Air Hockey Bot für ABB-Roboter weiterentwickeln

Studienarbeit

für die Prüfung zum

Bachelor of Engineering

des Studiengangs Informationstechnik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Claus Burkhart & René Glockan

Mai 2016

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	8864278 & 5274517, TINF13B3
Betreuer	Prof. Dr.-Ing. Clemens Reitze

Erklärung

Wir erklären hiermit ehrenwörtlich:

1. dass wir unsere Studienarbeit mit dem Thema *Air Hockey Bot für ABB-Roboter weiterentwickeln* ohne fremde Hilfe angefertigt haben;
2. dass wie die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet haben;
3. dass wir unsere Studienarbeit bei keiner anderen Prüfung vorgelegt haben;
4. dass die eingereichte elektronische Fassung exakt mit der eingereichten schriftlichen Fassung übereinstimmt.

Wir sind uns bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Karlsruhe, Mai 2016

Claus Burkhart & René Glockan

Inhaltsverzeichnis

Abbildungsverzeichnis **III**

Listings **IV**

1 Einleitung **1**

2 Aufgabenstellung **2**

3 Herangehensweise **3**

4 Robotersteuerung und -Programmierung **4**
4.1 Robotersteuerung mit dem FlexPendant 5
4.2 Einführung in RAPID 6

5 Kommunikation zwischen Bilderkennung und Roboter **11**
5.1 Kommunikation 11
5.2 Erweiterung des Bilderkennungsprogramms um Kommunikationsmodul . . 12
5.3 Umrechnung vom Bildkoordinatensystem und Roboterkoordinatensystem . 14

6 Untersuchung der Bewegungen des Roboters **15**
6.1 Versuchsdurchführung 15
6.2 Versuchsauswertung 17
6.3 Schlussfolgerungen 21

7 Implementierung der Spielstrategie **23**
7.1 Spielstrategien 23
7.2 Strategieimplementierung 24
7.3 Strategieentscheidung 31

8 Manual **34**
8.1 Einrichtung der Produktivumgebungen 34
8.2 Kalibrierung des Roboters 36
8.3 Ein Spiel starten 39

9 Fazit und Ausblick **42**

Literatur **44**

Abbildungsverzeichnis

4.1 Schaltschrank 5
4.2 FlexPendant 5
4.3 Achsenkonfigurationen für cfx = 0 und cfx =5 ([**rapid2 S. 1160ff**]) 8

5.1 Veranschaulichung der Socketkommunikation 13
5.2 Veranschaulichung der Koordinatensystemunterschiede 14

6.1 Koordinatensystem des Roboters 15
6.2 Bewegung entlang der y-Achse auf der Grundlinie 18
6.3 Bewegung entlang der x-Achse bei y = 570 18
6.4 Diagonalbewegung 19
6.5 Zusammenhang zwischen y-Koordinate und Steigung der Trendlinie 20
6.6 Zusammenhang zwischen y-Koordinate und y-Achsenabschnitt der Trendlinie 20
6.7 Bewegung entlang der x-Achse bei y = 345 21

7.1 Veranschaulichung der Schnittpunktberechnung 25
7.2 Quadranten bei der Spiegelung 26
7.3 Entscheidungsbaum für die Strategieauswahl 33

8.1 Controller hinzufügen 35
8.2 Verzeichnisstruktur 35
8.3 Datentransfer auf den Roboter-Controller 36
8.4 Die drei Positionen zur Kalibrierung des Roboters 38
8.5 Workobject über drei Punkte definieren 38
8.6 Einzeichnen des Spielfeldes 41

Listings

4.1 Beispiel für die Definition einer Position 8
4.2 Beispielaufruf der MoveL-Instruktion 10

6.1 Python-Funktion für die Zeitmessung 16
6.2 Abfahren der Grundlinie 17

7.1 Python-Funktion für Schnittpunktberechnung 24
7.2 Python-Funktion für Punktspiegelung 27
7.3 python-funktion für Defensivstrategie 28
7.4 python-funktion für Offensivstrategien 31

8.1 RAPID-Routine: Kalibrierungstest 36
8.2 RAPID-Routine: Jump-Home 37
8.3 Terminal-Befahl zum starten der Bilderkennung 40

