Physikalische Realisierung der Bewegung von Ball in einer Simulationsumgebung zum Fußballspielen

Studienarbeit

der Hochschule Koblenz
Ingenieurwesen
Mechatronik

von
Oussama Koubaa
Matrikelnummer: 535511

Betreuer der Hochschule:
Prof. Dr.-Ing. Matthias Flach

Abgabedatum: 24.10.2023

Selbstständigkeitserklärung

Ich, Oussama Koubaa erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum: Unterschrift:

Koblenz,den 24.10.2023

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Mechatroniklabor der Hochschule Koblenz wird derzeit ein spannendes Projekt durchgeführt, bei dem Roboter in einem Fußballspiel gegeneinander antreten sollen. Dieses Projekt nutzt MINDSTORMS EV3 Bricks von LEGO als Basis. Das letztendliche Ziel dieses Projekts im Mechatroniklabor ist es, einen autonomen Roboter zu entwickeln, der in der Lage ist, taktische Spielzüge auszuführen, um Tore zu erzielen und gegen andere Roboter anzutreten. Um dieses Ziel zu erreichen, ist die Implementierung einer künstlichen Intelligenz (KI) von entscheidender Bedeutung.

Im Rahmen dieses Projekts ist es entscheidend, eine Simulation des Fußballspiels zu entwickeln, um einen wichtigen Schritt in der Integration der Künstlichen Intelligenz (KI) erfolgreich umzusetzen. Diese Simulation unterteilt sich in zwei Hauptkomponenten: die Simulation der Roboterbewegungen in der Ebene und die Simulation der Ballbewegungen in der Ebene.
Zusätzlich ist die grafische Darstellung der Simulation notwendig, um die Bewegungen und Interaktionen zu visualisieren und anzupassen.

Diese Studienarbeit konzentriert sich hauptsächlich auf die physikalische Realisierung der Ballbewegung, die Erkennung, die Kollision mit den Wänden des Fußballfeldes sowie die Erkennung und die Interaktion mit dem Roboter.

2 Rahmenbedingungen

2.1 Spielfeld

Das Fußballfeld hat eine rechteckige Form mit den Maßen 1220 mm x 1830 mm. Um sicherzustellen, dass Roboter und der Ball nicht vom Spielfeld fallen können, wurde es auf allen vier Seiten eingefasst.
An den Rändern des Spielfelds gibt es Rampen, die dafür sorgen, dass der Ball niemals an den Seiten oder in den Ecken des Spielfelds stecken bleibt. Das hilft den Robotern dabei, immer hinter dem Ball zu bleiben.
In der Mitte der beiden kürzeren Seiten des Spielfelds gibt es jeweils ein Tor, das wie eine 45 mm breite Tasche in das Spielfeld integriert ist.

Abbildung 2-1: Spielfeld
Quelle: Eigene Darstellung

Der Ball wiegt 12 Gramm und hat einen Radius von 25 mm.

Abbildung 2-2: Ball

Quelle: Eigene Darstellung

Der Roboter hat ein Gewicht von 600 Gramm und ist mit einem Radstand von 105 mm sowie einer Gesamtlänge von 150 mm ausgestattet.

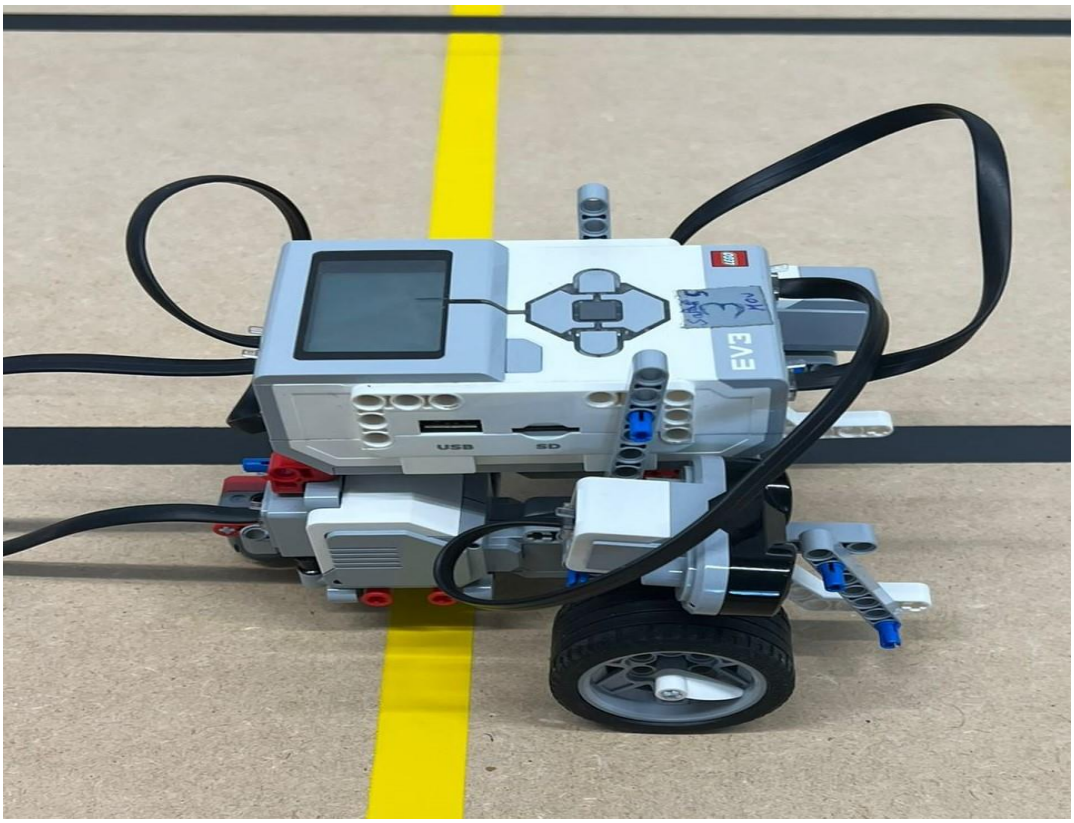

Abbildung 2-3: EV3 Roboter

Quelle: Eigene Darstellung

2.2 Simulation

Für diese Studienarbeit müssen einige kleine Änderungen vorgenommen werden, um dieses Projekt zu vereinfachen, da es schwierig ist, eine völlig realistische Simulation durchzuführen.

Die gleichen Spielfeldgrößen und die gleiche Form, nämlich ein Rechteck, werden beibehalten, wobei die Rampen an den vier Seiten vernachlässigt werden. Der Roboter wird als Rechteck modelliert, um Kollisionen zu erleichtern, da es sonst schwierig ist, den Roboter zu erkennen. Das Koordinatensystem hat nur zwei Dimensionen, was bedeutet, dass sich der Ball nur entlang der x- und y-Achse bewegt, wobei die z-Achse vernachlässigt wird. In dem Versuch, die Simulation etwas realistischer zu gestalten, wird Bodenreibung hinzugefügt, und die Ballgeschwindigkeit wird nach jeder Kollision mit den Wänden verringert.

3 Theoretische Grundlagen

3.1 Koordinatensysteme

Der Ursprung des Koordinatensystems wird in diesem Fall nicht in der Mitte des Feldes gefunden. Aufgrund der Verwendung der Pygame-Bibliothek durch Herr Ben Ammar für die grafische Darstellung der Simulation ist eine Änderung erforderlich.
Das Koordinatensystem von Pygame setzt den Ursprung (0,0) in die obere linke Ecke des Fensters.
Die x-Achse befindet sich an der oberen Wand und die y-Achse an der linken Wand.
Die x-Koordinate wird bei Bewegung von links nach rechts und die y-Koordinate bei Bewegung von oben nach unten erhöht.

