP um ein blockierenden Vorgang handelt, haben wir eine von AsyncTask abgeleitete Klasse UDPCommand geschrieben. In der doInBackground() Methode der Klasse wird zunächst der String Befehl in ein Byte Array gecastet und dieses Array, sowie ier Ip-Adresse des Zielcomputers und der gewünschten Port, in unserem Fall 4711, in ein DatagramPacket verpackt und letztlich über die DatagramSocket Klasse verschickt.

Das ausführende Java Programm des Raspberry Pi erstellt seinerseits ein UDP-Listener für den Port 4711. Da die Klasse ständig auf eintreffende UDP Pakete warten muss wird der Listener in einem unabhängigen Task ausgeführt. Zunächst wird ein DatagramSocket Objekt für den Port 4711 erstellt, in einer while(true) Schleife wird durchgehend auf eingehende DatagramPackets des DatagramSockets gewartet. Die gesendeten Daten, in unserem Fall einzelne Chars, werden aus dem DatagramPacket extrahiert und in ein char gecastet, letztlich werden die Chars für die Weiterverarbeitung in eine BlockingQueue eingereiht.

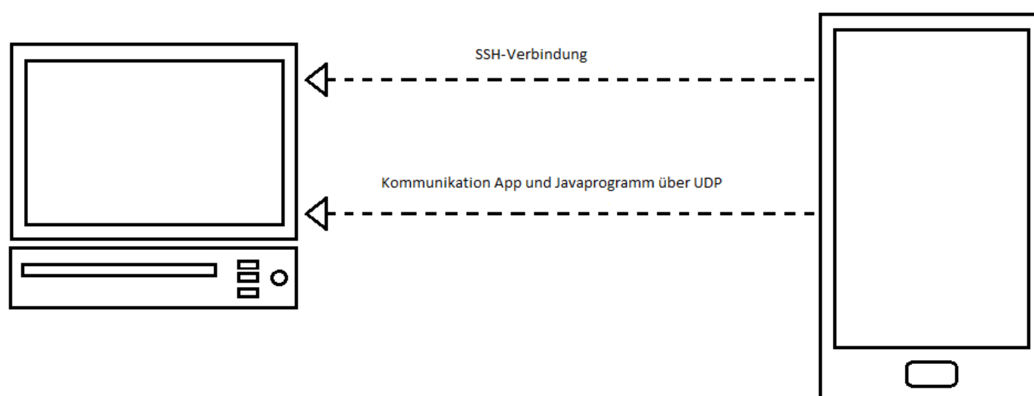


Abbildung 1 Skizze Verbindung zwischen App und Raspberry Pi

KOMMUNIKATION JAVA PROGRAMM UND RP6

Für die Kommunikation zwischen dem, auf dem Raspberry Pi laufenden, Java Programm und dem RP6 Roboter haben wir die bereits vorhandene RP6PiLib verwendet. Das Programm startet zunächst den UDP-Listener in einem extra Task, danach wird eine Methode aufgerufen, welche erst eine RP6RobotI2C Instanz erstellt, und danach in einer Endlosschleife auf eingehende Befehle der BlockingQueue wartet. Befindet sich in der BlockingQueue ein Befehl, wird dieser über ein Switch-Statement überprüft und je nach Befehl die entsprechende Methode der RP6RobotI2C Instanz ausgeführt. Die App kann sechs verschiedene Befehle senden: Geradeaus, Zurück, Links, Rechts, Stopp und einen Befehl zum Beenden. Die moveAtSpeedRight bzw. moveAtSpeedLeft Methoden der RP6RobotI2C Instanz werden mit drei verschiedenen Variablen aufgerufen, einer Geschwindigkeit, einem Teiler und der Richtung. Die Geschwindigkeit definiert wie schnell sich der Roboter vorwärtsbewegen soll, der Teiler bestimmt im welchen Verhältnis sich die Kette, welche sich auf der Seite des Roboters, in die er sich rotieren soll, befindet, langsamer als die andere Kette bewegt. Bei einer Geschwindigkeit von 100 und einem Teiler von 4 bewegt sich die Kette in Rotationsrichtung also nur 25% der Geschwindigkeit, während sich die andere Kette mit 100% der Geschwindigkeit bewegt, dadurch kommt es zu einer Kurve in die Richtung der langsameren Kette. Da unser Roboter rückwärtsfährt ist ebenfalls zu beachten, dass die Richtungsvariable der beiden Methoden auf 1 gesetzt sein muss, sodass der Roboter die Kurven nicht in die falsche Richtung ausführt, ist die Richtungsvariable auf 0 gesetzt fährt der Roboter standardmäßig vorwärts. Wird von der App aus der Befehl zum Beenden gesendet, wird dieser nicht erst in die BlockingQueue eingereiht und dann abgearbeitet, sondern das Programm direkt beendet, dadurch kann bei Fehlern, beispielsweise dem Fehlerhaften mehrfachsenden von Befehlen, das Programm unabhängig von der BlockingQueue beendet werden.