1 Einleitung

Diese Studienarbeit ist eine Fortsetzung mehrerer Studienarbeiten des vergangenen Jahres. Diese beiden Studienarbeiten bestanden darin einen Airhockey-Tisch zu bauen an denen ein ABB Roboter anschließend Airhockey spielen kann. Die Zweite Studienarbeit bestand darin diesem Roboter das Airhockeyspielen beizubringen. Wir führen somit die zweite Studienarbeit weiter. Unsere Vorgänger sind so weit gekommen, dass sie die Bilderkennung soweit fertig hatten, dass der Puck auf dem Spielfeld mithilfe einer Kamera erkannt werden konnte. Zusätzlich konnte eine recht genaue Flugbahn[1] des Puckes bereits berechnet werden. Diese Infos wurden alle auf einer grafischen Oberfläche anschaulich dargestellt. Die Studienarbeit ist dann daran hängen geblieben, dass der Roboter noch nicht zur Verfügung stand. An dieser Stelle setzen wir mit unserer Studienarbeit an. Wir werden dafür die bereits erstellt Softwarearchitektur der Vorgänger übernehmen und erweitern. Sie haben eine Art des Event-Busses verwendet. Jede Komponente bekommt dabei vom Event-Bus eine bestimmte Art von Info und gibt dem Event-Bus eine bestimmte Art von Info. Diese Infos werden in einer „Bag“ gespeichert und weitergegeben. Da dieser Event-Bus sequenziell ausgeführt wird und jede Komponente nur eine spezifische Info einer „Vorgänger“-Komponente benötigt und jede Komponente alle Infos der anderen Komponenten durch das „Bag“ bekommt, hat es mehr den Anschein einer Schleife anstatt eines Event-Busses.[2]

[1] kann man sowas Flugbahn nennen?

[2] Sollte ich das wirklich noch schreiben? klingt irgendwie als würde ich die Architektur der Vorgänger nieder machen.

2 Aufgabenstellung

Das Ziel dieser Studienarbeit ist es einem Roboter von ABB das Airhockey spielen beizubringen. Da dies eine Fortsetzung einer früheren Studienarbeit ist, waren einige Dinge schon fertig. Wir werden auf diese Dinge in unserer Studienarbeit nicht weiter eingehen, da sie in der Studienarbeit unserer Vorgänger schon ausreichend erklärt wurden. Die Arbeit besteht grundlegend aus zwei Teilen. Der eine Teil ist die Ansteuerung des Roboters, das durch ein Programm auf dem Roboter realisiert wird. Der zweite Teil ist die Strategie des Roboters, die auf einem externen Rechner ausgearbeitet wird. Durch diese beiden physikalisch getrennten Programme ist eine Kommunikation zwischen beiden vonnöten. Dies war ein weiterer Punkt dem wir uns in der Studienarbeit gewidmet haben.

3 Herangehensweise

Unser erster Schritt bestand darin uns mit dem Projekt vertraut zu machen. Dies bestand darin uns mit ABB Robot Studio vertraut zu machen und den PC einzurichten auf dem die Bildanalyse und die Roboter-KI laufen soll. Dazu mussten auf dem PC einige Bibliotheken nachinstalliert werden, insbesondere Bibliotheken von OpenCV, die zur Bildanalyse benötigt werden. Anschließend wurde die Kommunikation zwischen PC und Roboter in Angriff genommen. Die Kommunikation findet mithilfe von Sockets statt. Nachdem die Kommunikation zwischen dem PC und dem Roboter stand, wurde eine erste simple Defensivstrategie ausgearbeitet. Diese hat bis zum Ende des 5. Semesters erstmal funktioniert. In einigen Punkten hatte diese noch etwas geschwächelt, aber für den ersten Versuch hat sie besser funktioniert als erwartet.

Für das 6. Semester wird als erstes die Kommunikation erweitert, da diese im Moment nur eine x-Koordinate sendet und der Roboter eine feste y-Koordinate verwendet. Da der Roboter und das Bilderkennungsprogramm leicht unterschiedliche Koordinatensysteme verwenden, muss dafür noch eine bessere Umrechnungsfunktion gefunden werden, bei der möglichst geringe Ungenauigkeiten entstehen. Danach kann man sich voll und ganz auf die Strategien des Roboters wenden. Dazu wird zunächst die Defensive Strategie überarbeitet. Sobald diese funktioniert, kann man an offensive und letztendlich ausbalancierte Strategien denken. Innerhalb des Codes befinden sich einige Konfigurationsvariablen. Bis zum ende des fünften Semesters konnte man diese nur im Code ändern. Diese sollten bis zum Ende in eine Konfigurationsdatei ausgelagert werden. Bis zum Ende der Studienarbeit haben wir es nicht geschafft diese Konfigurationsdatei zu erstellen. Wir haben lediglich eine eigene Klasse in der alle Konstanten drin stehen, die im kompletten Programm verwendet werden.

