

Hochschule Bochum
Bochum University
of Applied Sciences
Campus
Velbert/Heiligenhaus



NAVIGATION EINES SPHERO MINI ROBOTERS IN ROS2

VERTIEFUNG ROBOTIK

DENNIS THIELE UND SIMON WEBER
018103365 UND 018104160

Hochschule: Hochschule Bochum Campus Velbert/Heiligenhaus

Fachbereich: Elektrotechnik

Studiengang: Mechatronik & IT

Bearbeitungszeitraum: Sommersemester 2023

Betreuer: M.Sc. Jan Weber

Inhaltsverzeichnis

1. Einleitung und Aufgabenstellung (Weber)	2
2. Grundlagen (Weber).....	3
2.1 Sphero Mini	3
2.2 ROS2	4
2.3 Bilderkennung	4
2.4 Vektorberechnung.....	5
2.5 Fahrplanungsalgorithmus	5
3. Software (Thiele)	7
3.1 Sphero Mini Treiber (Thiele)	7
3.2 Bilderkennung mit OpenCV (Thiele).....	9
3.3 Sphero Node (Thiele).....	10
3.3.1 Verbindung zum Sphero herstellen (Thiele).....	10
3.3.2 Sphero-Erkennung (Thiele).....	10
3.3.3 Fahrbefehle (Weber)	14
3.3.4 Initialfahrt (Weber).....	14
3.3.5 Berechnung von Winkel und Geschwindigkeit für Soll- und Ist-Position (Thiele).....	15
3.3.6 Ziel-Regelung (Thiele).....	16
3.3.7 Bild- und Weltkoordinatensystem (Thiele)	17
3.3.8 Winkelumrechnung (Thiele).....	18
3.4 Wavefrontplanner (Thiele).....	18
3.5 Kartenerstellung (Thiele).....	23
3.5.1 Kartenerstellung (Weber).....	23
3.5.2 Skalieren von Start- und Zielpunkt (Thiele).....	25
3.5.3 Zurückskalieren der Pfadkoordinaten (Thiele)	25
3.6 Main-Methode (Weber).....	26
4. Fazit und Ausblick (Weber).....	28
Anhang	29
Abbildungsverzeichnis.....	29
Literaturverzeichnis.....	29
Eidesstaatliche Erklärung Thiele.....	30
Eidesstaatliche Erklärung Weber.....	30

1. Einleitung und Aufgabenstellung (Weber)

Die ROS2 Community ist im stetigen Wachstum. Im Rahmen des Moduls Vertiefung Robotik gilt es einen eigenen Anteil an diesem Wachstum in Form von Entwicklungsprojekten beizutragen. Diese Projekte finden im Sommersemesters 2023 an der Hochschule Bochum am Campus Velbert/Heiligenhaus statt.

Der Inhalt dieser Ausarbeitung bezieht sich auf die Navigation und Ansteuerung eines Sphero Mini Kugelroboters. Die dafür relevanten Themen, die auch in dieser Ausarbeitung behandelt sind: Ist die Software für eine ROS2 Applikation, der Navigationsalgorithmus Wave-Front-Planner und die Bildverarbeitung über Open CV2.

Die Konkrete Aufgabenstellung für dieses Projekt ist die Inbetriebnahme des Sphero-Kugelroboters in ROS2. Inbetriebnahme heißt hierbei eine kontrollierte Ansteuerung des Spheros über Fahrbefehle innerhalb eines ROS2 Nodes.

Neben der Inbetriebnahme soll eine Kamera den Roboter in einer vorgegebenen Fläche lokalisieren. Innerhalb dieser Fläche soll der Sphero, farblich markieren Hindernissen ausweichen und einer berechneten Navigation folgen.

2. Grundlagen (Weber)

Die folgenden Abschnitte sollen einen Überblick über die Schnittstellen des Projekts geben.

Dabei behandeln sie die grundlegenden Konzepte, Prinzipien oder Techniken, die bei der Gestaltung und Implementierung der Schnittstellen zum Einsatz kommen.

2.1 Sphero Mini

Der Mini-Kugelroboter der Firma Sphero ist ein ferngesteuerter Roboter, der primär für die Verwendung als Spielzeug gedacht ist (siehe *Abbildung 1*). Neben dem spielerischen Betrieb soll der Sphero aber auch das Programmieren für Anfänger ermöglichen, indem sich einzelne Fahrbefehle mit Kontrollstrukturen aus der Informatik verbinden lassen. Hierfür verfügt der Roboter über zwei Smartphone-Apps: Sphero Play und Sphero Edu.

Der Roboter hat in etwa die Größe eines Tischtennisballs und verfügt über ein Gyroskop einen Beschleunigungssensor und ansteuerbaren LEDs. Der eigentliche Roboter befindet sich in einem kugelförmigen Gehäuse, indem sich der Roboter frei drehen kann. Mithilfe des Gyroskops kann der Sphero seine Lage im Raum messen und darauf basierend seine internen Motoren so gezielt ansteuern, dass er sich innerhalb des Gehäuses bewegt. Die tatsächliche Bewegung auf dem Untergrund kommt durch eine Schwerpunktverschiebung innerhalb des Gehäuses zustande.

Parameter wie Richtung oder Geschwindigkeit lassen sich über eine Bluetooth Verbindung übertragen und Sensoren wie die Internal Measurement Unit (IMU) lassen sich darüber auslesen. Mit der passenden Schnittstelle ist es möglich die Bluetooth-Befehle nicht über eine App, sondern über ein ROS2-Node zu versenden (siehe RoboLab ,2023).



Abbildung 1: Sphero Mini Roboter

[Quelle: <https://www.zavvi.de/geschenk-gadgets/sphero-mini-roboterball-weiss/11529468.html>]

2.2 ROS2

Das Robot Operating System 2 (ROS2) ist ein Framework zur Unterstützung bei der Programmierung von Roboteranwendungen. Es findet seinen Einsatz im Forschungsgebiet der Robotik und autonomer Systeme. ROS2 wird seit 2012 von der gemeinnützigen Organisation Open Source Robotics Foundation entwickelt (siehe Wurth, 2022).

ROS2 ist die fortschrittlichere Version von seinem Vorgänger ROS und bietet viele Verbesserungen wie zum Beispiel die Verwendung von Middleware zur Kommunikation mit DDS-Standard oder eine bessere Verteilung der Rechenleistung auf mehrere Recheneinheiten (siehe Mazzari, 2019).

Eine weitere wichtige Komponente von ROS2 ist die Möglichkeit, verschiedene Programmiersprachen zu nutzen, unter anderem C++ und Python. Dadurch können Entwickler die Sprache wählen, die am besten zu ihren Anforderungen, Kenntnissen und Vorlieben passt.

Insgesamt ist ROS2 ein umfassendes Framework, das Entwicklern eine leistungsfähige und flexible Plattform bietet, um Roboteranwendungen zu entwickeln.

2.3 Bilderkennung

Ein maßgeblicher Anteil des Projektes stellt die Lokalisierung des Sphero-Roboters über eine Kamera dar. Neben dem Sphero sollen auch das Spielfeld sowie die Hindernisse über die Kamera eingelesen werden. Das Bild und die darin enthaltenen Positionen von Sphero und Hindernissen ist notwendig, um damit im späteren Verlauf dieser Ausarbeitung eine Navigation zu berechnen.

Für das Auslesen der Kamera sowie das Vor- und Nachbearbeiten der Bilder bietet sich die Programm-bibliothek OpenCV an. Diese ist eine freie Software und lässt sich gut in Python-Code einbinden. Die Bibliothek bietet über 2500 bereits optimierte Algorithmen zur Bilderkennung und verfügt über eine große Community (siehe OpenCV, 2023).

Mit OpenCV können Entwickler grundlegende Operationen auf Bildern ausführen, wie zum Beispiel das Lesen und Speichern von Bildern, das Anwenden von Filtern, das Ändern von Helligkeit und Kontrast, das Skalieren und Zuschneiden von Bildern sowie das Zeichnen von geometrischen Formen. Darüber hinaus bietet die Bibliothek erweiterte Funktionen wie die Erkennung und Verfolgung von Objekten, die Merkmalsextraktion, die Stereo-Vision, die Gesichtserkennung und die Bildklassifizierung (siehe OpenCV, 2023).

Eine dieser Algorithmen ist die Blob-Erkennung. Diese sorgt dafür, eine Gruppe von Pixeln mit einstellbaren Eigenschaften, als ein Blob-Objekt zu erkennen. Je nach Parameterwahl ist es somit möglich die genaue Position von Objekten in einem Bild auszulesen.

Dieser Algorithmus eignet sich gut für die Lokalisierung des Sphero-Roboters, da sich dessen Erscheinungsbild als weiße Kugel als Blob definieren lassen.

2.4 Vektorberechnung

Für die Berechnung der Navigation des Sphero-Roboters, ist es erforderlich verschiedene Vektoren und Winkeln zu berechnen. Diese Berechnungen sind nötig, um die Ausrichtung des Roboters relativ zum Ursprung zu ermitteln.