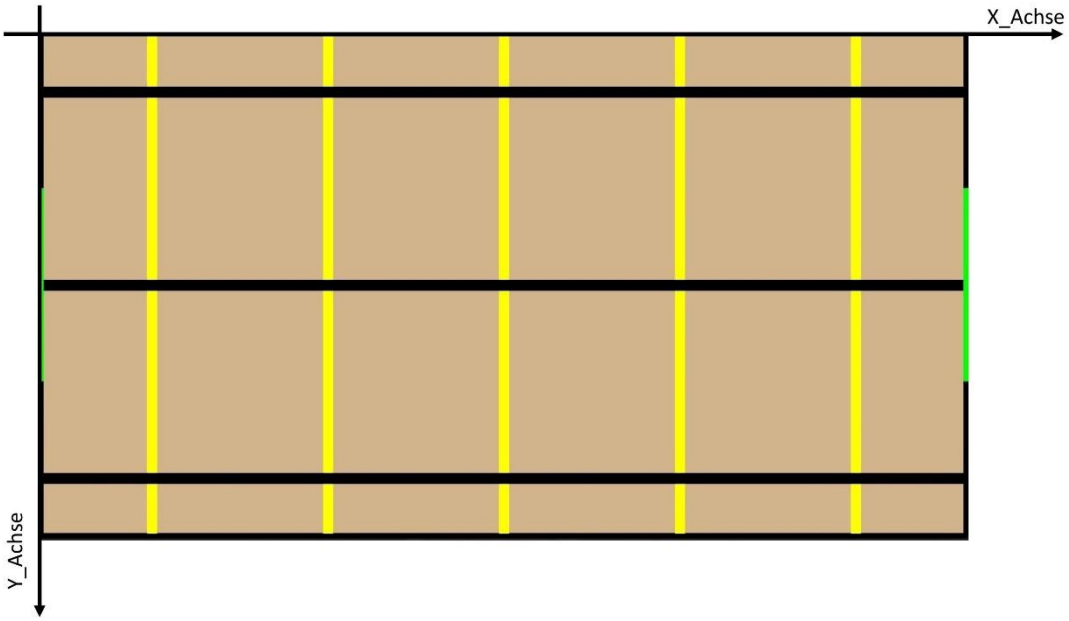

Abbildung 3-1: Inertialkoordinatensystem

Quelle: Eigene Darstellung

3.2 Bewegung des Balls

Zu Beginn der Bewegung des Balls wird die Anfangsposition x_{b0}, y_{b0}, seine Anfangsgeschwindigkeit v_{b0} und der Winkel α seiner Bewegung im Code persönlich ausgewählt.

Mit $$v_{b0} = \begin{bmatrix} \dot{x}_{bo} \\ \dot{y}_{bo} \end{bmatrix} \quad (3.1)$$

Und $$\dot{x}_{bo} = v_{b0}.\cos\alpha \quad (3.2)$$

$$\dot{y}_{bo} = v_{b0}.sin\alpha \quad (3.3)$$

Während der Bewegung wird eine Reibungskraft hinzugefügt, um die Geschwindigkeit des Balls zu verlangsamen und dem Ball ein realistischeres Aussehen zu verleihen.

Mit $$m\ddot{x}_b = -R.cos\alpha \quad (3.4)$$

$$m\ddot{y}_b = -R.sin\alpha \quad (3.5)$$

Und $$R=m_b.g.\mu \quad (3.6)$$

Der Reibung wird in der Simulation durch einen Vektor dargestellt, der entgegengesetzt zur Richtung des Geschwindigkeitsvektors ist.

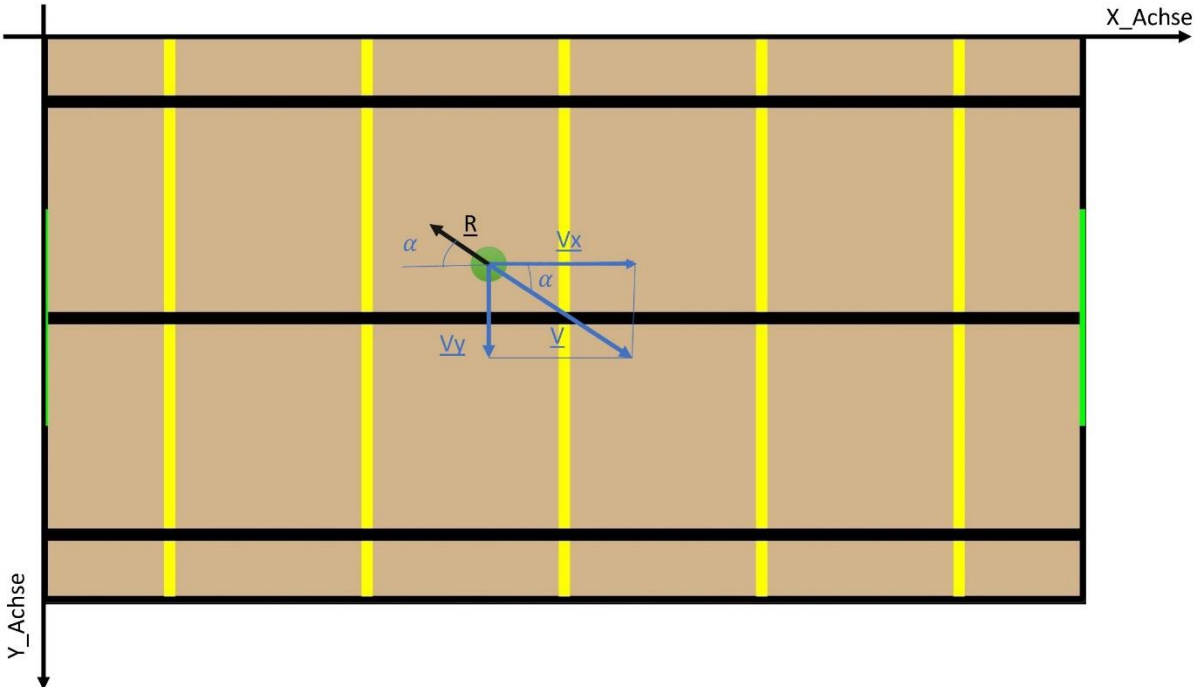

Abbildung 3-2: Ball Bewegung

Quelle: Eigene Darstellung

3.3 Kollision mit Wand

3.3.1 Detektion der Kollision

Die Erkennung von Wänden erfolgt durch die Überprüfung der Position des Balls, wobei der Ball und das Spielfeld dasselbe Inertialsystem haben. Wenn die x-Position des Balls zuzüglich des Ballradius größer ist als die Position der rechten Wand, wird eine Kollision erkannt. Wenn die x-Position des Balls abzüglich des Ballradius kleiner ist, wird ebenfalls eine Kollision erkannt.
Mit derselben Methode kann eine Kollision mit der oberen und unteren Wand erkannt werden, indem die y-Position des Balls mit der Position der oberen und unteren Wand verglichen wird.