4 Robotersteuerung und -Programmierung

Bei dieser Studienarbeit wurde ein ABB-Industrieroboter vom Typ „IRB 140“ eingesetzt. Dieser wird gesteuert über eine „IRC 5 Einzelschrank-Steuerung“. Die besteht aus folgenden Modulen: ([3] S.7)

- Antriebsmodul, in dem sich das Antriebsmodul befindet.
- Steuerungsmodul, das den Computer, den Netzschalter, Kommunikationsschnittstellen und den FlexPendant-Anschluss enthält. Die Steuerung enthält auch die Systemsoftware (RobotWare-OS), welche alle grundlegenden Funktionen für Betrieb und Programmierung umfasst.

Sämtliche Operationen und Programmierungen können mithilfe des portablen FlexPendant und über RobotStudio extern ausgeführt werden.

RobotStudio ist eine Software von ABB, die dafür gedacht ist ganze Arbeitsstationen von Industrierobotern virtuell zu erstellen, um daran dann Programme für diese Roboter entwickeln und simulieren zu können, bevor sie auf den Roboter aufgespielt werden. Die Funktionalitäten dieser Software sind sehr umfangreich und deren Beschreibung würde den Rahmen dieser Studienarbeit bei weitem sprengen. Alle Funktionalitäten, die wir genutzt haben und die man benötigt um das Projekt fortzuführen, werden in Kapitel 8 ab Seite 34 beschrieben.

Möchte man sich in die Benutzung dieser Software einarbeiten, findet man auf der Seite http://new.abb.com/products/robotics/robotstudio/how-to-use-it/getting-started gut verständliche Tutorials für Einsteiger. Weiterführende Tutorials findet man auf der Seite http://new.abb.com/products/robotics/robotstudio/tutorials.

Wie man den Roboter über das FlexPendant steuern kann, wird in Abschnitt 4.1 beschrieben.

Zur Programmierung steht die Hochsprache RAPID zur Verfügung, die speziell für die Steuerung von Industrierobotern von ABB entwickelt wurde. Die Datentypen und Funktionen, die nötig sind um damit eine Bewegungen des Roboterarms zu realisieren, sind in Abschnitt 4.2 beschrieben.

4.1 Robotersteuerung mit dem FlexPendant

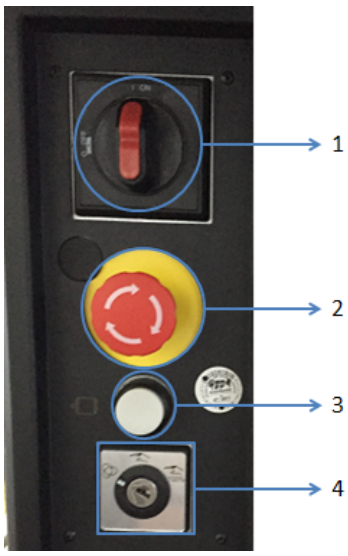

Abbildung 4.1: Schaltschrank

Um den Roboter über das FlexPendant bedienen zu können, muss man zunächst den Betriebsartenwahlschalter (Schalter 4 in Abb. 4.1) auf Handbetrieb stellen. Dafür benötigt man den Schlüssel mit der Aufschrift „Schaltschrank Roboter 1“. Für den Handbetrieb gibt es zwei Modi: Schlüssel auf die rechte Position („Handbetrieb 100%“) bedeutet der Roboter bewegt sich mit voller Geschwindigkeit. Schlüssel in mittlerer Position bedeutet, dass sich der Roboter mit einer reduzierten Geschwindigkeit von maximal 250 mm/s fortbewegt.