Hierfür kommt die Funktion `arctan2` zum Einsatz. Diese Funktion berücksichtigt im Gegensatz zum einfachen Arcus Tangens den X- und Y-Wert der Vektoren und ist somit besser geeignet um den vollständigen Winkelbereich von -180° bis $+180^\circ$ abzudecken. Das erreicht die Funktion, indem sie die Vorzeichen der Vektoren berücksichtigt. Außerdem vermeidet die Funktion ein Teilen durch 0.

Ein Beispiel für die Anwendung von `atan2` wäre die Berechnung des Winkels, den ein Roboter auf einer Ebene relativ zu einer bestimmten Ausrichtung hat. Mit Hilfe einer Differenz zwischen den aktuellen Koordinaten des Roboters und den Zielkoordinaten, kann die `atan2`-Funktion den Winkel bestimmen, um den sich der Roboter drehen muss, damit er sich zum Ziel ausrichtet. Im Code sieht ein entsprechender Aufruf folgendermaßen aus.

```
start_ref = ref-startpunkt
phi = np.arctan2(start_ref[1],start_ref[0])
```

start_ref ist die Differenz der Startposition und der Ziel- bzw. der Referenzposition. Die Variable *phi* speichert die Ausgabe der `arctan2`-Funktion Abhängigkeit zu den Differenzkoordinaten. Die Ausgabe der Funktion ist ein Winkel in Bogenmaß, der von der X-Achse ausgeht und positiv gegen den Uhrzeigersinn zählt.

2.5 Fahrtplanungsalgorithmus

Der Fahrtplanungsalgorithmus, der für die Navigation des Spheros zum Einsatz kommt, ist der Wavefront-Planner. Dieser findet den schnellsten Weg von einer Startposition zu einer Zielposition in einer vorgegebenen Weltkarte. Die Weltkarte wird für den Algorithmus auf ein Raster projiziert.

Zunächst initialisiert der Algorithmus alle Zellen der Karte mit bestimmten Startwerten. Positionen an denen Hindernisse sind, initialisiert er mit einer 1 und die Zielposition initialisiert er mit 2. Die Startposition spielt vorerst keine Rolle. Alle Zellen auf der Karte, die kein Hindernis enthalten oder nicht der Zielpunkt sind werden mit einer 0 gekennzeichnet.

Der Algorithmus schaut sich alle Nachbarmfelder der Zielposition an, die mit einer 0 gekennzeichnet sind und markiert diese mit einer 3.

An dieser Stelle gibt es zwei Varianten des Planers. Eine Variante berücksichtigt die direkten vier Nachbarzellen einer Zelle und die andere Variante berücksichtigt auch die diagonal anliegenden Zellen und berücksichtigt somit acht Nachbarmfelder. Das folgende Beispiel basiert auf der ersten Variante, mit vier Nachbarzellen.

Im nächsten Schritt markiert der Algorithmus alle 0-er Felder, die neben allen 3-er Feldern liegen mit einer 4.

Das wiederholt der Algorithmus danach mit den 4-er Zellen und danach mit den 5-er Zellen, bis keine 0-er Felder mehr übrig sind.

Egal auf welchem Punkt sich der Roboter jetzt befindet, er muss sich nur noch den absteigenden Zahlen folgen. Somit wird der Sphero innerhalb der Rasterwelt auf schnellstem Weg zum Ziel geleitet.

Ein Beispiel für einen Pfad von den Koordinaten [1, 1] zu [6, 1] ist in Abbildung 3 dargestellt und hervorgehoben.

Die Berechnung des Pfades basiert auf der Karte, die in Abbildung 2 dargestellt ist. Hierbei hat der Algorithmus bereits das Ziel als auch die Hindernisse mit den entsprechenden Startwerten initialisiert.

	0	1	2	3	4	5	6	7
0	0	0	0	1	1	0	0	0
1	0	0	0	1	1	0	2	0
2	0	0	0	1	1	0	0	0
3	0	0	0	1	1	0	0	0
4	0	0	0	1	1	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0

Abbildung 2: Karte bevor Wavefront

	0	1	2	3	4	5	6	7
0	17	16	15	1	1	4	3	4
1	16	15	14	1	1	3	2	3
2	15	14	13	1	1	4	3	4
3	14	13	12	1	1	5	4	5
4	13	12	11	1	1	6	5	6
5	12	11	10	9	8	7	6	7
6	13	12	11	10	9	8	7	8
7	14	13	12	11	10	9	8	9
8	15	14	13	12	11	10	9	10
9	16	15	14	13	12	11	10	11

Abbildung 3: Karte nach Wavefront

3. Software (Thiele)

Das folgende Kapitel beschäftigt sich mit der Software, die notwendig ist, um den Sphero wie in der Aufgabenstellung gefordert anzusteuern. Dabei gibt es für jeden Teil der Aufgabenstellung bzw. für jeden Abschnitt des Programms ein eigenes Kapitel, welches die Software dazu erläutert. Die wesentlichen Aspekte sind hierbei:

- Bilderkennung
- ROS2 Node für den Sphero Mini erstellen
- Fahrplanungsalgorithmus
- Kartenerstellung
- Programmablauf

Zunächst wird das Erstellen des ROS2 Nodes erläutert. Im weiteren Verlauf wird jeweils zunächst der wesentliche Code erklärt und dieser im Anschluss abgebildet. Das vollständige Projekt ist unter GitLab zu finden (<https://gitlab.cvh-server.de/dthiele/sphero-mini-ros-2>)

3.1 Sphero Mini Treiber (Thiele)

Zum aktuellen Stand der Arbeit, gab es kein fertiges ROS2 Paket, mit dem der Sphero Mini angesteuert werden kann. Allerdings wurde nach einiger Recherche ein ROS1 Paket auf GitHub für den Sphero Mini gefunden (siehe Goubard, 2023). Dieses ist von *cedricgoubard* geschrieben und basiert auf einer ROS Noetic Version.

Um nun ein neues ROS2 Paket zu entwickeln, werden im Wesentlichen die drei untenstehenden Dateien benötigt:

- `core.py`
- `_constants.py`
- `sphero_conf.py`

Diese werden im Folgenden angepasst, um ein ROS2 Node zu erstellen. Die Basis für das neue Sphero Mini ROS2 Paket ist hierbei ein einfacher Python-Node. Dazu wird zunächst ein Workspace Ordner `ros2_ws` erstellt und das Python-Paket `sphero_mini_controller` erstellt sowie eine `sphero_mini.py` Datei im `src` Ordner für den Quellcode angelegt, die später den Node enthalten soll. Die Datei `_constants.py` kann so in das neue Paket übernommen werden. In der `sphero_conf.py` Datei ist lediglich die MAC-Adresse des Spheros hinterlegt. Jedoch müssen in der `core.py` Datei einige Code-Zeilen umgeschrieben werden. Unter ROS gab es für Python das Paket `rospy`, dieses trägt allerdings unter ROS2 den Namen `rclpy` und hat unterschiedliche Funktionalitäten als unter ROS. Daher wird auf den Import von `rospy` verzichtet. Dieser diente in der ROS `core.py` Datei dazu `rospy.logdebug()` aufzurufen. In der neuen `coreROS2.py` Datei können diese Aufrufe durch ein einfaches `print()` ersetzt werden. Damit sind alle wesentlichen Änderungen abgeschlossen und die drei Dateien können in den `src` Ordner übernommen werden.

In der *sphero_mini.py* Datei kann nun der Sphero Node erstellt werden. Im Folgenden werden die Funktionen sowie die Struktur des Sphero Nodes erklärt. *Abbildung 4* zeigt die einzelnen Abschnitte und Klassen der *sphero_mini.py* Datei. Der Anfang des Programms beinhaltet zwei Abschnitte zum einen für die Imports und zum anderen für die globalen Variablen sowie die Konfiguration des Blob Detectors. Im Anschluss folgen vier Klassendefinitionen für die jeweiligen Funktionalitäten aus der Aufgabenstellung z. B. für die Bilderkennung, den Sphero Node, den Wavefrontplanner sowie die Map erstellung und die Main Methode.

Die Ordner Struktur des Projekts zeigt *Abbildung 5*. Hierbei sind im Quellcode Ordner *sphero_mini_controller* im Wesentlichen drei *.py* Dateien (blau markiert) wichtig, die während der Entwicklung zu Debug-Zwecken verwendet wurden sowie die *sphero_mini.py* Datei, die den Node enthält.