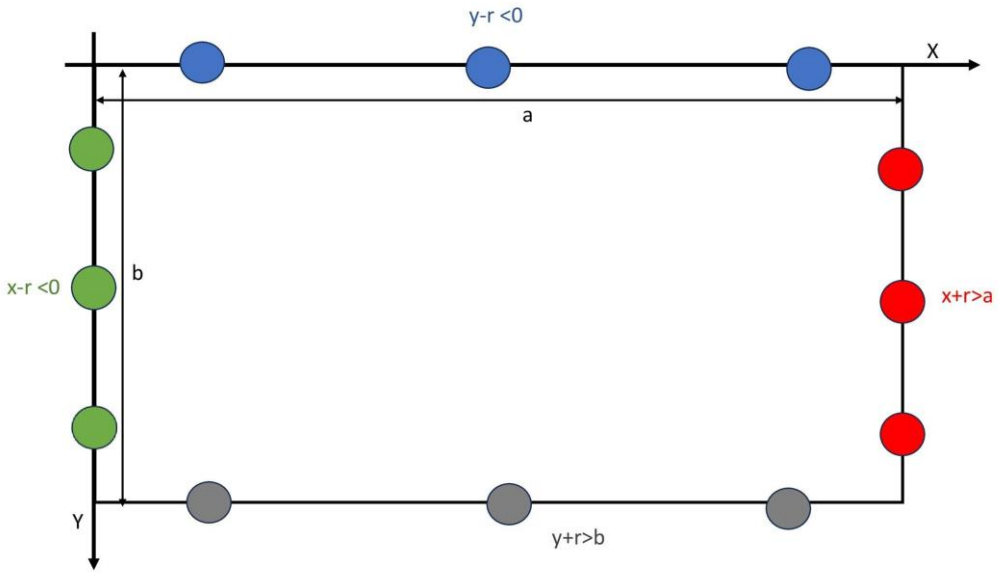

Abbildung 3-3: Bedingung Wandkollision

Quelle: Eigene Darstellung

3.3.2 Zentraler elastischer und teil-elastischer Stoß

Bei der Interaktion zwischen einem Ball und einer Wand können verschiedene Stoßarten beobachtet werden: elastische und teil-elastische. Bei einem elastischen Stoß wird festgestellt, dass die kinetische Energie vor und nach dem Stoß erhalten bleibt. Wenn ein Ball gegen eine vertikale Wand gestoßen wird, wird beobachtet, dass seine horizontale Geschwindigkeitskomponente invertiert wird, während die vertikale Komponente unverändert bleibt. Bei einem teil-elastischen Stoß hingegen wird ein Teil der kinetischen Energie in andere Formen umgewandelt. Der Grad der Elastizität dieses Stoßes wird durch den

Koeffizienten e bestimmt. Ein Wert von e =1 zeigt an, dass ein vollständig elastischer Stoß vorliegt, während ein Wert von e = 0 darauf hinweist, dass ein vollständig unelastischer Stoß vorliegt, bei dem der Ball an der Wand haften bleibt und für teilelastische Stöße liegt e zwischen 0 und 1.

Für eine annähernd realistische Simulation wird der Zusammenstoß mit der Wand nicht als vollständig elastisch betrachtet. Das bedeutet, dass nicht nur die Richtung des Balls geändert wird, sondern auch Geschwindigkeit verloren wird, abhängig vom Elastizitätskoeffizienten des Zusammenstoßes [1].

Nach einer Kollision des Balls mit der Wand wird die Geschwindigkeitskomponente in der jeweiligen Richtung (entweder x oder y) invertiert. Die Geschwindigkeitskomponente Vx wird umgekehrt, wenn der Ball entweder die linke oder rechte Wand berührt.

Die Geschwindigkeitskomponente Vy wird umgekehrt, wenn der Ball entweder die obere oder untere Wand berührt.

$$v_x' = -v_x \tag{3.7}$$

$$v_y' = v_y \tag{3.8}$$

Und für teilelastischer Stoß [1]:

$$v_x' = e.(-v_x) \tag{3.9}$$

$$v_y' = v_y \tag{3.10}$$

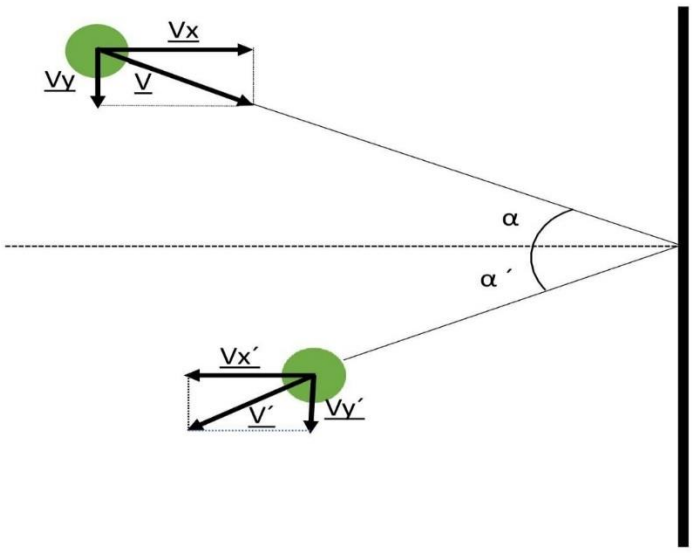

Abbildung 3-4: Reflexionsgesetz

Quelle: Eigene Darstellung

Im Code kann ausgewählt werden, ob die Kollision als elastisch oder teilelastisch betrachtet wird, indem der Elastizitätswert e festgelegt wird.

3.4 Kollision mit Roboter

3.4.1 Koordinatentransformation

Zu Beginn wird die Position des Balls aus den Inertialkoordinaten in das Koordinatensystem des Roboters transformiert. Dies erfolgt mit Hilfe einer einfachen Rotationsmatrix unter Verwendung des Roboter Winkels. Die gleiche Transformation wird auf die Geschwindigkeitsvektoren des Balls und des Roboters angewendet. Dadurch werden die nachfolgenden Kollisionsprüfungen vereinfacht, da der Roboter nach der Transformation effektiv achsenausgerichtet ist.

3.4.2 Relativposition des Balls zum Roboter

Die Ballposition in Inertialkoordinaten wird um die Position des Roboters verschoben, um sie relativ zum Roboter darzustellen.

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_{ball} \\ y_{ball} \end{bmatrix} - \begin{bmatrix} x_{robot} \\ y_{robot} \end{bmatrix} \tag{3.11}$$

3.4.3 Rotation dieser relativen Koordinaten

Die neue Position des Balls $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$ wird dann mithilfe der Rotationsmatrix um den Winkel θ der Roboter gedreht.

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = R(\theta) \times \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \tag{3.12}$$

Mit:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & cos(\theta) \end{bmatrix} \tag{3.13}$$

3.4.4 Umwandlung der Geschwindigkeitskoordinaten des Balls

Genau wie die Position des Balls wird auch seine Geschwindigkeit mithilfe der Rotationsmatrix in das Koordinatensystem des Roboters transformiert.

$$\begin{bmatrix} v_{x1} \\ v_{y1} \end{bmatrix} = R(\theta) \begin{bmatrix} v_{x\_ball} \\ v_{y\_ball} \end{bmatrix} \tag{3.14}$$

3.4.5 Umwandlung der Geschwindigkeitskoordinaten des Roboters

Ähnlich wird die Geschwindigkeit des Roboters, die ursprünglich im Inertialkoordinaten angegeben wurde, in das lokale Koordinatensystem des Roboters transformiert.

$$\begin{bmatrix} v_{robot\_x1} \\ v_{robot\_x2} \end{bmatrix} = R(\theta) \begin{bmatrix} v_{robot\_x} \\ v_{robot\_y} \end{bmatrix} \tag{3.15}$$

Durch diese Transformationen befinden sich sowohl der Ball als auch der Roboter im selben Koordinatensystem. Dies vereinfacht die anschließenden Berechnungen, insbesondere die Kollisionsprüfung.

3.4.6 Detektion Der Kollision

Nach den durchgeführten Transformationen wurde ein rechteckiger Roboter mit den Dimensionen l (Länge) und b (Breite) so positioniert, dass sein Mittelpunkt im Ursprung des Koordinatensystems liegt. Ein Ball mit dem Radius r, dessen Mittelpunkt sich bei den Koordinaten (x_{ball}, y_{ball}) befindet, wurde ebenfalls in dasselbe Koordinatensystem wie der Roboter eingefügt.
Drei Hauptmöglichkeiten für eine Kollision zwischen dem Ball und dem Roboter werden berücksichtigt.

3.4.6.1 Kollision mit vertikalen Seiten des Roboters

Die vertikalen Grenzen des Roboters sind durch die Halblänge des Roboters definiert, wobei die linke Seite bei ´l/2´ und die rechte Seite bei ´l/2´ liegt.
Um festzustellen, ob eine Kollision mit den vertikalen Seiten des Roboters (linke und rechte Seiten des Roboters) vorliegt, werden die äußersten horizontalen Punkte des Balls (links und rechts) mit diesen Grenzen verglichen. Ein Kontakt liegt vor, wenn der Ball sich innerhalb dieser Grenzen befindet.
Die vertikale Position des Balls (seine y-Koordinate) muss sich innerhalb der vertikalen Grenzen des Roboters befinden, was durch $-\frac{b}{2} \leq y_{ball} \leq \frac{b}{2}$ sichergestellt wird.