Über das FlexPendant kann der Roboter dann entweder über den Joystick (Schalter A in Abb. 4.2) ferngesteuert werden, oder es können Programme ausgeführt werden, die sich auf dem Controller befinden. Dabei muss man jedoch beachten, dass immer der sogenannten „Totmannschalter“ betätigt wird, welcher sich seitlich rechts am FlexPendant befindet. Dieser darf nicht zu schwach und nicht zu stark gedrückt werden und soll damit gewährleisten, dass bei einem Unfall der Roboter sofort stehen bleibt. Ob er richtig betätigt wurde erkennt man, wenn Schalter 3 am Schaltschrank (siehe Abb. 4.1) durchgängig leuchtet.

4.1.1 Fernsteuerung des Roboters

Abbildung 4.2: FlexPendant

Möchte man den Roboter über den Joystick fernsteuern, so muss man zunächst in der oberen linken Ecke des Touchscreens auf das ABB-Logo klicken und anschließend auf „Bewegen“. Der Roboterarm kann auf zwei verschieden Arten bewegt werden:

1. Möchte man das Werkzeug linear entlang der drei Raumachsen bewegen, so muss man Knopf B auf dem FlexPendant (siehe Abb. 4.2) betätigen. Es werden dabei die entsprechenden kartesischen Koordinaten angezeigt, je nachdem welches Werkobjekt man auswählt hat. Diese Positionsangaben beziehen sich auf den Arbeitspunkt des angegebenen Werkzeugs. Ist dies „tool0“ so bezieht sich diese Angabe auf den Mittelpunkt des Werkzeughalters. Zusätzlich wird auch die Orientierung des Werkzeughalters (als Quaternion[1] ausgedrückt) angezeigt.

2. Es ist auch möglich jede des sechs Achsen einzeln anzusteuern. Um die ersten drei Achsen anzusteuern muss man Knopf C auf dem FlexPendant (siehe Abb. 4.2) betätigen. Möchte man die Achsen vier bis sechs bewegen, einfach Knopf C erneut betätigen.

4.1.2 Programm starten

Möchte man über das FlexPendant ein RAPID-Programm auf dem Controller starten, so muss man zunächst in der oberen linken Ecke des Touchscreens auf das ABB-Logo klicken und anschließend auf „Programm Editor“. Hier kann man nun neue Programme schreiben oder bereits vorhandene editieren. Möchte man ein Programm starten, muss man in der unteren Leiste auf den Touchscreen auf „Testen“ klicken. Nun kann man noch den Programmzähler an die gewünscht Stelle setzen: An den Anfang des Programms mit „PZ –> main“, oder zu einem Unterprogramm „PZ –> Routine ...“, oder an die Stelle an der sich der Cursor gerade befindet „PZ –> Cursor“. Zum Starten dann auf „Play“ drücken (siehe Abb. 4.2 Bereich D). Mit den Pfeiltasten in Bereich D kann man das Programm auch schrittweise vorwärts bzw. rückwärts ablaufen lassen.

4.2 Einführung in RAPID

RAPID ist eine Programmiersprache, die von ABB eigens für die Programmierung von Industrierobotern entwickelt wurde. Dabei handelt es sich um eine höhere Programmiersprache. Sie umfasst den Großteil der Funktionalitäten anderer höherer Programmiersprachen

[1] Quaternionen sind eine Erweiterung der reellen Zahlen auf vier Dimensionen ähnlich den komplexen Zahlen. Sie sind sehr vielfältig einsetzbar, beispielsweise könne sie auch zur Beschreibung von Orientierungen im Raum genutzt werden [2].

(bedingt Anweisungen, verschieden Schleifen, so wie zahlreiche Funktionen und Datentypen). Des weiteren enthält RAPID jedoch Instruktionen zum Bewegen des Roboters.

In diesem Abschnitt werden nur die wichtigsten Datentypen und Funktionen beschrieben, die benötigt werden um den Roboterarm bewegen zu können.

Möchte man sich mit dieser Programmiersprache vertraut machen, sollte man sich zunächst Dokument [1] anschauen. Eine vollständig Auflistung und Beschreibung aller Funktion, Instruktionen und Datentypen findet man im Referenzhandbuch [5].

4.2.1 Positionen angeben

Zur Definition von Positionsdaten gibt es in RAPID den Datentyp „robtarget“. Dieser Datentyp ist ein Array aus vier Komponenten:

1. **Translation:**

   Ein dreistelliges Tupel, mit dem die Postion (x,y,und z) des Werkzeugarbeitspunkts in Millimeter angegeben wird.