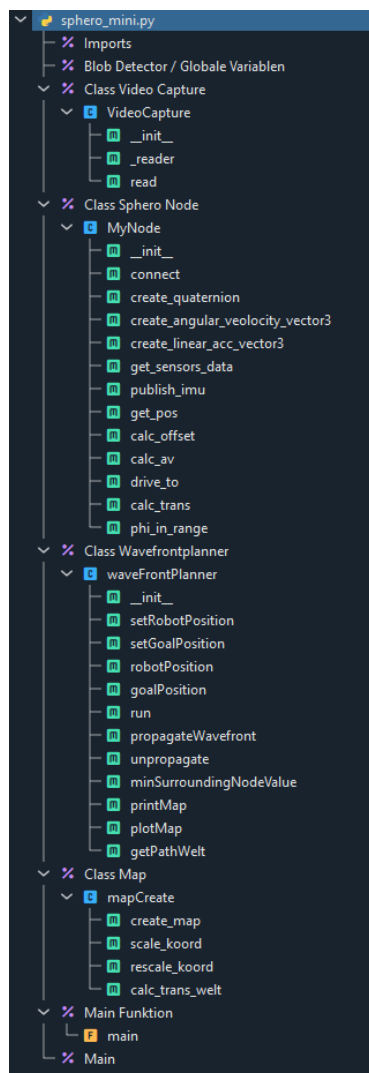


Abbildung 4 sphero_mini.py Code-Struktur

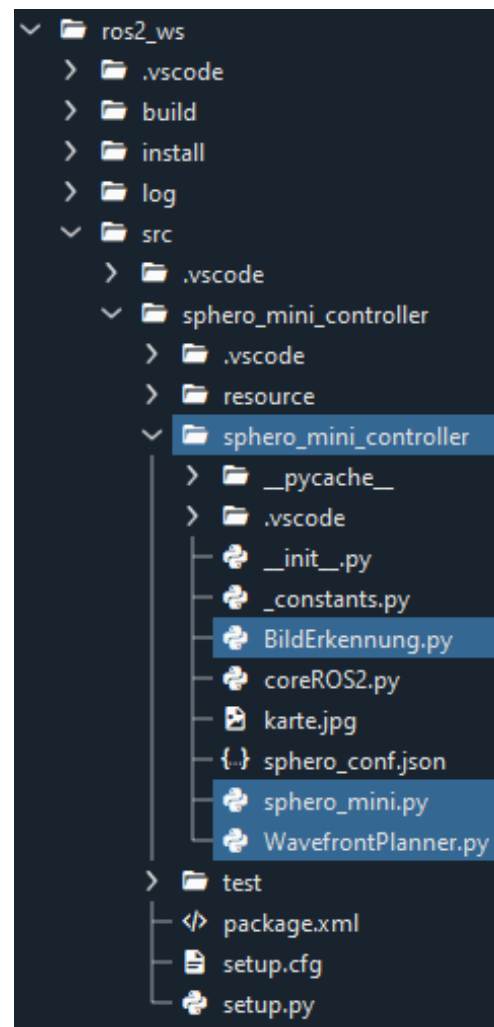


Abbildung 5 Ordner Struktur

Um den eigentlichen ROS2 Node zu erstellen, muss vom Paket *rcipy* die Klasse *node* importiert werden und der Konstruktor muss von der Node-Klasse erben. Danach kann in der Main-Methode das Node Objekt erstellt werden. Der weitere Programmcode wird in Kapitel 3.6 beschrieben.

3.2 Bilderkennung mit OpenCV (Thiele)

Die Bilderkennung in diesem Projekt erfolgt durch das Python Paket OpenCV. Dieses wird am Anfang der *sphero_mini.py* Datei importiert. Mithilfe von OpenCV kann ein Kamerabild eingelesen und verarbeitet werden. Dazu muss zunächst ein Capture Objekt erzeugt und anschließend ausgelesen werden. Im Folgenden ist ein Beispiel dafür abgebildet. Bei dem VideoCapture Objekt kann entweder ein USB-Port oder eine IP-Adresse angegeben werden. Durch den Aufruf von *cap.read()* wird das Bild im *img*-Format zurückgegeben und kann im weiteren Verlauf verarbeitet werden.

```
import cv2
cap = cv2.VideoCapture("http://root:root@10.128.41.239:80/mjpg/video.mjpg")
success, img = cap.read()
```

Das Problem dabei ist allerdings, dass die Bilder zwischengespeichert werden. Sobald ein VideoCapture Objekt erzeugt wird, erhält der User immer das erste Bild, was im Speicher liegt, sprich dem ältesten Bild, und nicht das aktuelle. Dies ist allerdings problematisch in Bezug auf die Bilderkennung des Spheros, da hier immer der aktuelle Status des Spheros erkannt werden muss. Um dieses Problem zu beheben, wurde eine VideoCapture Klasse von Stack Overflow verwendet, die immer das aktuelle Bild zurückliefert (siehe Overflow, 2023). Hierbei verwendet die Klasse eine Queue in der kontinuierlich die Bilder gespeichert werden und bei Bedarf das letzte Element der Queue ausgegeben wird. Dabei wird das aktuelle Bild mit *cap.read()* zurückgegeben. Die Beschreibung der Bilderkennung erfolgt im Kapitel 3.3.

```
#%% Class Video Capture
# Bufferless VideoCapture
# https://stackoverflow.com/questions/43665208/how-to-get-the-latest-frame-
from-capture-device-camera-in-opencv
class VideoCapture:
    def __init__(self, name):
        self.cap = cv2.VideoCapture(name)
        self.q = queue.Queue()
        t = threading.Thread(target=self._reader)
        t.daemon = True
        t.start()
    # read frames as soon as they are available, keeping only most recent one
    def _reader(self):
        while True:
            ret, frame = self.cap.read()
            #cv2.imshow("Cap", frame)

            if not ret:
                break
            if not self.q.empty():
                try:
                    self.q.get_nowait() # discard previous (unprocessed) frame
                except queue.Empty:
                    pass
            self.q.put(frame)
    def read(self):
        return self.q.get()
```

3.3 Sphero Node (Thiele)

Das Folgende Kapitel beschreibt die Klasse für den Sphero Node. Hierbei gibt es verschiedene Methoden der Klasse und die wesentlichen Methoden sind in eigenen Unterkapiteln aufgeführt. Die Kapitel sind chronologisch an den Programmcode orientiert (siehe *Abbildung 4*). Die Ausnahme stellt hierbei Kapitel 3.3.3 dar. Dieses gibt einen allgemeinen Überblick über die Fahrbefehle des Spheros. Zunächst erfolgt die Beschreibung des Verbindungsaufbaus mit dem Sphero.

3.3.1 Verbindungsaufbau (Thiele)

Der erste Schritt in der Erstellung des Sphero Nodes ist das Verbinden mit dem Sphero selbst. Hierbei wird zunächst die MAC-Adresse definiert und ein Sphero Objekt erstellt. Dieses Sphero-Objekt wird von der *SpheroMini* Klasse aus der Datei *coreROS2.py* erstellt. Im Anschluss wird das Objekt zurückgegeben. Mit diesem Sphero-Objekt werden alle wesentlichen Funktionen aufgerufen, die die Bedienung des Spheros betreffen wie z. B. die Fahrbefehle oder die Hintergrundbeleuchtung.

```
def connect(self):
    MAC_ADDRESS = "C6:69:72:CD:BC:6D"

    # Connect:
    sphero = SpheroMini(MAC_ADDRESS, verbosity = 4)
    # battery voltage
    sphero.getBatteryVoltage()
    print(f"Battery voltage: {sphero.v_batt}v")

    # firmware version number
    sphero.returnMainApplicationVersion()
    print(f"Firmware version:
          {'.'.join(str(x) for x in sphero.firmware_version)}")
    return sphero
```

3.3.2 Sphero-Erkennung (Thiele)

Für die Identifikation des Sphero Mini Roboters innerhalb des Kamerabildes kommt die Blob Erkennung von OpenCV zum Einsatz. Dies ist ein Verfahren zur Identifizierung von zusammenhängenden Bereichen oder Objekten in einem Bild. Ein Blob bezieht sich hierbei auf eine Region mit ähnlichen Eigenschaften wie Farbe, Helligkeit oder Textur, die von der umliegenden Umgebung abweicht (siehe Learnopencv, 2023).

Der Folgende Codeabschnitt dient dazu, die Bilderkennung zu testen und die Parameter der Blob-Erkennung anzupassen. Dieses ist im Folgenden abgebildet. Zunächst werden die Pakete eingebunden und die verschiedenen Parameter definiert. Die wichtigsten Parameter sind hierbei die Farbe, die Größe und die Rundheit. Einige Parameter sind hierbei auskommentiert, da diese von den aktuellen Umgebungsbedingungen abhängen und nicht immer benötigt werden wie z. B. von Lichtverhältnissen oder Abstand der Kamera zum Spielfeld.

```
import cv2
import numpy as np;

# Img Objekt
cap = cv2.VideoCapture("http://root:root@10.128.41.239:80/mjpg/video.mjpg")
#cap = cv2.VideoCapture(1)

# Setup SimpleBlobDetector parameters
params = cv2.SimpleBlobDetector_Params()

# Change thresholds
#params.minThreshold = 5
#params.maxThreshold = 500

# Filter by Color
params.filterByColor = True
params.blobColor = 255

# Filter by Area.
params.filterByArea = True
params.minArea = 20
params.maxArea = 200

# Filter by Circularity
#params.filterByCircularity = True
#params.minCircularity = 0.5
#params.maxCircularity = 1

# Filter by Convexity
#params.filterByConvexity = True
#params.minConvexity = 0.7
#params.maxConvexity = 1

# Filter by Inertia
params.filterByInertia = True
params.minInertiaRatio = 0.5

# Create a detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)
```