Gleichzeitig muss überprüft werden, ob der linke oder rechte Rand des Balls den Roboter berührt oder überschreitet.

$$-\frac{l}{2} \leq x_{ball} + r \quad \text{und} \quad \frac{l}{2} \geq x_{ball} - r$$

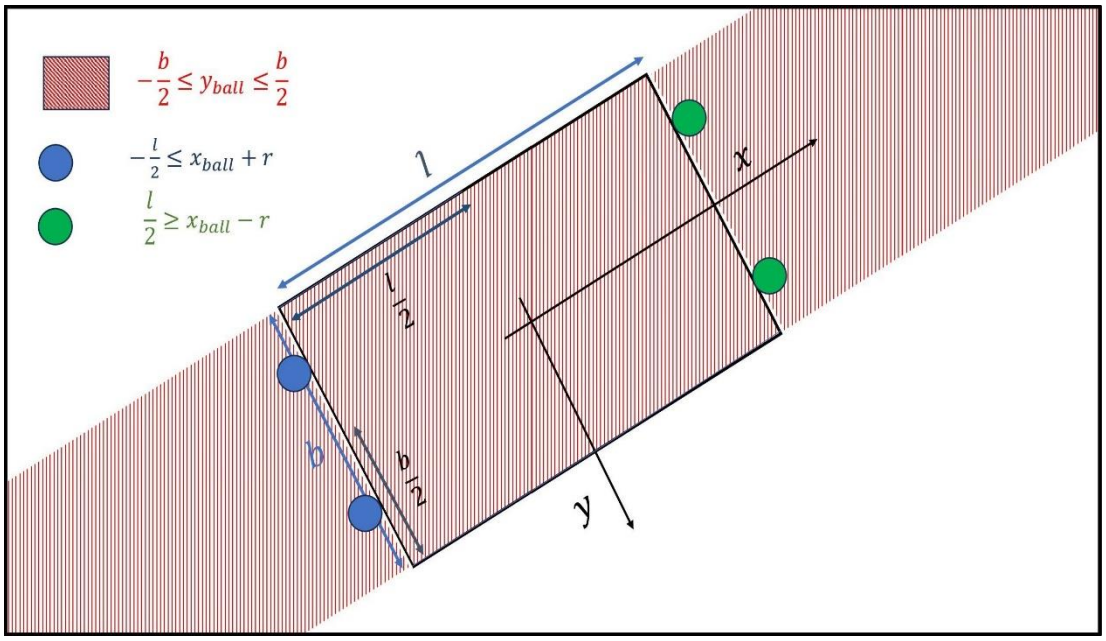

Abbildung 3-5: Bedingung Roboterkollision

Quelle: Eigene Darstellung

3.4.6.2 Kollision mit horizontalen Seiten des Roboters

Mit derselben Logik, die im letzten Kapitel 3.4.6.1 verwendet wurde, wird die Kollision erkannt.

Die horizontale Position des Balls (seine x-Koordinate) muss sich innerhalb der horizontalen Grenzen des Roboters befinden. Dies wird durch die Gleichungen

$$-\frac{l}{2} \leq x_{ball} \leq \frac{l}{2}$$ gewährleistet.

Gleichzeitig muss festgestellt werden, ob der obere oder untere Rand des Balls den Roboter berührt oder überschreitet.

$$-\frac{b}{2} \leq y_{ball} + r \quad \text{und} \quad \frac{b}{2} \geq y_{ball} - r$$

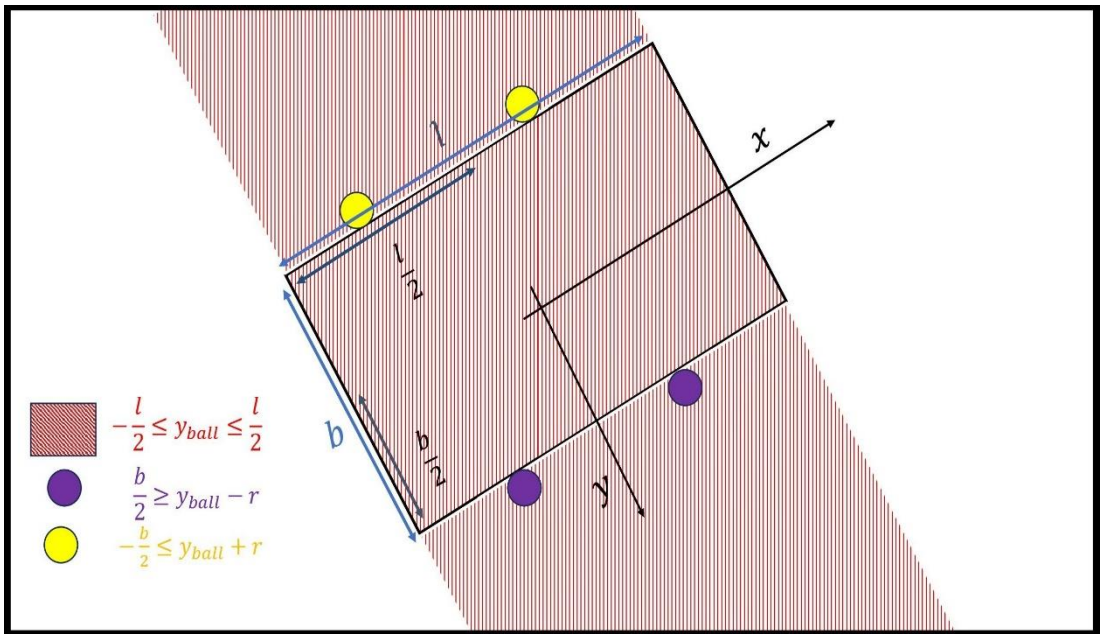

Abbildung 3-6: Bedingung horizontale Roboterkollision

Quelle: Eigene Darstellung

3.4.6.3 Kollisionsprüfung an den Ecken

Bei der Methode zur Kollisionserkennung mit den Ecken des Roboters wird auf das Prinzip des Abstands zwischen einem Punkt (dem Ball) und den Ecken des Rechtecks (dem Roboter) zurückgegriffen. Dabei wird der euklidische Abstand zwischen dem Mittelpunkt des Balls und jeder Ecke des Rechtecks berechnet.

Die Positionen der vier Ecken des Rechtecks, lassen sich wie folgt.

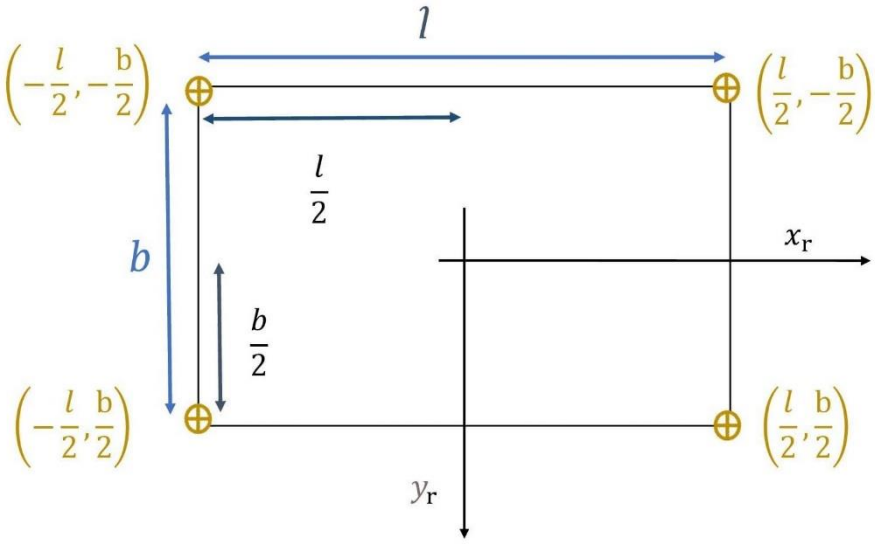

Abbildung 3-7: Positionen der Ecken

Quelle: Eigene Darstellung

Die euklidische Distanz zwischen zwei Punkten (x_1, y_1) und (x_2, y_2) ist definiert als:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.16}$$