2. **Rotation:**

   Ein vierstelliges Tupel mit dem die Orientierung des Werkzeugs als Quaternion angegeben wird.

3. **Roboter Konfiguration:**

   Ein vierstelliges Tupel mit dem die Achsenkonfiguration des Roboters (cf1, cf4, cf6 und cfx) angegeben wird. Diese wird als Viertelumdrehung von Achse 1 (cf1), Achse 4 (cf4) und Achse 6 (cf6) definiert. Die erste positive Viertelumdrehung (0° - 90°) der drei Achsen wird dabei zum Beispiel mit 0 definiert. Mit cfx wird eine der acht möglichen Roboterkonfigurationen gewählt, die von 0 bis 7 nummeriert sind. Diese legen fest, wie die Roboterposition in Relation zu den drei Singularitäten stehen. Abbildung 4.3 zeigt an zwei Beispielen, wie sich unterschiedliche Werte auswirken können.

4. **Externe Achsen:**

   Es ist möglich bis zu sechs weiter externe Achsen zu berücksichtigen. Hat man keine externen Achsen, muss man hier für alle sechs Werte „9E9“ einsetzen.

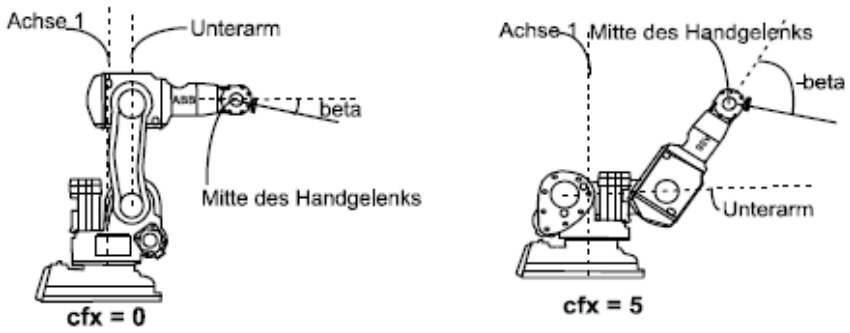

Abbildung 4.3: Achsenkonfigurationen für cfx = 0 und cfx =5 ([**rapid2 S. 1160ff**])

Dabei ist zu beachten, dass die Translations- und Rotations-Koordinaten sich immer auf das aktuelle Objekt-Koordinatensystem beziehen. Ist kein Werkobjekt angegeben, ist dies das Weltkoordinatensystem.

Die Definition einer Positions-Variablen könnte damit beispielsweise lauten:

```
1 CONST robtarget Target_1:=[[453,26,610],[1,0,0,0],[-1,0,0,0],[9E9,9E9,9
      E9,9E9,9E9,9E9]];
```

Listing 4.1: Beispiel für die Definition einer Position

4.2.2 Pfade abfahren

Zum Bewegen des Roboters bietet die Programmiersprache RAPID zahlreiche Möglichkeiten. So kann man beispielsweise mit MoveAbsJ-Instruktion jede Achse einzeln ansteuern, indem man dem Roboter eine absolte Achsenposition übergibt. Zur Realisierung einer Kreisbewegung kann MoveC verwendet werden. MoveJ wird verwendet, um den Roboter rasch von einem Punkt an einen anderen zu bewegen, wenn diese Bewegung nicht geradlinig sein muss. In diesem Projekt sollte dies aber gerade der Fall sein. Zur Ansteuerung des Roboters wurde deshalb ausschließlich die Instruktion MoveL genutzt, mit der sich lineare Bewegungen realisieren lassen.

Die Instruktion MoveL erwartet mindestens vier Pflicht-Übergabeparameter, außerdem können ihr noch acht weitere optionale Parameter übergeben werden. Im folgenden werden die vier Pflichtkomponenten und die beiden wichtigsten optionalen Komponenten beschrieben.

1. **Zielpunkt:**

   Eine Variable vom Typ „robtarget“ (siehe 4.2.1).

2. **Geschwindigkeit:**

   Parameter von Datentyp „speeddata“, der die Geschwindigkeit des Werzeugarbeitspunktes, der Werkzeugumorientierung und der externen Achsen definiert. Im System-Module „BASE“ auf dem Roboter-Controller findet man bereits vordefinierte Variablen dieses Typs. Übergibt man beispielsweise die Variable „v1000“, soll sich der Roboter mit einer Geschwindigkeit von 1000 mm/s bewegen.