Nach dem Definieren der Parameter kann der *detector* erzeugt werden. Innerhalb einer while-Schleife werden nun wiederholt die Kamerabilder eingelesen und mithilfe von `cv2.cvtColor()` in ein Grayscale-Bild umgewandelt. Danach wird der *detector* auf dem Bild angewendet und die sogenannten Keypoints ausgelesen, also die Koordinaten der erkannten Blobs. Im Anschluss können dann die Keypoints in das Bild eingezeichnet und das Bild angezeigt werden. Durch Betätigen der Taste q wird das Programm beendet. *Abbildung 6* zeigt ein Beispielbild der Blob-Erkennung.

```
while True:
    success, img = cap.read()
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY )

    # Detect blobs.
    keypoints = detector.detect(gray)

    for keyPoint in keypoints:
        x = keyPoint.pt[0]
        y = keyPoint.pt[1]

    im_with_keypoints = cv2.drawKeypoints(gray, keypoints, np.array([]),
(0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

    # Show blobs
    cv2.imshow("Keypoints", im_with_keypoints)

    if cv2.waitKey(10) & 0xFF == ord('q'):
        break
```

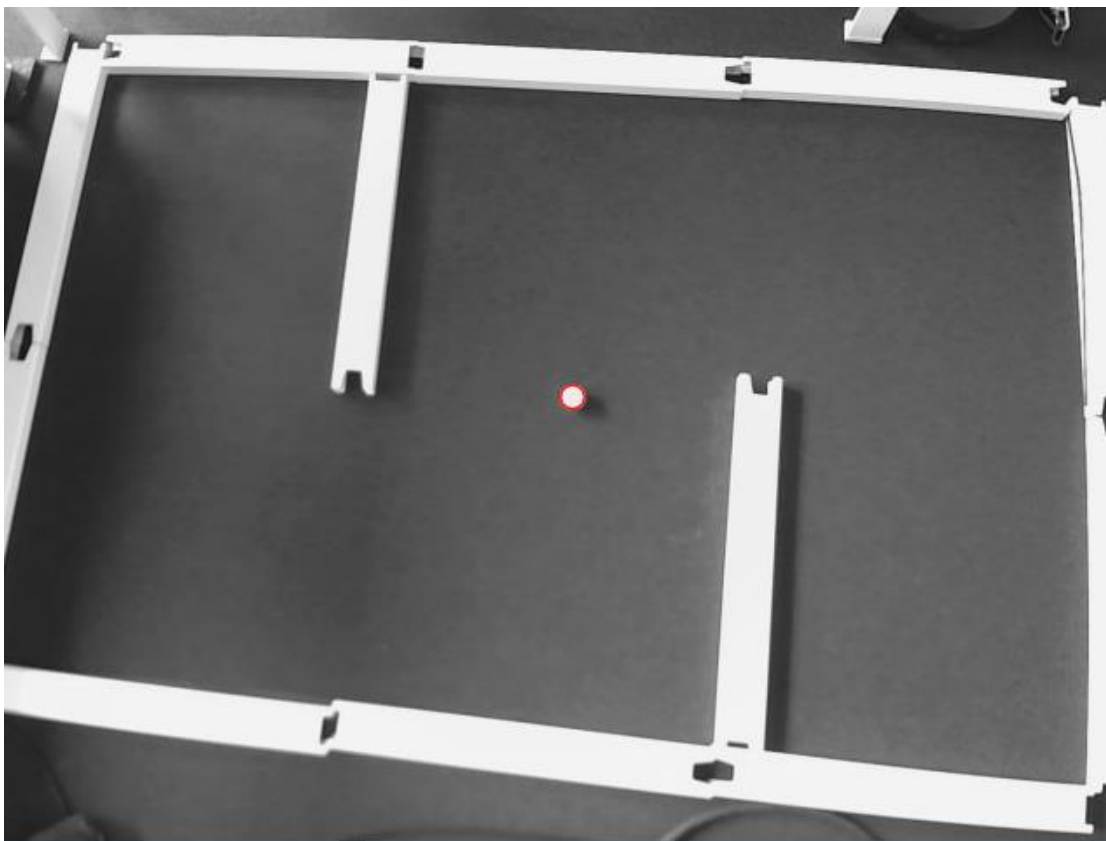


Abbildung 6 Blob Erkennung

In der *sphero_mini.py* Datei wurde der Test-Code abgewandelt und in die Funktion `get_pos()` integriert. Diese liefert immer die aktuelle Sphero Position zurück und benötigt das Capture Objekt *cap* sowie die Höhe des Bildes *height*. Die auskommentierten Zeilen dienten während der Entwicklung zu Debug-Zwecken. Die Variable *success* ist notwendig, falls der Sphero einmal für einen kurzen Zeitpunkt nicht im Bild erkannt wird, das das Programm nicht durch eine Fehlermeldung abstürzt, sondern durch die try und catch Anweisung den Fehler abfängt und das nächste Bild erneut in der while-Schleife ausliest, solange bis der Sphero wieder erkannt wird. Wichtig in der *get_pos()* Methode ist noch, dass hierbei die Sphero Bildkoordinaten in Weltkoordinaten durch den Aufruf der Methode *calc_trans()* umgewandelt werden. Auf die Problematik in Bezug auf Bild- und Weltkoordinaten geht Kapitel 3.3.4 bzw. Kapitel 3.3.7 genauer ein.

```
def get_pos(self, cap, height):
    #zeitanfang = time.time()

    success = False

    while(success == False):
        try:
            img = cap.read()

            gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY )
            keypoints = detector.detect(gray)

            for keyPoint in keypoints:
                x = keyPoint.pt[0]
                y = keyPoint.pt[1]
                success = True

            xTrans, yTrans = self.calc_trans(x,y, height)
        except Exception:
            continue

    #zeitende = time.time()
    #print("Dauer Programmausführung:",(zeitende-zeitanfang))
    #print("Get_Pos: X,Y:", xTrans,yTrans)

    return xTrans,yTrans
```

3.3.3 Fahrbefehle (Weber)

Um den Sphero innerhalb des ROS2-Treibers anzusteuern, benötigt das Programm folgenden Fahrbefehl:

```
sphero.roll(v,a)
```

Die Fahrgeschwindigkeit lässt sich mit in Form eines Integers von 0-255 einstellen. Hierbei ist jedoch zu betonen, dass sich der Sphero je nach Untergrund erst ab einem bestimmten Wert in Bewegung setzen kann, da er zunächst ein gewisses Losbrechmoment überwinden muss. Erfahrungsgemäß liegt dieser Wert bei einer Gummiunterlage bei 80 und bei Teppichboden bei 90. Die Übergabe des Wertes 0 bewirkt einen Stopp des Roboters.

Der zweite Übergabeparameter in der Funktion ist das α . Dabei handelt es sich um den Winkel, den der Roboter während des Fahrauftrages einnehmen soll. Dieser Winkel bezieht sich immer auf den Nullwinkel des Spheros. Der Nullwinkel ist der Winkel den der Sphero anfahren würde, wenn er für α eine 0 erhält. Dieser ist jedoch abhängig davon, wie der Roboter ausgerichtet ist. Die Ausrichtung kann sich mit jeder Neupositionierung des Roboters ändern und ist nicht von außen ersichtlich. Um den Nullwinkel nun auf ein umliegendes Koordinatensystem abzubilden, benötigt der Roboter zunächst eine Initialfahrt, die im folgenden Abschnitt behandelt ist.

3.3.4 Initialfahrt (Weber)

Die Funktion *calc_offset* führt die Initialfahrt des Roboters aus. Die Initialfahrt ist ein wichtiger Bestandteil, um den Roboter sinnvoll innerhalb eines Koordinatensystems navigieren zu lassen. Ziel der Initialfahrt ist es einen Offsetwinkel zu berechnen, der zwischen dem Nullwinkel des Roboters und einem charakteristischen Vektor im Koordinatensystem liegt. In diesem Fall handelt es sich bei dem charakteristischen Vektor um die X-Achse des Weltkoordinatensystems (siehe Kapitel 3.3.7).

Der Offsetwinkel bleibt für jede Betriebsperiode des Roboters konstant. Im eingeschalteten Zustand kann jede Änderung der Orientierung des Spheros über die IMU (*Internal Measurement Unit*) detektieren und so gut es geht ausgleichen werden. Wenn der Roboter jedoch ausgeschaltet ist oder dieser von Hand gedreht wird, bemerkt die IMU die Änderung nicht. Daher muss der Roboter jedes Mal, wenn dieser neu startet, die Initialfahrt durchführen und einen neuen Offsetwinkel berechnen.

Um den Offsetwinkel zu berechnen, erhält der Roboter einen Fahrbefehl in Richtung 0 mit einer vorgegebenen Geschwindigkeit. In diese Richtung fährt der Roboter eine Sekunde und hält an. Das Programm speichert die Position, an der er sich jetzt befindet.

Daraufhin erhält der Sphero Mini einen weiteren Fahrbefehl mit derselben Geschwindigkeit aber in die entgegengesetzte Richtung ($\alpha = 180^\circ$). Diesen Fahrbefehl führt der Roboter auch eine Sekunde lang aus und stoppt anschließend. Der Sphero befindet sich jetzt ungefähr wieder an dem Punkt, an dem er gestartet ist.

Mithilfe der aktuellen Position und der zuvor gespeicherten Position, lässt sich ein Richtungsvektor konstruieren.