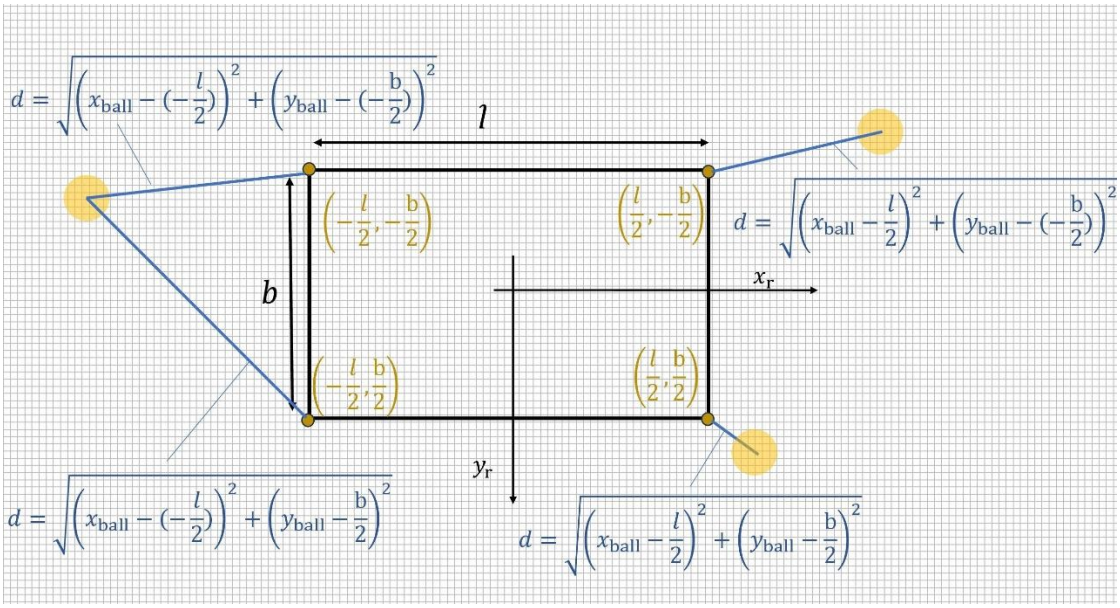

Abbildung 3-8: Die euklidische Distanz

Quelle: Eigene Darstellung

Wenn der Abstand zwischen einer Ecke des Roboters und dem Mittelpunkt des Balls kleiner oder gleich dem Radius des Balls ist, wird diese Ecke des Roboters von dem Ball berührt, und eine Kollision wird verursacht.

Wenn mindestens eine der oben genannten Bedingungen erfüllt wird (Kapitel 3.4.6.1, 3.4.6.2, 3.4.6.3) wird festgestellt, dass eine Kollision zwischen dem Ball und dem Roboter vorliegt.

3.4.7 Zentraler elastischer und teil-elastischer Stoß

Auch wenn die Masse des Balls im Vergleich zur Masse des Roboters vernachlässigbar ist und man ihn als feste Wand betrachten könnte, wurde entschieden, den Zusammenstoß zwischen zwei Körpern unterschiedlicher Masse (m_r, m_b) und Geschwindigkeit (v_r, v_b) zu untersuchen.

Unter Anwendung des Impulserhaltungssatzes wurde festgestellt, dass der gesamte Impuls des Systems vor und nach der Kollision konstant bleibt. Die Impulserhaltung in beiden Dimensionen (x- und y-Richtung) kann separat betrachtet werden.

Dann gilt für die x-Richtung:

$$m_b \cdot v_{bx} + m_r \cdot v_{rx} = m_b \cdot v'_{bx} + m_r \cdot v'_{rx} \tag{3.17}$$

Und für die y-Richtung:

$$m_b \cdot v_{by} + m_r \cdot v_{ry} = m_b \cdot v'_{by} + m_r \cdot v'_{ry} \tag{3.18}$$

Mit den Erhaltungssätzen wurden die Geschwindigkeiten nach dem Stoß mit den vertikalen Seiten des Roboters zum Beispiel wie folgt berechnet [2].

$$v'_{bx} = \frac{m_b \cdot v_{bx} + m_r \cdot v_{rx} - (v_{bx} - v_{rx}) \cdot m_r \cdot k}{m_b + m_r} \tag{3.19}$$

$$v'_{rx} = \frac{m_b \cdot v_{bx} + m_r \cdot v_{rx} - (v_{bx} - v_{rx}) \cdot m_b \cdot k}{m_b + m_r} \tag{3.20}$$

Genau wie im Kapitel 3.3.2 können im Code zwei Fälle unterschieden werden, um den Zusammenstoß des Balls mit dem rechteckigen Roboter zu definieren: entweder elastisch oder teil-elastisch, indem der Wert von k modifiziert wird.
Wenn k gleich 1 ist, wird während der Kollision keine Energie verloren. Das bedeutet, dass die Kollision vollkommen elastisch ist.
Wenn k zwischen 0 und 1 liegt, geht ein Teil der Energie während der Kollision verloren. Je kleiner der Wert von k ist, desto größer wird der Energieverlust. Das bedeutet, dass die Kollision als teilelastisch betrachtet wird und der Ball nach dem Stoß weniger Geschwindigkeit aufweist.

4 Simulation in Python

Nachdem das Modell des Balls erstellt und alle erforderlichen theoretischen Grundlagen berücksichtigt worden sind, wird dem nächsten Schritt zugewendet: der Simulation. Für diese Simulation wird die Programmiersprache Python verwendet.

4.1 Bibliotheken

Numpy (np) ist eine Python-Bibliothek zur Unterstützung von großen, mehrdimensionalen Arrays und Matrizen. Das math Modul stellt mathematische Funktionen bereit, die durch den C-Standard definiert sind, wie z.B. Sinus.

```
1   import numpy as np
2   import math
```

Abbildung 4-1: Bibliotheken
Quelle: Eigene Darstellung

4.2 Eigenschaften des Balls

Hierbei wurde ein Array genutzt, um die Koordinaten [x, y] festzulegen, wobei die Werte als (Datentyp float) erfasst wurden. Wurde ebenfalls ein Array verwendet, das die Geschwindigkeitskomponenten [vx, vy] enthält.
Die Werte für die Position, Geschwindigkeit und den Radius des Balls wurden im Hauptprogramm von Herrn Ben Ammar deklariert und können dort modifiziert werden.

```
7
8  class Ball:
9
10     def __init__(self, x, y , vx ,vy  ) -> None:
11         # Initialisiere die Eigenschaften des Balls
12         self.Mb = 0.0012  # Masse des Balls
13         self.Mr = 0.6  # Masse des Roboters
14
15         self.e_w= 0.9 # Energieverlust-Koeffizient bei Kollision mit der Wand
16         self.k_rb = 1   # Energieverlust-Koeffizient bei Kollision mit dem Roboter
17         self.reibung = 0.2* 9.81  # Reibungskraft
18         self.position = np.array([x, y], dtype=float)  # Anfangsposition
19         self.geschwindigkeit = np.array([vx, vy], dtype=float)  # Anfangsgeschwindigkeit
20
21
```

Abbildung 4-2: Initialisierung
Quelle: Eigene Darstellung

4.3 Bewegung des Balls

Innerhalb der ´ball_bewegung´ Funktion beginnt die Simulation mit der Berechnung der Reibung des Balls in x und y Richtung. Die Funktion signum_block [3] wird dabei verwendet, um das Vorzeichen der Geschwindigkeitskomponenten zu bestimmen. Bei einem positiven vx wird 1.0 zurückgegeben. Bei einem negativen vx wird -1.0 zurückgegeben. Bei einem vx von null wird 0.0 zurückgegeben und und dasselbe für vy. Und so wird sichergestellt, dass die Geschwindigkeit und die Reibung das gleiche Zeichen haben.