3. **Zone:**

   Variable vom Typ „zonedata“, die angibt wie nahe sich der Werkzeugarbeitspunkt an der angegeben Zielposition befinden muss, bevor der Roboter an die nächste Position bewegt werden kann. Auch hierfür sind im System-Modul „BASE“ bereits Variablen vordefiniert. In diesem Projekt sollte immer die übergebene Position erreicht werden. Dies kann man durch Übergabe der vordefinierten Variable „fine“ erreicht werden. Verwendet man stattdessen beispielsweise „z10“, bedeutet dies, dass der Roboter die Ecken schneiden kann, sobald er weniger als 10 mm von der Zielposition entfernt ist.

4. **Werkzeug:**

   Angabe, welches Werkzeug am Roboter montiert wird. Dessen Arbeitspunkt wird dann zur angegebenen Position bewegt. In diesem Projekt wurde hier immer „tool0“ verwendet. D.h. es wird immer der Mittelpunkt des Montagflansches an der Spitze des Roboters an die übergebene Position bewegt.[1]

5. **Werkobjekt:**

   Dies ist ein optionaler Übergabeparameter vom Typ „wobjdata“. Er gibt an auf welches Koordinatensystem sich die Roboterposition in der Instruktion bezieht. Übergibt man kein Werkobjekt wird das Weltkoordinatensystem verwendet.

6. **Zeit:**

   Dies ist ein optionaler Übergabeparameter vom Typ „num“. Also einfach ein Zahlenwert, der die Gesamtdauer, die der Roboter für die Bewegung benötigen soll, in Sekunden angibt. Dieser Wert ersetzt dann die entsprechenden Geschwindigkeitsdaten.

Der Aufruf einer MoveL-Instruktion könnte dann beispielsweise lauten:

[1] Dies kann man nur machen, wenn der Roboter ausschließlich linear bewegt wird und somit der Arbeitspunkt des Werkzeugs immer um den gleichen Vektor zum Mittelpunkt des Montageflansches verschoben ist.

```
1 MoveL Target_1,v500,fine,tool0 \WObj:=Workobject_Table;
```

Listing 4.2: Beispielaufruf der MoveL-Instruktion

5 Kommunikation zwischen Bilderkennung und Roboter[1]

5.1 Kommunikation

Die Kommunikation zwischen dem Roboter und dem Airhockeyprogramm erfolgt über das Ethernet. Dabei haben wir uns für die Kommunikation über Sockets entschieden, da dies die gängigste Methode ist und sowohl von der Programmiersprache des Roboters als auch von Python beherrscht wird. Der Roboter fungiert dabei als Client der sich mit einem Server verbindet. Der PC mit der Roboter-KI ist in diesem Fall der Server. Es hat keinen besonderen Grund wieso wir das so rum getan haben. Dies kann genauso gut auch andersrum implementiert werden, also mit dem Roboter als Server und dem PC-Programm als Client. Wir hatten beide Variante ausgetestet und mehrfach zwischen beiden hin- und hergewechselt. Da wir aber bei unseren Tests keine Geschwindigkeitsunterschiede zwischen beiden feststellen konnten, haben wir dann letzten Endes die oben genannte Variante einfach gelassen.

Mithilfe von Sockets wird eine Folge von Bytes übertragen. Da sowohl erweiterter ANSI-Code als auch UTF-8 ein Byte groß sind, könnte man diese Folge von Bytes auch als eine Folge von Zeichen interpretieren. Wie dies die jeweilige Programmiersprache interpretiert sollte man am besten im jeweiligen Handbuch nachlesen. In unseren Fall hat RAPID und Python 2.7 die übertragenen Daten als String angesehen, also als eine Folge von Zeichen. Übertragen wurde nur die Position, zu der sich der Roboter bewegen soll. Da sich das Spielfeld nur auf einer 2-Dimensionalen Ebene befindet, waren dazu nur eine X- und Y-Koordinate nötig. Diese wurden durch ein Semikolon getrennt, wodurch die Daten wieder recht einfach auf Seite des Roboters getrennt werden konnten. Anfänglich war das nur eine Kommunikation in eine Richtung, wurde aber später noch ergänzt um eine Art „Bereit“, das vom Roboter zurück gesendet wurde, sobald dieser seine Position erreicht hat und neue Befehle entgegennehmen kann.