AUFBAU DER APPLIKATION

Die App besteht aus vier verschiedenen Seiten, welche hierarchisch angeordnet sind. Neben den Hauptseiten besitzt die Applikation noch eine Klasse, welche die Anmeldedaten des Netzwerks in statischen Variablen speichert, wodurch man von jeder Seite aus auf diese Daten zugreifen kann. Zu den vier Seiten gehören die Startseite, eine Auswahlseite, eine Steuerseite, sowie eine Algorithmenseite, welche dem Benutzer eine Interaktion mit dem RP6 Roboter ermöglicht, während dieser die Labyrinth Algorithmen ausführt. Die Benutzung der App lässt sich in drei Phasen einteilen, in der ersten Phase trägt der Benutzer die erforderlichen Netzwerkdaten in die App ein, wo sie so lange gespeichert werden, bis die App beendet wird, diese Aufgabe übernimmt die Startseite. In der zweiten Phase wählt der Benutzer das gewünschte Programm aus, welches auf dem Raspberry Pi ausgeführt werden soll, hierfür ist die Auswahlseite zuständig. In der letzten Phase, der Interaktionsphase, kann der Benutzer, je nachdem welches Programm er auf dem Pi laufen lässt, entweder den RP6 Roboter direkt, über die Steuerseite, selbst steuern, oder er erhält über die Algorithmenseite Feedback von den laufenden Programmen und die Möglichkeit diese zu Beenden.

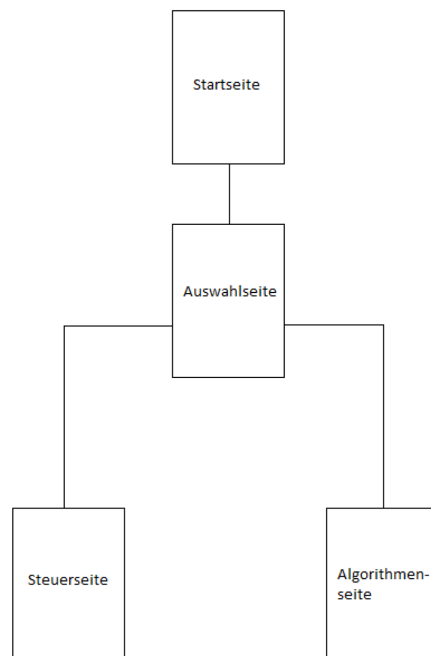


Abbildung 2 Hierarchie der Seiten

STARTSEITE

Die Startseite besteht aus zwei PlainText Elementen, einem Password Element und einem Button. Die beiden PlainText Elemente sind für die Eingabe der Netzwerk Ip-Adresse, sowie den Benutzernamen, welche für den Aufbau der SSH-Verbindung benötigt werden, das Passwort wird in das Password Element eingegeben, der Unterschied zwischen den PlainText Elementen und dem Password Element besteht darin, dass bei den PlainText Elementen der eingegebene Text als Klartext angezeigt wird, während beim Password Element der eingegebene Text als Punkte „verschlüsselt" wird. Der verwendete Button übernimmt zwei Aufgaben, zum einen speichert er die eingegebenen Daten in statische Variablen der Parameter Klasse, zum anderen navigiert er den Benutzer, über ein Intent, auf die nächste Seite der Applikation, welche der Auswahlseite entspricht, zudem beendet er Aktuelle Seite. Die Funktionalität wird des Buttons wird per Code über den onClickListener des Buttons implementiert. Außerdem wird noch die onBackPressed Methode der Seite überschrieben, sodass beim Drücken des Zurück Buttons des Smartphones die Seite erst beendet und letztlich die App mit mittels System.exit(0) geschlossen wird.