Nach der Rechnung wird die Geschwindigkeit des Balls verändert. Dafür wird die Reibung mit der Zeitinkrement (dt) multipliziert. Dieser Wert wird dann von der alten Geschwindigkeit des Balls abgezogen. Das bedeutet, der Ball wird durch die Reibung langsamer.

Im Anschluss wird die Position des Balls basierend auf dieser aktualisierten Geschwindigkeit aktualisiert.

Danach werden Randbedingungen überprüft. Wenn der Ball die Grenzen des Fußballfeldes berührt, wird seine Position so korrigiert, dass er sich wieder innerhalb dieser Grenzen befindet.

Es wurde beschlossen, die Position zu korrigieren, da sonst bei der Simulation häufig Darstellungsprobleme entstehen könnten.

```
22    def ball_bewegung(self, dt, ball_radius, HEIGHT, WIDTH ,winkel_b ):
23
24        # Berechne und wende die Reibung in x- und y-Richtung an
25        reibung_x = signum_block(round(self.geschwindigkeit[0],1)) * self.reibung * np.abs(np.cos(winkel_b))
26        reibung_y = signum_block(round(self.geschwindigkeit[1],1)) * self.reibung * np.abs(np.sin(winkel_b))
27        # Aktualisiere die Position des Balls basierend auf Geschwindigkeit und Reibung
28
29        self.geschwindigkeit[0] -= reibung_x * dt
30        self.geschwindigkeit[1] -= reibung_y * dt
31        self.position[0] += self.geschwindigkeit[0]*dt
32        self.position[1] += self.geschwindigkeit[1]*dt
33
34        # Überprüfung und Anpassung der Ballposition, falls er die Ränder berührt
35        if self.position[0] + ball_radius > HEIGHT:
36            self.position[0] = HEIGHT - ball_radius
37        if self.position[0] - ball_radius < 0:
38            self.position[0] = ball_radius
39        if self.position[1] + ball_radius >WIDTH:
40            self.position[1] = WIDTH - ball_radius
41        if self.position[1] -ball_radius < 0:
42            self.position[1] = ball_radius
43
```

Abbildung 4-3: Funktion der Bewegung des Balls

Quelle: Eigene Darstellung

4.4 Simulation Wandkollision

Die Funktion ball_wand_kollision_x behandelt die Kollision des Balls mit vertikalen Wänden. Das wird durch die Prüfung der x-Position des Balls in Relation zur Feldlänge (HEIGHT) unter Verwendung der if-Bedingung. Wenn die rechte oder linke Wand vom Ball berührt wird, wird die x-Komponente seiner Geschwindigkeit durch eines Elastizitätsfaktors e_w invertiert. Dasselbe wird mit der Funktion ball_wand_kollision_y gemacht, indem die Position des Balls auf der y-Achse in Bezug auf die Position der horizontalen Wände überprüft wird.

```
45      def ball_wand_kollision_x(self,HEIGHT,ball_radius):
46          # Behandlung der Ball-Kollision mit der Wand in x-Richtung
47          if self.position[0] + ball_radius >= HEIGHT or self.position[0] - ball_radius <= 0:
48              self.geschwindigkeit[0] = -self.geschwindigkeit[0] *self.e_w
49
50          return self.geschwindigkeit[0]
51
52      def ball_wand_kollision_y(self,WIDTH,ball_radius):
53          # Behandlung der Ball-Kollision mit der Wand in y-Richtung
54          if self.position[1] + ball_radius >= WIDTH or self.position[1] - ball_radius <= 0:
55              self.geschwindigkeit[1] = -self.geschwindigkeit[1] *self.e_w
56
57          return self.geschwindigkeit[1]
58
```

Abbildung 4-4: Funktionen Wandkollisionen

Quelle: Eigene Darstellung

4.5 Simulation Roboterkollision

In der Funktion "kollision_detektion" werden Koordinatentransformation, Kollisionserkennung, Kollisionsauflösung und Rücktransformation der Koordinaten als vier Hauptziele festgelegt. Der unten angegebene Codeabschnitt beschäftigt sich mit der Umwandlung von Koordinaten, die im Kapitel 3.4.1 bis 3.4.5 theoretisch besprochen wurde.

```
62  def kollision_detektion(self, dt , HEIGHT, WIDTH, ball_radius, robot_x, robot_y, robot_width, robot_height, winkel, robot_vx, robot_vy):
63      # Umwandlung der Ballkoordinaten ins Koordinatensystem des Roboters
64      x1_ball = self.position[0] - robot_x
65      y1_ball = self.position[1] - robot_y
66      x2_ball = x1_ball * math.cos(math.radians(winkel))  - y1_ball * np.sin(math.radians(winkel))
67      y2_ball = x1_ball * math.sin(math.radians(winkel)) + y1_ball * np.cos(math.radians(winkel))
68
69      # Umwandlung der Geschwindigkeitskoordinaten des Balls ins Koordinatensystem des Roboters
70      vx_ball = self.geschwindigkeit[0]
71      vy_ball = self.geschwindigkeit[1]
72      vx1_ball = vx_ball * math.cos(math.radians(winkel)) - vy_ball * np.sin(math.radians(winkel))
73      vy1_ball = vx_ball * math.sin(math.radians(winkel)) + vy_ball * np.cos(math.radians(winkel))
74
75      # Umwandlung der Geschwindigkeitskoordinaten des Roboters ins Roboters Koordinatensystem
76      robot_vx1 = robot_vx * math.cos(math.radians(winkel)) - robot_vy * np.sin(math.radians(winkel))
77      robot_vy1 = robot_vx * math.sin(math.radians(winkel)) + robot_vy * np.cos(math.radians(winkel))
78
```

Abbildung 4-5: Umwandlung von Koordinaten

Quelle: Eigene Darstellung

Danach wird überprüft, ob zwischen einem Ball und einem Roboter eine Kollision vorliegt. Die Logik beruht auf einer Reihe von Bedingungen, die feststellen, ob der Ball innerhalb des Roboterbereichs kommt.

Die ersten beiden if- und elseif-Bedingungen dienen der Überprüfung einer Kollision zwischen dem Ball und dem Roboter.

Die letzte if-Bedingung, dient der Überprüfung der Nähe des Balls zu den Ecken des Roboters. Durch den Einsatz des Satzes des Pythagoras wird der Abstand zwischen dem Zentrum des Balls und jeder Ecke des Roboters ermittelt. Wenn einer dieser berechneten Abstände kleiner oder gleich dem Radius des Balls ist, wird festgestellt, dass eine Kollision des Balls mit dieser Ecke stattgefunden hat.

Alle möglichen Kollisionsfälle werden durch die drei if-Bedingungen im untenstehenden Code berücksichtigt.

```
78
79      # Überprüfung auf Kollision zwischen Ball und Roboter
80      robot_ball_kollision = False
81      if (-robot_height/2 <= x2_ball + ball_radius) and (robot_height/2 >= x2_ball - ball_radius) and (-robot_width/2 <= y2_ball) and (robot_width/2 >= y2_ball):
82          robot_ball_kollision = True
83      elif (-robot_height/2 <= x2_ball) and (robot_height/2 >= x2_ball) and (-robot_width/2 <= y2_ball + ball_radius) and (robot_width/2 >= y2_ball - ball_radius):
84          robot_ball_kollision = True
85      elif np.sqrt((robot_height/2 - x2_ball)**2 +  (robot_width/2- y2_ball)**2) <= ball_radius or \
86          np.sqrt((-robot_height/2 - x2_ball)**2 +  (robot_width/2- y2_ball)**2) <= ball_radius or \
87          np.sqrt((robot_height/2 - x2_ball)**2 +  (-robot_width/2- y2_ball)**2) <= ball_radius or \
88          np.sqrt((-robot_height/2 - x2_ball)**2 +  (-robot_width/2- y2_ball)**2) <= ball_radius :
89          robot_ball_kollision = True
90
```

Abbildung 4-6: Mögliche Roboter-Kollisionen

Quelle: Eigene Darstellung

Wenn eine der letzten drei if-Bedingungen bestätigt wird, muss mit weiteren if-Bedingungen bestimmt werden, mit welcher Seite oder welcher Ecke des Roboters der Ball kollidiert ist.