Der Offsetwinkel berechnet sich aus dem `atan2` zwischen dem konstruierten Richtungsvektor und der X-Achse. Für den Vektor der X-Achse benötigt man lediglich die Punkte `[0, 0]` und `[0, n]`

mit $x \in \mathbb{R} > 0$

Da die Ausgabe des `atan2` in Radiant erfolgt, kann das Programm den Winkel zunächst mit

```
phi = np.degrees(phi)
```

in Grad umrechnen. Anschließend konvertiert das Programm den Winkel in einen sinnvollen Wertebereich von $0^\circ - 360^\circ$, indem es dem Winkel so lange um 360° reduziert bzw. erhöht, bis er in diesem Bereich liegt. Das ist notwendig, da der Übergabeparameter α auch nur in diesem Wertebereich arbeiten kann.

3.3.5 Berechnung von Winkel und Geschwindigkeit für Soll- und Ist-Position (Thiele)

Dieses Kapitel beschreibt die Berechnung des notwendigen Winkels und der Geschwindigkeit, basierend auf der aktuellen Start-Position und der Ziel-Position bzw. Soll-Position.

Ähnlich wie bereits in der Methode `calc_offset()` beschrieben, wird in der Methode `calc_av()` ein Vektor von Start- zu Zielpunkt aufgespannt und im Anschluss mithilfe der Methode `np.arctan2()` der Winkel von diesem Vektor zum Koordinaten Ursprung berechnet. Der Winkel wird dann in Grad umgerechnet und mit der Methode `phi_in_range()` überprüft, ob dieser auch im gültigen Bereich für den Sphero liegt (siehe Kapitel 3.3.8). Der Winkel entspricht dann dem Winkel, unter welchem der Sphero den Zielpunkt vom aktuellen Startpunkt aus anfahren muss. Dieser Winkel muss allerdings noch mit dem Offset-Winkel aus dem vorherigen Kapitel verrechnet werden. Die Geschwindigkeit wurde hier erstmal auf $v = 100$ festgelegt, da bei den Testszenarien kleinere bzw. größere Geschwindigkeiten, aufgrund des Bodenbeschaffenheit, nicht zum gewünschten Erfolg geführt haben. Auf die Problematik mit der Geschwindigkeit wird im Fazit bzw. Ausblick genauer eingegangen.

```
def calc_av(self, startPos, sollPos):
    startPos = np.array(startPos)
    sollPos = np.array(sollPos)

    pktSpheroKord = sollPos - startPos

    phi = np.arctan2(pktSpheroKord[1], pktSpheroKord[0])
    phi = np.degrees(phi)

    phiZiel = int(phi)

    phiZiel = self.phi_in_range(phiZiel)

    v = 100

    #print("Calc_AV: a,v", phiZiel, v)

    return phiZiel, v
```


3.3.6 Ziel-Regelung (Thiele)

Die Methode *drive_to()* dient dazu, einen Punkt anzufahren bzw. im weiteren Verlauf den Pfad des Wavefrontplanners abzufahren. Die Methode benötigt das Sphero Objekt, die Ziel Position, den Offset Winkel, das Capture Objekt sowie die Höhe des Bildes.

Der Parameter *dmin* gibt den Abstand an, der Minimal erreicht werden muss, sodass der Sphero anhalten und das Ziel als erreicht angesehen werden kann. Der erste Schritt besteht darin die aktuelle Position des Spheros auszulesen und in *d* den Abstand vom Sphero zum Zielpunkt zu berechnen. Die Liste *ar* dient dazu die Fahrbefehle des Spheros zu speichern, um diese im späteren Verlauf zu vergleichen. Der erste Fahrbefehl ist der Winkel 0.

```
def drive_to(self, sphero, targetPos, aOffset, cap, height):
    dmin = 20
    pos = self.get_pos(cap, height)
    pos = np.array(pos)

    diff = targetPos - pos

    d = np.linalg.norm(diff, ord = 2)

    ar = []
    ar.append(0)

    i = 1
```

Innerhalb einer while-Schleife werden nun so lange neue Fahrbefehle berechnet, bis der Abstand *d* kleiner ist als der minimale Abstand *dmin*. Zunächst werden der neue Winkel *a* und die Geschwindigkeit *v* berechnet, der Offset-Winkel *aOffset* abgezogen und in der Liste *ar* gespeichert. In einer If-Abfrage wird nun überprüft, ob sich der neue Winkel vom alten Winkel unterscheidet und dementsprechend entweder ein neuer Fahrbefehl mit *sphero.roll* gesendet oder mit dem letzten Fahrbefehl für weitere 0.05 s gefahren. Die Parameter *dmin* und das Delay in *sphero.wait* wurden empirisch ermittelt.

```
while d > dmin:
    a,v = self.calc_av(pos,targetPos, cap, height)

    aR = -aOffset - a #Fallunterscheidung?
    ar.append((self.phi_in_range(aR)))

    #Fahrbefehl
    if(ar[i] != ar[i-1]):
        sphero.roll(v, ar[i])
        sphero.wait(0.05)
    else:
        sphero.wait(0.05)
```

Im Anschluss wird wieder die aktuelle Position ausgelesen und die neue Abweichung d berechnet sowie die Zählvariable i um eins erhöht. Falls das Ziel erreicht wurde, wird dies auf der Konsole ausgegeben und der Sphero gestoppt.

```
#Aktuelle Pos
pos = self.get_pos(cap, height)
pos = np.array(pos)

#Abweichung
diff = targetPos - pos
diff = np.array(diff)
d = np.linalg.norm(diff, ord = 2)

i = i + 1

sphero.roll(0,0)
print("Ziel Erreicht")

return
```

3.3.7 Bild- und Weltkoordinatensystem (Thiele)

Die Methode `calc_trans()` wird dazu benötigt von Bildkoordinaten zu Weltkoordinaten umzurechnen. Es ist erforderlich ein neues Koordinatensystem einzuführen, da die Bilder die mit OpenCV aufgenommen werden, den Koordinatenursprung oben links haben. Das heißt nach rechts wird die X-Achse positiv und nach unten die Y-Achse positiv. Jedoch beziehen sich die Winkelberechnungen z. B. mit dem `np.arctan2()` auf ein Koordinatensystem welches den Ursprung unten links hat, mit der X-Achse nach rechts positiv und der Y-Achse nach oben positiv. Deshalb muss zwischen diesen beiden Koordinatensystemen transformiert werden. Die X-Achse ist bei den beiden Koordinatensystemen gleich und muss nicht umgerechnet werden. Allerdings muss die Y-Achse umgekehrt werden, indem von der Höhe die Y-Koordinate abgezogen wird.

```
def calc_trans(self, x, y, height):
    yTrans = height - y
    xTrans = x

    return xTrans, yTrans
```

3.3.8 Winkelumrechnung (Thiele)

Der Sphero erwartet in seiner Methode *sphero.roll()* einen Winkel zwischen 0° und 360°. Jedoch kann es bei den Berechnungen vorkommen, dass der Winkel negativ wird oder über die 360° hinaussteigt. Deshalb muss dieser einheitlich auf den Bereich, durch die Methode *phi_in_range()*, gemappt werden. Dazu wird entweder so lange 360° addiert bis der Winkel größer 0° ist oder 360° abgezogen, bis der Winkel kleiner als 360° ist.

```
def phi_in_range(self, phi):  
    while(phi < 0):  
        phi = phi + 360  
    while(phi > 360):  
        phi = phi - 360  
    return phi
```

3.4 Wavefrontplanner (Thiele)

Die Pfadplanung des Spheros erfolgt durch die Verwendung des zuvor bereits erläuterten Wavefrontplanners. Dieses Kapitel beschreibt die Implementation von diesem. Bei dem verwendeten Code handelt es sich um eine Implementation von *societyofrobots*, 2023. Diese wurde abgewandelt und auf die spezielle Aufgabenstellung angepasst. Im Anhang ist der komplette Code abgebildet.

Die Klasse *waveFrontPlanner* bietet verschiedene Funktionen, um z. B. die Start- und Zielposition festzulegen, den Pfad durch Hindernisse zu berechnen oder die Map auf der Konsole auszugeben. Die genauen Methoden können der *Abbildung 4* entnommen werden.

Um den Wavefrontplanner zu nutzen, wird eine spezielle Karte benötigt. Die Karte besteht aus einem 2D-Array, das bei einem Hindernis den Wert 999 und bei einer freien Fläche den Wert 000 haben muss. Um ein solches Array zu erzeugen und weitere kartenspezifische Aktionen auszuführen wie z. B. die Skalierung, wurden diese Funktionen in eine sperate Klasse *mapCreate* ausgelagert, die in Kapitel 3.5 genauer beschrieben wird. *Abbildung 7* zeigt das Karten-Array in einem Plot visualisiert. Hierbei entsprechen die schwarzen Pixel dem Wert 999 und die weißen Pixel dem Wert 000. Das Array basiert auf einem Bild vom Spielfeld, das mit einer Kamera aufgenommen und mit OpenCV bearbeitet wurde. Um jedoch Rechenleistung zu sparen, wurde das Bild vor der Erstellung des Karten-Arrays um den Faktor zehn verkleinert.