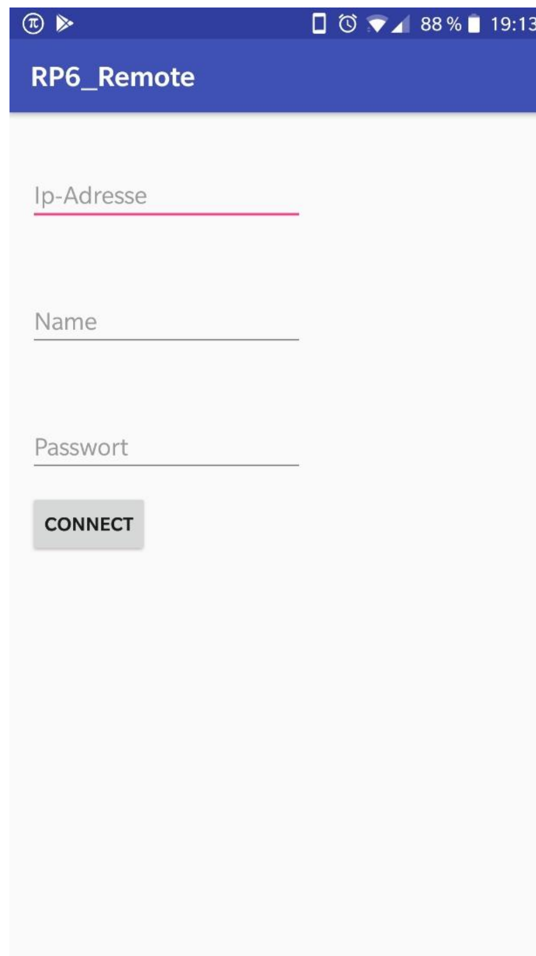


Abbildung 3 Startseite der Applikation

AUSWAHLSEITE

Die Auswahlseite besteht aus zwei TextViews, einer RadioButtonGroup mit drei RadioButton und einem normalen Button. Eine der beiden TextViews dient als Überschrift für die RadioButtonGroup, die andere, welche sich über dem Button befindet, wird als Statusanzeige verwendet, sie gibt dem Benutzer beispielsweise eine Rückmeldung, wenn beim Aufbau der SSH-Verbindung ein Fehler aufgetreten ist. Die RadioButtonGroup, sowie die enthaltenen RadioButtons, sind für die Auswahl des zu startenden Programms. Mithilfe eines Switch-Case-Statements wird überprüft welcher der RadioButton ausgewählt ist, und dem entsprechend der Befehl für den Aufbau der SSH-Verbindung gesetzt. Die Funktionalität des Buttons wird wieder über die setOnClickListener Methode implementiert, zunächst wird der Text der Status TextView geändert, sodass dieser dem Benutzer mitteilt, dass eine SSH-Verbindung aufgebaut wird, danach wird versucht die SSH-Verbindung aufzubauen, falls beim Verbindungsaufbau eine Ausnahme ausgelöst wird, wird in der Activity Klasse der Auswahlseite ein boolean Flag auf false gesetzt. Danach wird überprüft ob Flag auf true oder false gesetzt ist, falls es auf false gesetzt ist, wird der Text der Status TextView dementsprechend angepasst, ist das Flag auf true gesetzt, wird über ein Switch-Case-Statement geprüft welcher RadioButton ausgewählt ist und je nach ausgewähltem Programm entweder die Steuerseite oder die Algorithmenseite aufgerufen. Letztlich wird noch die Aktuelle Auswahlseite beendet. Ausserdem ist noch die onBackPressed Methode der Auswahlseite überschrieben, sodass der Knopf auf die Startseite zurück navigiert wird.