Nachdem bestimmt wurde, mit welcher Seite des Roboters der Ball kollidiert, wird entschieden, welche Komponente der Geschwindigkeit vx oder vy umzukehren ist, indem die beiden Gleichungen 19 und 20 aus Kapitel 3 angewendet werden.

Für eine optimierte Simulation, genau wie im Kapitel 4.3 mit den Wänden, wird die Position des Balls nach jeder Kollision mit dem Roboter aktualisiert.

Zum Beispiel wenn der Ball mit der rechten Seite des Roboters kollidiert, dann wird die Zeile 105 im Code, den Ball direkt rechts von der Seite des Roboters zu positionieren, sodass er nicht mehr mit dem Roboter überlappt.

```
91      # Behandlung der Kollision des Balls mit der linken Seite des Roboters
92      if robot_ball_kollision:
93          if -robot_height/2 >= x2_ball and -robot_width/2 <= y2_ball and robot_width/2 >= y2_ball:
94              vx1_ball = (vx1_ball * self.Mb + robot_vx1 * self.Mr) / (self.Mb + self.Mr) - ( (vx1_ball-robot_vx1 )*self.k_rb * self.Mr ) / (self.Mb + self.Mr)
95              robot_vx1 =(vx1_ball * self.Mb + robot_vx1 * self.Mr) / (self.Mb + self.Mr) -  ( (robot_vx1- vx1_ball)*self.k_rb * self.Mb ) / (self.Mb + self.Mr)
96
97              x2_ball = -robot_height/2 - ball_radius
98
99          # Behandlung der Kollision des Balls mit der rechten Seite des Roboters
100         if robot_height/2 <= x2_ball and -robot_width/2 <= y2_ball and robot_width/2 >= y2_ball:
101
102             vx1_ball = (vx1_ball * self.Mb + robot_vx1 * self.Mr) / (self.Mb + self.Mr) - ( (vx1_ball-robot_vx1 )*self.k_rb * self.Mr ) / (self.Mb + self.Mr)
103             robot_vx1 =(vx1_ball * self.Mb + robot_vx1 * self.Mr) / (self.Mb + self.Mr) -  ( (robot_vx1- vx1_ball)*self.k_rb * self.Mb ) / (self.Mb + self.Mr)
104
105             x2_ball = robot_height/2 +  ball_radius
106
```

Abbildung 4-7: Simulation Roboterkollision

Quelle: Eigene Darstellung

Nachdem alle if-Bedingungen überprüft wurden, werden die aktualisierten Positionen und Geschwindigkeiten des Balls aus dem Roboter-Koordinatensystem von diesem Codeabschnitt genommen und ins Inertial-Koordinatensystem zurückübersetzt.

```
152
153             # Rückumwandlung der Ballkoordinaten vom Roboter-Koordinatensystem ins Welt-Koordinatensystem
154             winkel = - winkel
155             vx2_ball = vx1_ball * math.cos(math.radians(winkel)) - vy1_ball * np.sin(math.radians(winkel))
156             vy2_ball = vx1_ball * math.sin(math.radians(winkel)) + vy1_ball * np.cos(math.radians(winkel))
157
158             self.geschwindigkeit[0] = vx2_ball
159             self.geschwindigkeit[1] = vy2_ball
160
161             x3_ball = x2_ball * math.cos(math.radians(winkel)) - y2_ball * np.sin(math.radians(winkel)) + robot_x
162             y3_ball = x2_ball * math.sin(math.radians(winkel)) + y2_ball * np.cos(math.radians(winkel)) + robot_y
163
164             self.position[0] = x3_ball
165             self.position[1] = y3_ball
```

Abbildung 4-8: Rückwandlung der Koordinaten

Quelle: Eigene Darstellung

Alle Funktionen der Klasse 'ball' (ball_bewegung, ball_wand_kollision_x, ball_wand kollision_y, kollision_detektion) werden im Hauptprogramm aufgerufen und in einer Schleife wiederholt.

4.6 Modellierung als elastischer Stoß

Mit Hilfe der Bibliothek matplotlib.pyplot können zwei Diagramme der Ballgeschwindigkeit und seiner Position in Abhängigkeit von der Zeit angezeigt werden, um die Simulation besser zu verstehen.

Bei der Modellierung des Kontakts des Balls mit den Wänden und dem Roboter als elastischer Stoß wird beobachtet, dass die Geschwindigkeit des Balls mit der Zeit linear abnimmt, verursacht durch die Reibung. Die Geschwindigkeit des Balls wurde beispielsweise auf 2 m/s initialisiert, mit einem Bewegungswinkel von 60 Grad. Durch die Kollision mit den Wänden und dem Roboter wird die Geschwindigkeit des Balls nicht geändert, sondern nur seine x- oder y-Position, da eine der Geschwindigkeitskomponenten, vx oder vy, nach jeder Kollision umgekehrt wird.

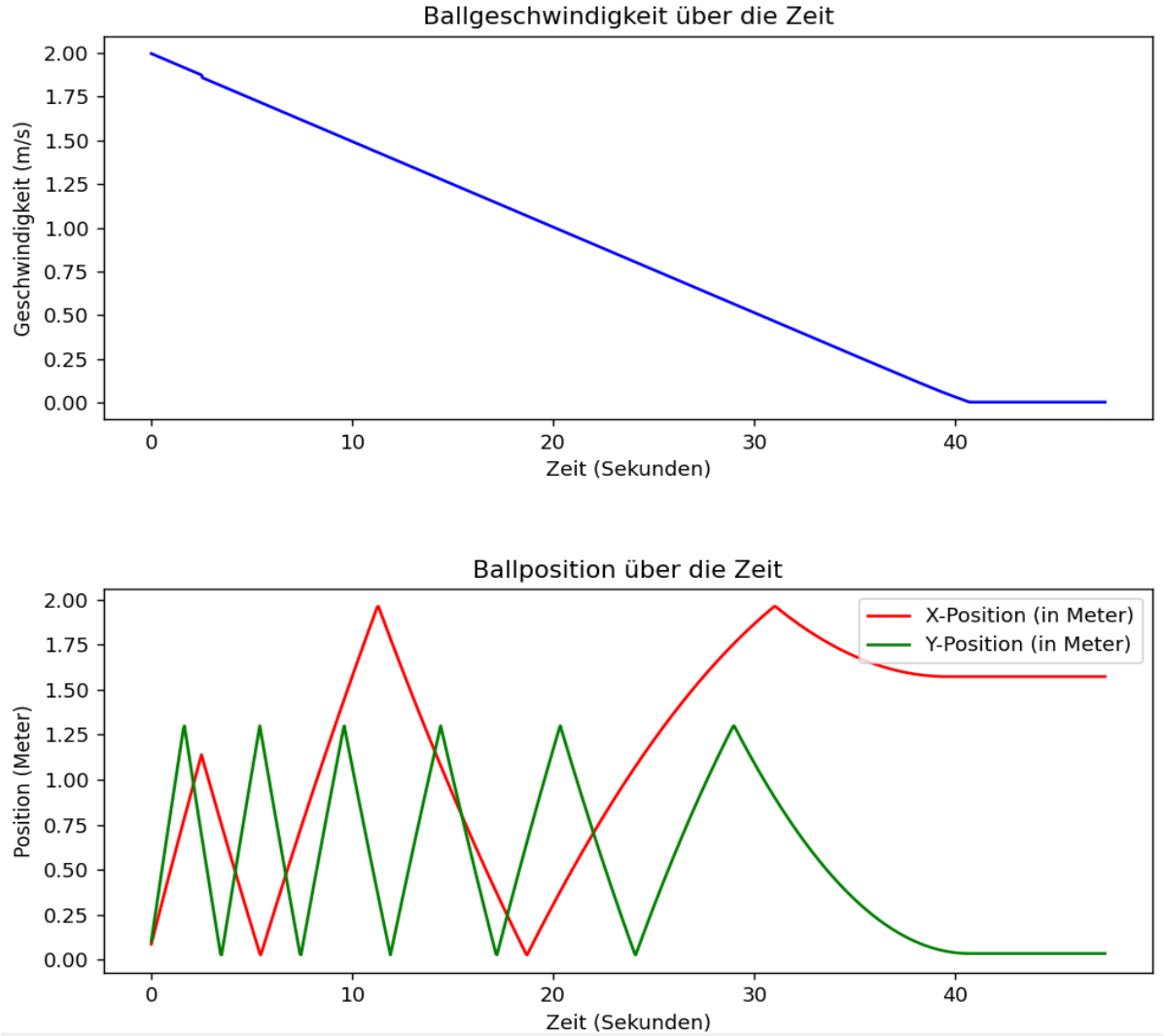

Abbildung 4-9: Modellierung als elastischer Stoß

Quelle: Eigene Darstellung

4.7 Modellierung als teil-elastischer Stoß

Derselbe Geschwindigkeitswert von 2 m/s und derselbe Winkel von 60 Grad wurden beibehalten, jedoch wurden die beiden Energieverlust-Koeffizienten e_w und k_rb von 1 auf 0,9 geändert.