Da der Wavefrontplanner in der Praxis leider dazu neigt sehr eng an den Hindernissen vorbei zu Navigieren und der Sphero somit gegen die Hindernisse fährt anstatt sie zu umfahren, wurden die Hindernisse vergrößert. *Abbildung 8* zeigt das vergrößerte Karten-Array. In grau ist ein Beispielpfad eingezeichnet. Auf der linken Seite ist der Startpunkt und auf der rechten Seite der Zielpunkt. Die Beschreibung der Methoden wie das Array erstellt sowie vergrößert wird erfolgt im nächsten Kapitel.

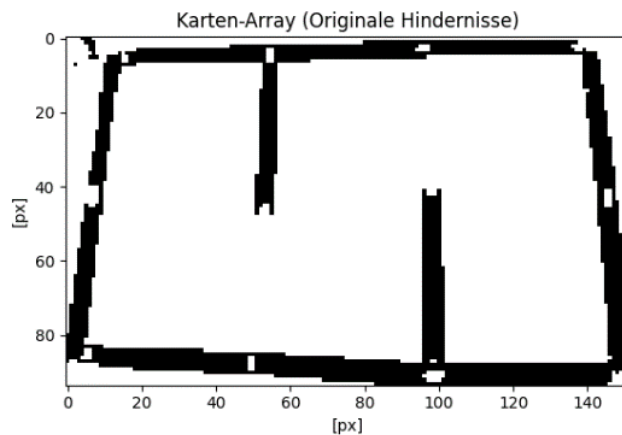


Abbildung 7 Karten-Array Original

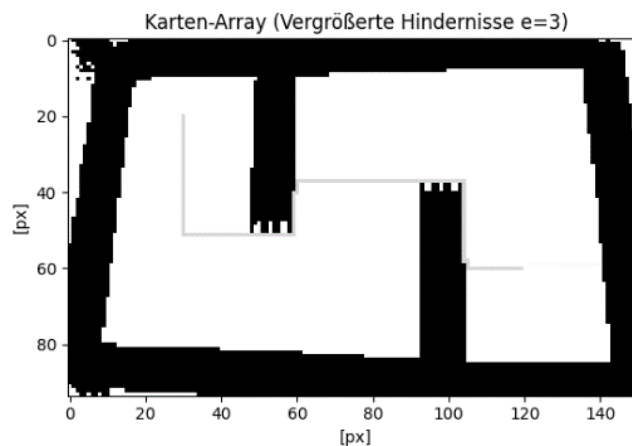


Abbildung 8 Karten-Array vergrößert

Wichtig bei der Verwendung der *waveFrontPlanner* Klasse ist, dass bei dieser Implementation ein Koordinatensystem verwendet wurde, bei dem die X-Achse vertikal und die Y-Achse horizontal verläuft. Dies muss dementsprechend bei dem Festlegen von Start- und Zielpunkt beachtet werden und die X- und Y-Koordinaten vertauscht werden.

Damit ein Pfad durch verschiedene Hindernisse berechnet werden kann, muss neben dem Aufnehmen eines Bildes vom Spielfeld zunächst ein Objekt der Karten-Klasse erstellt und ein Skalierungsfaktor festgelegt werden. Danach kann das Karten-Array durch die methode *Map.create_map()* erstellt werden. Nach dem deklarieren der Start- und Zielkoordinaten, mit den vertauschten X- und Y-Achsen, können diese genau wie die Karte durch die Methode *Map.scale_koord()* skaliert werden und das Wavefrontplanner Objekt mit der Übergabe des Map-Arrays erstellt werden. Durch die Set-Methoden werden Start- und Zielpunkt festgelegt sowie durch den Aufruf von *planner.run()* die Pfadplanung gestartet. Der Pfad von Start- zu Zielpunkt ist in der Variable *path* gespeichert. Zuletzt müssen die Einträge in dem Pfad-Array zurückskaliert werden. Die Koordinaten der Start- und Zielwerte sind hierbei im Bildkoordinatensystem anzugeben und werden lediglich ganz am Ende des Programms in Weltkoordinaten umgerechnet, damit der Sphero diese sinnvoll ansteuern kann.

```

cap = VideoCapture("http://root:root@10.128.41.239:80/mjpg/video.mjpg")
#cap = VideoCapture(4)

img = cap.read()
height, width = img.shape[:2]

scale = 90 #Verkleinerungsfaktor in %
Map = mapCreate() #Map Objekt erzeugen
mapWorldScaled, mapWorldOrg = Map.create_map(scale, img)

#Startpunkt im Bildkoordinaten System
sy = 300      #X-Koordinate im Bildkoordinaten System
sx = 200      #Y-Koordinate im Bildkoordinaten System

#Zielpunkt im Bildkoordinaten System
gy = 1200     #X-Koordinate im Bildkoordinaten System
gx = 200      #Y-Koordinate im Bildkoordinaten System

sx,sy,gx,gy = Map.scale_koord(sx,sy,gx,gy,scale) #Verkleinern der Punkte

start = time.time()
planner = waveFrontPlanner(mapWorldScaled)
planner.setGoalPosition(gx,gy)
planner.setRobotPosition(sx,sy)
path = planner.run(False) #Pfad berechnen
end = time.time()
print("Took %f seconds to run wavefront simulation" % (end - start))

path = Map.rescale_koord(path, scale) #Pfad in Bildkoordinaten
#print(path)

```

Die Anzeige des Karten-Arrays sowie das Plotten des Pfades in das Bild des Spielfeldes wurden in eine separate Methode `plotMap()` ausgelagert. Diese erstellt mehrere Sub-Plots und beschriftet diese. *Abbildung 9* und *Abbildung 10* zeigen den Plot des Spielfelds mit dem eingezeichneten Pfad. Dabei stellt *Abbildung 9* das Spielfeld in dem Koordinatensystem des Programms dar und *Abbildung 10* zeigt das Spielfeld in Bildkoordinaten.

```

def plotMap(self, img, path):
    #Programm Koordinaten
    imgTrans = cv2.transpose(img) # X und Y-Achse im Bild tauschen
    imgPlot, ax0 = plt.subplots()
    ax0.set_title('Programm Koordinaten')
    ax0.imshow(imgTrans)
    ax0.set_xlabel('[px]')
    ax0.set_ylabel('[px]')
    ax0.scatter(path[:,0], path[:,1], color='r')

```

```

#Karten Array Scaled
imgPlot1, ax1 = plt.subplots()
ax1.set_title('Karten-Array (Vergrößerte Hindernisse e=3)')
ax1.imshow(mapWorldScaled, cmap = 'Greys')
ax1.set_xlabel('[px]')
ax1.set_ylabel('[px]')

#Karten Array Normal
imgPlot2, ax2 = plt.subplots()
ax2.set_title('Karten-Array (Originale Hindernisse)')
ax2.imshow(mapWorldOrg, cmap = 'Greys')
ax2.set_xlabel('[px]')
ax2.set_ylabel('[px]')

#Bild Koordinaten
imgPlot3, ax3 = plt.subplots()
ax3.set_title('Bild Koordinaten')
ax3.set_xlabel('[px]')
ax3.set_ylabel('[px]')
ax3.imshow(img)
ax3.scatter(path[:,1], path[:,0], color='r')
return

```

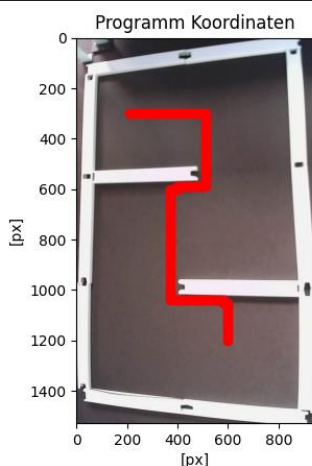


Abbildung 9 Pfad in Programm Koordinaten

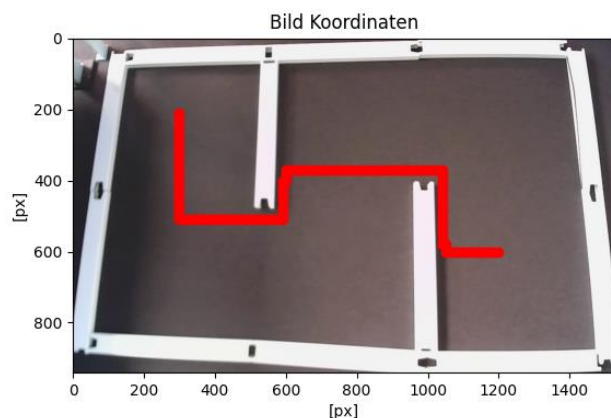


Abbildung 10 Pfad in Bild Koordinaten

Der nachfolgende Abschnitt beschäftigt sich mit dem Umwandeln des Pfades in Weltkoordinaten und dem Transformieren von diesem in einen für den Sphero sinnvollen Pfad. Dieser Abschnitt ist vom Herrn Weber geschrieben und bezieht sich auf die Methode *getPathWelt()*.

Der Pfad, den der Algorithmus ausgibt, besteht aus sehr vielen Punkten, die jedoch alle nahe aneinander liegen. Damit der Roboter nun nicht mit zu vielen Fahrbefehlen überfordert ist, ist es notwendig diesen Pfad zu reduzieren.