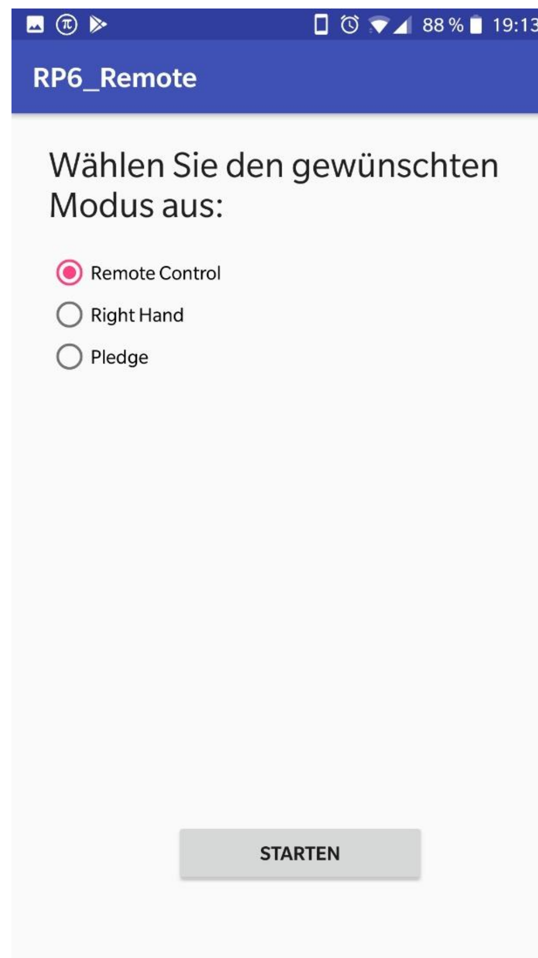


Abbildung 4 Auswahlseite

STEUERSEITE

Die Steuerseite enthält lediglich fünf ImageButton, sowie eine interne private Klasse, welche UDP Packete über ein DatagramSocket an den Port 4711 des Zielrechners, in diesem Fall den Raspberry Pi, versendet. Die Image Button bieten die Möglichkeit anstatt Text auch Icons anzuzeigen. Diese Icons werden dafür im drawable Ordner Innerhalb des resource Ordners gespeichert und bei den entsprechenden Buttons in der srcCompat gesetzt. Die OnClickListener der ImageButton erzeugen eine neue Instanz der privaten UWP Klasse, welche in einem neuen Task erstellt wird, und versenden je nach Button unterschiedliche Strings, entweder „up“, „down“, „stop“, „right“ oder „left“. Je nach String wird im Java Programm ein anderer Befehl ausgeführt. Der Backbutton des Smartphones sendet einen Abbruchbefehl über die UDP Verbindung an das Java Programm, navigiert zurück zur Auswahlseite und beendet letztlich die Steuerseite.

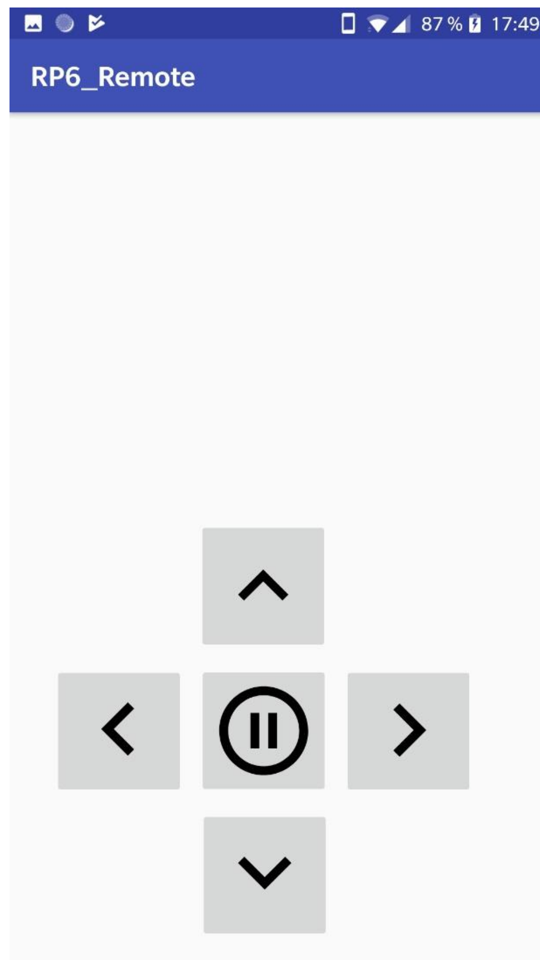


Abbildung 5 Steuerseite

ALGORITHMENSEITE

Die Algorithmenseite besteht aus eine TextView und einem Button, zudem besitzt sie noch eine BlockingQueue, in der, eingehende Daten, welche von einem Java Programm ausgesendet werden, gespeichert werden. Beim Aufrufen der Seite wird zunächst der onClickListener des Buttons initialisiert, wird der Listener ausgelöst, wird zunächst der String „exit“ über UDP an das Java Programm gesendet, und letztlich zurück zur Auswahlseite navigiert und die Algorithmenseite beendet. Nachdem der Button initialisiert wurde, wird eine neue Instanz einer privaten UDP Empfangsklasse, welche in einem extra Task läuft, erstellt. Die UDP Empfangsklasse wartet auf UDP Pakete, welche auf Port 4711 empfangen werden, und reiht diese in die BlockingQueue ein. Nachdem die Empfangsklasse gestartet wurde, springt die App in eine while(true) Schleife, in dieser Schleife werden Befehle aus der BlockingQueue entnommen und verarbeitet, falls in der BlockingQueue der entsprechende String eingereiht wurde, wird dem Benutzer über die TextView mitgeteilt, dass der Roboter einen Weg aus dem Labyrinth gefunden wurde, zudem wird ein boolean Flag auf false gesetzt. Der Backbutton des Smartphones überprüft zunächst ob das boolean Flag auf false gesetzt ist, ist dies der Fall wird auf die Auswahlseite zurück navigiert, ist das Flag auf true gesetzt wird beim drücken des Backbuttons kein Code ausgeführt.