Die Geschwindigkeit des Balls nimmt nicht mehr linear ab, sondern in Stufen. Es wird angenommen, dass eine Kollision des Balls mit einem Roboter oder einer Wand stattgefunden hat. Bei jeder Kollision wird dem Ball Energie entzogen, wodurch seine Geschwindigkeit abrupt abnimmt.

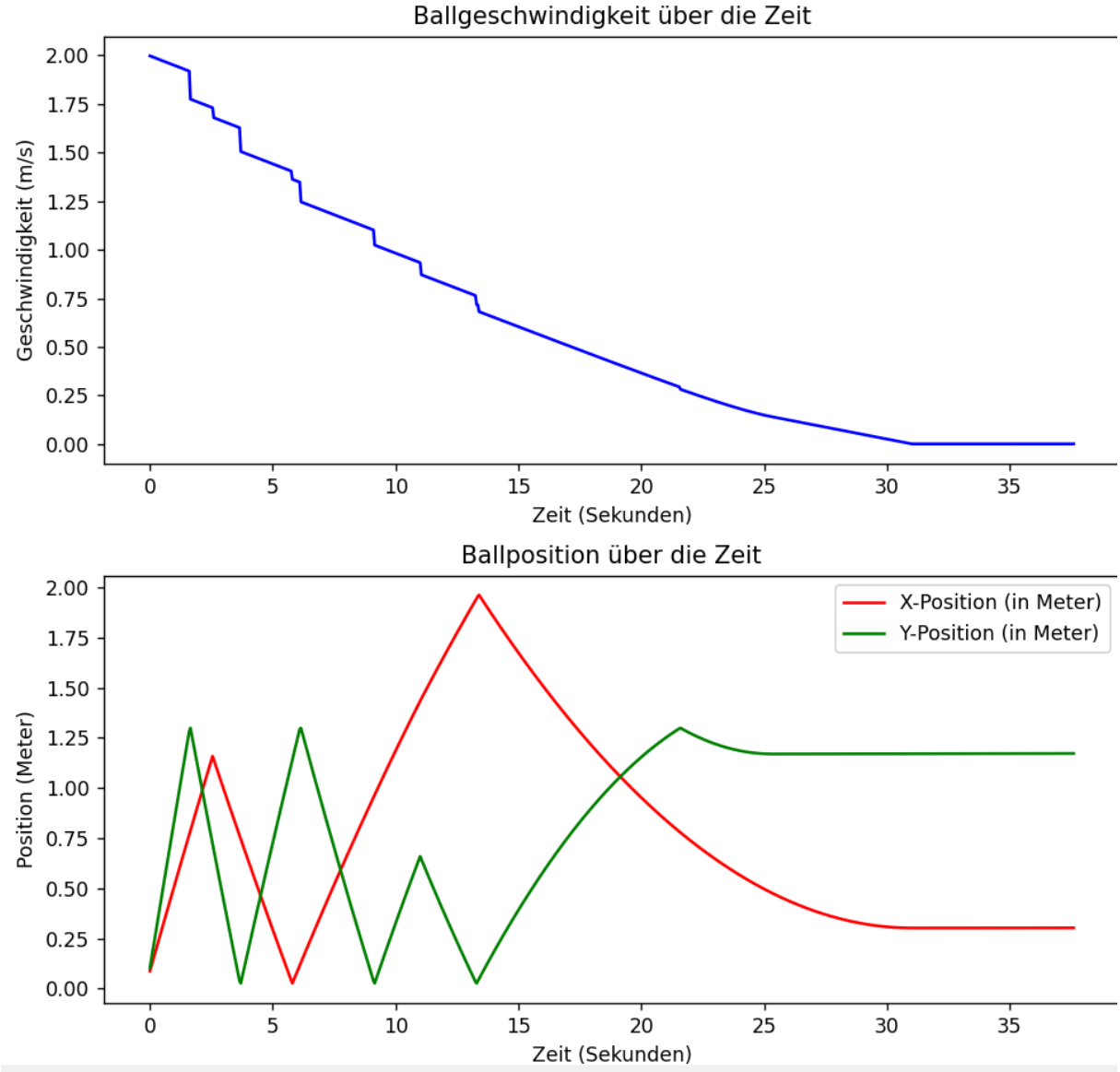

Abbildung 4-10: Modellierung als teil-elastischer Stoß

Quelle: Eigene Darstellung

Fazit und Ausblick

In Zusammenarbeit mit zwei weiteren Arbeiten wurde in dieser Studienarbeit die Basis für ein Fußballspiel mit EV3 Robotern aus dem Mechatronik Labor der Hochschule Koblenz gelegt. Ein zentraler Bestandteil der Arbeit war die Modellierung und Simulation des Balls auf dem Spielfeld.

Durch die integrierten physikalischen Prinzipien und Berechnungen wird eine realistische Darstellung von Bewegungen und Kollisionen ermöglicht. Die Einführung von Reibung und Energieverlust bei Kollisionen erhöht die Authentizität der Simulation.
Im Allgemeinen wird der Code größtenteils ohne Probleme ausgeführt. Die Bewegung des Balls und die Reibung werden fast realistisch dargestellt. Die Erkennung der Wände und des Roboters wird ebenfalls als gut betrachtet. Können aber in bestimmten Situationen, insbesondere bei hohen Geschwindigkeiten und größeren Zeitschritten (dt) Probleme mit der Kollisionserkennung beobachtet werden. Es kann vorkommen, dass der Ball in den Roboter gelangt, bevor eine Kollision erkannt wird, was in der Simulation zu unerwünschten Ergebnissen führen kann.

Mit einem elastischen Stoß wird die Simulation als perfekt angesehen. Bei teilelastischen Stößen kann es jedoch manchmal zu Problemen kommen, wenn der Kollisionswinkel zu groß oder zu klein ist.

Auf der Grundlage dieser Arbeit wurden viele Möglichkeiten für zukünftige wissenschaftliche Untersuchungen und Entwicklungen eröffnet. Könnte die Modellierung durch Hinzufügen des Ballrollens, der Berücksichtigung aller möglichen Kräfte und durch eine Umstellung auf eine 3D-Darstellung weiter verbessert werden.

Anhang

Im Anhang befindet sich auf einem beiliegenden USB-Stick elektronisch gespeichert:

- Studienarbeit im PDF-Format.
- Projekts Code im ZIP-Format.

Literaturverzeichnis

[1] „Ingenieurkurse,“ [Online]. Available: https://www.ingenieurkurse.de/technische-mechanik-dynamik/kinetik-des-massenpunktes/impulssatz-und-impulsmomentensatz/stossvorgaenge-massenpunkt.html. [Zugriff am 09 10 2023].

[2] Redusoft, „Redusoft,“ [Online]. Available: https://www.redusoft.de/info/physprof-hilfe/impulssatz.html. [Zugriff am 10 10 2023].

[3] numpy, „numpy,“ [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.sign.html. [Zugriff am 19 10 2023].