Zunächst transformiert die Funktion alle *path*-Koordinaten mit der Funktion *calc_trans* von Bildkoordinatensystem in das Weltkoordinatensystem.

Der Wavefront-Planner kann aufgrund der Rasteraufteilung des Spielfeldes nur eine Strecke mit 90°- bzw. 270°-Knicken angeben.

Um diese Strecke nun auf die nötigsten Punkte zu reduzieren, benötigt man lediglich die Eckpunkte dieser Strecke. Diese lassen sich über einen Algorithmus ermitteln, der die Koordinaten aller Pfadpunkte hintereinander vergleicht. Wenn sich nun eine Koordinate zum Vorgänger verändert und die jeweils andere Koordinate anders ist als die des Nachfolgers, dann schließt der Algorithmus auf einen Eckpunkt und speichert diesen in einer separaten Liste. Neben dem Zielpunkt besteht diese Liste nun aus allen Eckpunkten und reduziert den ursprünglichen Pfad somit um alle Punkte auf der Geraden, die zwischen den Ecken liegen.

Die *getPathWelt*-Funktion erhält als Übergabeparameter den ungekürzten Pfad *path*, die Höhe des Kartenobjekts *height* in Pixeln und das Kartenobjekt selbst. Die Funktion durchläuft die X- sowie die Y-Koordinate der Pfadpunkte an jeder Stelle der Liste und überprüft zunächst auf die Veränderung der Nachfolgerkoordinate und danach auf die Veränderung des Vorgängers der jeweils anderen Koordinate. Im Anschluss auf die Eckenerkennung fügt die Funktion manuell den Zielpunkt hinzu, da dieser über keinen Nachfolger verfügt. Zuletzt gibt die Funktion eine Liste zurück, die aus den X und Y-Koordinaten der Eckpunkte konstruiert ist.

```
def getPathWelt(self,path,height,Mapobj):

    pathWeltX, pathWeltY = Mapobj.calc_trans_welt(path[:,1], path[:,0], height)
    pathEckenX = []
    pathEckenY = []

    for i,px in enumerate(pathWeltX):
        if (i < len(pathWeltX)-1) & (i > 0):
            if px != pathWeltX[i+1]:
                if pathWeltY[i]!=pathWeltY[i-1]:
                    pathEckenX.append(px)
                    pathEckenY.append(pathWeltY[i])
            if pathWeltY[i] != pathWeltY[i+1]:
                if pathWeltX[i]!=pathWeltX[i-1]:
                    pathEckenX.append(px)
                    pathEckenY.append(pathWeltY[i])

    pathEckenX.append(pathWeltX[-1])
    pathEckenY.append(pathWeltY[-1])

    uebergabePath = []
    for i in range(0,len(pathEckenX)):
        uebergabePath.append((pathEckenX[i],pathEckenY[i]))

    return uebergabePath
```

3.5 Kartenerstellung (Thiele)

Dieses Kapitel beschreibt die Klasse *mapCreate()* und deren Methoden. Das vorherige Kapitel hat bereits einen Überblick darüber gegeben welche Operationen auf das Karten-Array angewandt werden müssen und in welchem Format die Karte zu erstellen ist. Die nachfolgenden Unterkapitel geben einen Einblick, wie diese Funktionen umgesetzt wurden. Zunächst wird die Erstellung des Karten-Arrays beschrieben.

3.5.1 Kartenerstellung (Weber)

Wie bereits in Abschnitt 3.4 diskutiert, handelt es sich bei der Implementation des Wavefrontplanners um eine fertige Version. Diese setzt als Eingabe der Karte eine zweidimensionale Liste voraus, die an den jeweiligen Positionen einen bestimmten Wert enthält. Der Wert 0 steht hierbei für eine freie Fläche und 999 für eine Wand. Da die Quelle für die Karte in Form eines Bildes gespeichert wird, ist es notwendig die nötige Schnittstelle zwischen den beiden Formaten herzustellen.

Um Ressourcen zu sparen, arbeitet die Funktion *create_map* nicht mit dem Kamerabild in der Originalauflösung, sondern mit einer skalierten Version des Bildes. Der Skalierungsfaktor von 10% kann in der Main-Methode angepasst werden. Mithilfe der openCV-Methode *resize* kann die Funktion das Kamerabild zu einer gewünschten Auflösung skalieren.

Basierend auf einer Vergleichsrechnung lässt sich sagen, dass die Laufzeit des Algorithmus enorm mit den Dimensionen des Bildes einhergeht. Die Pfadfindung einer Strecke auf einen um 80% verkleinerten Bild benötigt rund zehn Mal so lange wie die Pfadfindung zum selben Punkt auf einem um 90% verkleinerten Bild.

Als nächsten Schritt konvertiert die Funktion das skalierte Bild in ein Graustufenbild, mit der OpenCV-Methode *cvtColor*.

Hindernisse, die sich im Kamerabild als weiße Flächen auf schwarzem Untergrund hervorheben, speichert sich die Funktion in einer Maske ab. Diese Maske fungiert als eine Art Schablone, indem sie bestimmte Bildbereiche, die in einem Spektrum von Grauwerten liegen, erkennt und herausfiltert.

Der Graubereich erstreckt sich von 0 bis 255. Der Bereich, indem Hindernisse erkannt werden, ist abhängig von den Lichtverhältnissen und lässt sich in der Initialisierung der Maske anpassen. In diesem Anwendungsfall ist ein Bereich von 200 bis 255 gewählt, um die Hindernisse identifizieren zu können.

Mithilfe einer *listcomprehension* über die Maske, befüllt die Funktion ein zweidimensionales Array mit „999“, wenn es sich an der jeweiligen eine 1 in der Maske befindet und mit einer „000“, wenn sich dort eine 0 befindet. Anschließend konvertiert die Funktion diese Werte in den Datentypen Integer.

Abschließend gibt die Funktion eine zweidimensionale Liste aus, die den Anforderungen des Wavefrontplanners gerecht wird und auf dem Kamerabild basiert.


```

def create_map(self, scale, cap):
    # Map erstellen
    # Img Objekt
    img = cap.read()

    print('Original Dimensions : ',img.shape)

    scale_percent = 100-scale # percent of original size
    width = int(img.shape[1] * scale_percent / 100)
    height = int(img.shape[0] * scale_percent / 100)
    dim = (width, height)
    # resize image
    resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
    print('Resized Dimensions : ',resized.shape)

    gray = cv2.cvtColor(resized, cv2.COLOR_RGB2GRAY)
    mask = cv2.inRange(gray, np.array([200]), np.array([255]))
    #plt.plot(mask)
    mapOfWorld = [[0]*gray.shape[1]]*gray.shape[0]
    mask_list= np.ndarray.tolist(mask)
    #Markiere alle leeren Zellen mit 000 und alle Zellen mit Hinderniss
mit 999
    for i in range(0,mask.shape[0]):
        mapOfWorld[i] = ['999' if j > 200 else '000' for j in
mask_list[i]]

    mapOfWorld = np.array(mapOfWorld, dtype = int)
    mapOfWorldSized = mapOfWorld.copy()

    e = 3
    #Hindernisse Größer machen
    for i in range(0,mapOfWorld.shape[0]):
        for j in range(0,mapOfWorld.shape[1]):
            for r in range(0,e):
                try:
                    if (mapOfWorldSized[i][j] == 999):
                        mapOfWorld[i][j+e] = 999
                        mapOfWorld[i+e][j] = 999
                        mapOfWorld[i-e][j] = 999
                        mapOfWorld[i][j-e] = 999
                        mapOfWorld[i+e][j+e] = 999
                        mapOfWorld[i+e][j-e] = 999
                        mapOfWorld[i-e][j+e] = 999
                        mapOfWorld[i-e][j-e] = 999
                except Exception:
                    continue

    return mapOfWorld, mapOfWorldSized

```

3.5.2 Skalieren von Start- und Zielpunkt (Thiele)

Da das Karten-Array um den Faktor zehn verkleinert wurde, müssen die Start- und Zielpunkte im gleichen Verhältnis mitskaliert werden. Dazu dient die Funktion `scale_koord()`. Diese bekommt die Start- und Zielkoordinaten übergeben sowie die Variable, die den Skalierungsfaktor in Prozent enthält. Wenn der Skalierungsfaktor z. B. 90 beträgt, so werden die Koordinaten mit dem Faktor 0.1 multipliziert. Dies entspricht der Verkleinerung um den Faktor Zehn. Zuletzt übergibt die Funktion die skalierten Koordinaten.

```
def scale_koord(self, sx, sy, gx, gy, scale):
    scale_percent = 100-scale # percent of original size

    sx = int(sx*scale_percent/100)
    sy = int(sy*scale_percent/100)
    gx = int(gx*scale_percent/100)
    gy = int(gy*scale_percent/100)

    return sx, sy, gx, gy
```

3.5.3 Zurückskalieren der Pfadkoordinaten (Thiele)

Der Pfad, der am Ende des Wavefrontplanners zurückgegeben wird, ist ebenfalls verkleinert und benötigt eine Skalierung auf die ursprüngliche Bildgröße. Dazu wurde die Funktion `rescale_koord()` geschrieben. Diese bekommt die verkleinerten *path* Koordinaten in einem Array und den Skalierungsfaktor übergeben. Danach wird in einer For-Schleife jede Pfadkoordinate vergrößert. Im Anschluss wird der vergrößerte Pfad wieder zurückgegeben.

```
def rescale_koord(self, path, scale):
    scale_percent = 100-scale # percent of original size100

    path = np.array(path)

    for i in range(len(path)):
        path[i] = path[i]*100/scale_percent

    return path
```

3.6 Main-Methode (Weber)

Die Main-Methode ist der erste Aufruf, der über den in ROS2 gestarteten Node zum Einsatz kommt. Diese Methode steuert den gesamten Programmablauf und ruft die zuvor erläuterten Funktionen auf.