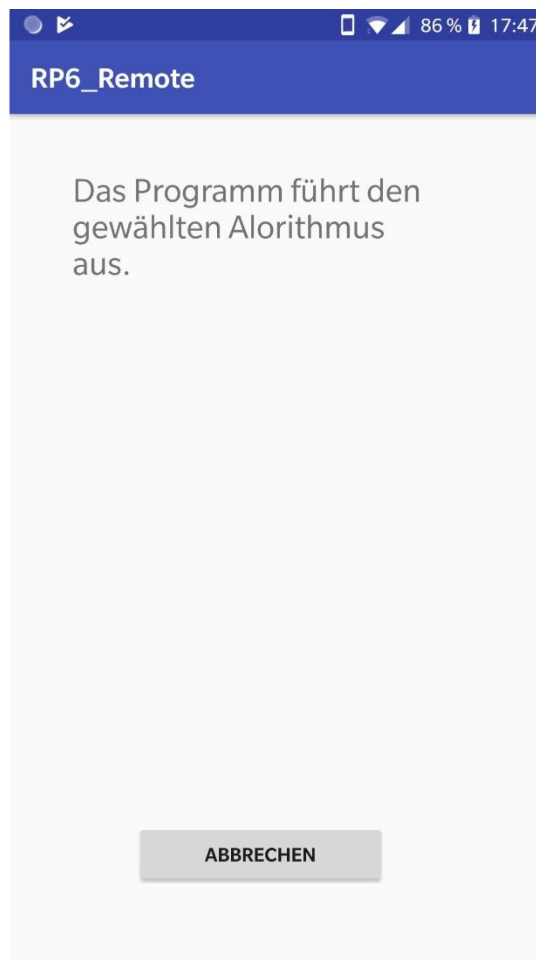


Abbildung 6 Algorithmenseite

FAZIT

Rückblickend lässt sich sagen, dass wir doch auf einige Probleme gestoßen sind, die wir leider nicht alle beheben konnten oder zukünftig bereits zu Beginn des Projekts einen anderen Ansatz verfolgen würden.

So führt beispielsweise der unterschiedliche Verschleiß, der im RP6 verbauten Antriebsmotoren leider in der praktischen Anwendung dazu, dass es dem Roboter schwer fällt die Spur zu halten oder verlässlich im angegebenen Winkel zu rotieren. Auch durch Anpassen der Übergabeparameter der Steuermethoden aus der RP6 Bibliothek ließ sich dieses Problem zwar verbessern, aber nicht gänzlich lösen.

Vollkommene Flexibilität bezüglich der Ausmaße des Labyrinths ist durch den Einsatz von nur einem drehbaren Sensor leider nicht erreichbar. Hierfür müssten mindestens seitliche, im besten Fall auch noch am Heck, Sensoren angebracht werden, die dauerhaft den Abstand zu den Wänden messen.

Trotz dieser Hürden war es uns am Ende möglich alle Ziele (inklusive der optionalen Visualisierung) zu erreichen. Alle Gruppenmitglieder waren sich einig darüber, dass es die richtige Entscheidung war dieses Projekt zu wählen, da hierbei eine ausgewogene Mischung an Zusammenbau der Hardware, Implementierung des nötigen Codes und Erstellung einer mobilen Applikation gegeben war.

QUELLENVERZEICHNIS

- http://www.jcraft.com/jsch/
- https://stackoverflow.com/questions/4194439/sending-commands-to-server-via-jsch-shell-channel
- https://de.wikipedia.org/wiki/L%C3%B6sungsalgorithmen_f%C3%BCr_Irrg%C3%A4rten
- https://tutorials-raspberrypi.de/entfernung-messen-mit-ultraschallsensor-hc-sr04/
- http://www.inf.fu-berlin.de/lehre/SS17/PSThInf/notes/06_pledge.pdf
- https://stackoverflow.com/questions/10097491/call-and-receive-output-from-python-script-in-java
- http://www.arexx.com/rp6/html/de/docs.htm
-