Zunächst versucht die Main-Methode eine Kommunikation zum Sphero-Roboter herzustellen sowie den Thread zu starten, der für die Abfrage der Kamerabilder zuständig ist.

Dann wird das jeweilige Videocapture-Objekt erstellt. Je nach Übergabeparameter kann das Programm intern auf die USB-Schnittstelle oder auf eine mit dem Netzwerk verbundene Kamera zugreifen.

Danach stellt das Programm einige Sphero-Parameter wie zum Beispiel die Stabilisierung an sowie die LED-Leuchten um und beginnt die Initialfahrt und die Offsetwinkel-Berechnung.

Mit dem Aufruf `Map.createMap()` wird die Karte basierend auf dem Kamerabild mit der eingestellten Skalierung erzeugt.

Für den Wavefrontplanner liest das Programm zunächst die Startposition des Roboters aus sowie die Zielposition in der Kommandozeile ein. Die Eingabe erfolgt in Bildkoordinaten. Start- wie auch Zielpunkt werden im Anschluss ebenfalls auf den Skalierungsfaktor angepasst. Mit den nun eingetragenen Parametern startet der Algorithmus und sucht einen Pfad von Start zu Ziel.

Wenn der Algorithmus erfolgreich ist, plottet dieser die berechnete Strecke über dem Bild der Karte. Somit kann der Benutzer den berechneten Pfad zunächst validieren.

Das Programm reduziert diesen Pfad auf die anzufahrenden Ecken und lässt den Roboter die jeweiligen Positionen im Pfad nacheinander anfahren. Das Programm endet, wenn die Zielposition erreicht ist.

```
def main(args = None):
    #Verbindung herstellen, Node erstellen
    try:
        rclpy.init(args=args) #Kommunikation Starten
        node = MyNode() #Node erstellen
        sphero = node.connect()
        thread = threading.Thread(target=rclpy.spin, args=(node, ), dae-
mon=True)
        thread.start()

    except Exception as e: # rclpy.ROSInterruptException
        print("Konnte nicht verbinden")
        raise e

    cap = VideoCapture("http://root:root@10.128.41.239:80/mjpg/video.mjpg")
    #cap = VideoCapture(4)
    Map = mapCreate()

    img = cap.read()
    height, width = img.shape[:2]

    scale = 90
```

```

sphero.setLEDColor(255,0,0)
sphero.wait(1)
sphero.setBackLEDIntensity(0)
sphero.stabilization(True)

aOffset = node.calc_offset(sphero, cap, height)

sphero.wait(1)

while(True):
    mapWorldScaled, mapWorldOrg = Map.create_map(scale, cap)

    #Befehle für WaveFrontPlanner/Maperstellung
    sy,sx = node.get_pos(cap,height) #Aktuelle Roboter Pos in Welt
    sy,sx = node.calc_trans(sy,sx,height)

    gx = int(input("Bitte geben Sie die X-Zielkoordinate im Bild Koordinaten-
system ein:")) #X-Koordinate im Weltkoordinaten System
    gy = int(input("Bitte geben Sie die Y-Zielkoordinate im Bild Koordinaten-
system ein:")) #Y-Koordinate im Weltkoordinaten System
    sx,sy,gx,gy = Map.scale_koord(sx,sy,gx,gy,scale)

    start = time.time()
    planner = waveFrontPlanner(mapWorldScaled)
    planner.setGoalPosition(gx,gy)
    planner.setRobotPosition(sx,sy)
    path = planner.run(False)
    end = time.time()
    print("Took %f seconds to run wavefront simulation" % (end - start))

    path = Map.rescale_koord(path, scale)
    planner.plotMap(img, path, mapWorldScaled, mapWorldOrg)
    pathWelt = planner.getPathWelt(path, height, Map)

    for p in pathWelt:
        node.drive_to(sphero, p, aOffset, cap, height)
        sphero.wait(1)

    print("Geschafft")

    if cv2.waitKey(10) & 0xFF == ord('q'):
        break

rcipy.shutdown()

```

4. Fazit und Ausblick (Weber)

Abschließend lässt sich in Bezug auf die Aufgabenstellung sagen, dass für die Entwicklung eines ROS2 *Nodes* ein Erfolg zu verzeichnen ist. Der entwickelte *Node* macht es möglich, den Sphero mit Fahrbefehlen anzusteuern und zu navigieren.

Außerdem ist es möglich dem Roboter Punkte vorzugeben, die er zielgenau nacheinander anfahren kann. Auf diese Funktion ist ein Navigationsalgorithmus aufgesetzt, der eine Ansteuerung des Roboters um Hindernisse herum möglich macht. Damit verbunden ist eine erfolgreiche Bildverarbeitung über eine Kamera, die neben der Position des Sphero-Roboters auch die Positionen der Hindernisse auslesen kann und daraus eine Karte erstellt.

Neben den Erfolgreichen Seiten des Projektes, gibt es auch einige ungelöste Aufgaben, die sich im Laufe des Projektes entwickelt haben. Diese bieten die Chance eine Untersuchung außerhalb des Projektrahmens. So ist es wichtig zu erwähnen, dass die Bluetooth-Verbindung in den Testläufen Verbindungsfehler aufgewiesen hat. Diese Probleme lassen sich vermutlich auf die Entfernung der beiden Kommunikations-Geräte zurückführen oder auf die Menge der gesendeten Fahrbefehle an den Roboter.

In Bezug auf die Bildverarbeitung sind die jeweiligen Lichtbedingungen ausschlaggebend für den Erfolg der Navigationsberechnung und die Positionserkennung des Roboters. Hierfür ist es wichtig die zugrundeliegenden Parameter in der Bilderkennung den eigenen Lichtverhältnissen anzupassen.

Anhang

Abbildungsverzeichnis

Abbildung 1: Sphero Mini Roboter.....	3
Abbildung 2: Karte bevor Wavefront	6
Abbildung 3: Karte nach Wavefront.....	6
Abbildung 4 sphero_mini.py Code-Struktur.....	8
Abbildung 5 Ordner Struktur	8
Abbildung 6 Blob Erkennung	12
Abbildung 7 Karten-Array Original	19
Abbildung 8 Karten-Array vergrößert	19
Abbildung 9 Pfad in Programm Koordinaten	21
Abbildung 10 Pfad in Bild Koordinaten	21

Literaturverzeichnis

- Goubard, C. (03. Juli 2023). *Github*. Von https://github.com/cedricgoubard/sphero_mini abgerufen
- Learnopencv. (10. Juli 2023). Von <https://learnopencv.com/blob-detection-using-opencv-python-c/> abgerufen
- Mazzari, V. (17. Dezember 2019). *ROS2: Was ändert sich gegenüber ROS?* Von <https://www.generationrobots.com/blog/de/ros2-was-andert-sich-gegenuber-ros/> abgerufen
- OpenCV Organisation. (2023). *OpenCV.org*. Von <https://opencv.org/about/> abgerufen
- Overflow, S. (10. Juli 2023). Von <https://stackoverflow.com/questions/43665208/how-to-get-the-latest-frame-from-capture-device-camera-in-opencv> abgerufen
- RoboLab. (2023). *Robolab Hamburg*. Von <https://robolab.hamburg/roboter-detail/sphero-bolt> abgerufen
- societyofrobots. (11. Juli 2023). Von <https://www.societyofrobots.com/robotforum/index.php?topic=10175.0> abgerufen
- Wurth, G. (10. November 2022). *Einstieg ROS2*. Von Hochschule Hamm-Lippstadt: https://wiki.hshl.de/wiki/index.php/Einstieg_ROS_2 abgerufen

Eidesstaatliche Erklärung Thiele

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Die Regelungen der geltenden Prüfungsordnung zu Versäumnis, Rücktritt, Täuschung und Ordnungsverstoß habe ich zur Kenntnis genommen.

Diese Arbeit hat in gleicher oder ähnlicher Form keiner Prüfungsbehörde vorgelegen.

Ratingen, den 14.09.2023



Dennis Thiele

Eidesstaatliche Erklärung Weber

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Die Regelungen der geltenden Prüfungsordnung zu Versäumnis, Rücktritt, Täuschung und Ordnungsverstoß habe ich zur Kenntnis genommen.

Diese Arbeit hat in gleicher oder ähnlicher Form keiner Prüfungsbehörde vorgelegen.

Ratingen, den 14.09.2023



Simon Weber