[1] besseren Namen einfallen lassen, der dem neuen Inhalt gerecht wird(Kommunikation und Umrechnung oder so

5.2 Erweiterung des Bilderkennungsprogramms um Kommunikationsmodul

Die Kommunikation zwischen PC und Roboter erfolgt wie in Kapitel 5 auf Seite 11 beschrieben mithilfe von Sockets. Der PC öffnet einen Serversocket und der Roboter einen Clientsocket. Die Socketkommunikation funktioniert nach dem TCP/IP Verfahren. Man weiß also immer ob die Kommunikation zu Stande kam. Ursprünglich hatten wir für jede Übertragung einen neuen Socket erstellt, die Daten übermittelt und anschließend den Socket wieder geschlossen. Dies haben wir so praktiziert da in der Python Dokumentation geschrieben stand, das Sockets normalerweise nur für eine Übertragung oder eine kleine Folge von Übertragungen genutzt werden und anschließend wieder geschlossen werden.

> When the connect completes, the socket s can be used to send in a request for the text of the page. The same socket will read the reply, and then be destroyed. That's right, destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).[4]

Da auch der Socket auf der Serverseite wieder freigegeben werden musste, konnte bei einem erneuten Kommunikationsversuches des Clients keine Verbindung zum Serversocket des Roboters hergestellt werden. Dadurch wussten wir auch automatisch, dass der Roboter sich noch bewegt und keine neuen Befehle entgegen nehmen kann. Wir hatten für die Sockets eine Timeoutzeit von 1/100s eingestellt. Somit hat ein Kommunikationsversuch mit dem Roboter maximal 1/50s benötigt. Einmal 1/100s für die Verbindung aufbauen und einmal 1/100s für die Daten senden. Da wir mit der angebauten 30 FPS Kamera nur alle 1/30s neue Bilder erhielten, mit denen man neu rechnen muss, war diese 1/50s Maximalzeit auch ausreichend. Es hat sich aber herausgestellt, das anscheinend durch das Socket öffnen und schließen eine viel größere Zeit benötigt wird. Denn zwischen dem senden eines Befehls und dem erneuten senden eines Befehls vergingen mindestens eine halbe Sekunde, selbst wenn sich der Roboter gar nicht bewegen muss. Hinzu kam noch, dass das Schließen des Sockets auf Serverseite länger benötigt als auf Clientseite. Die Zeit die zum Schließen des Serversockets benötigt wurde, hat wahrscheinlich auch diese halbe Sekunde Latenzzeit verursacht. Es entstand aber auch noch ein anderer Nebeneffekt dadurch. Denn der Clientsocket hat in dieser Zeit immer noch eine Verbindung mit dem Serversocket erstellen können und hat somit auch noch Daten geschickt. Diese Daten kamen aber nicht mehr beim Serversocket an, sondern wurden irgendwo zwischen gebuffert. Beim nächsten öffnen des Serversockets wurden dann diese Daten sofort empfangen und verarbeitet. Durch diesen Buffer entstanden immer Ausreißer des Roboters, die wir uns lange Zeit nicht erklären konnten. Wir haben dieses Problem dann erst mitbekommen, nachdem wir ein paar Tests durchgeführt haben um die Geschwindigkeit des Roboters

festzustellen. Das Programm für die Tests hat dann häufig bis zu 3 Werte hintereinander geschickt, bevor es gewartet hat, das der Roboter einen neuen Serversocket geöffnet hat.

Wir führten daraufhin einen Two-Way-Handshake ein. Nach dem der Roboter seine Zielposition erreicht hat, hat dieser ein OK zurückgeschickt.Ursprünglich hatten wir dann solange gewartet bis der Roboter sein OK geschickt hat. Dabei hatten wir nur eins nicht beachtet und zwar dass das Programm sequenziell abläuft und während der Wartezeit wie eingefroren ist. Das heißt wir haben dann auch nur alle halbe Sekunde das neue Bild von der Kamera ausgewertet. Dadurch konnte das ganze Programm nicht mehr ordentlich reagieren, da keine vernünftigen Puckbewegungen mehr errechnet werden konnten. Beim zweiten Versuch, wurde pro Programmzyklus nur einmal überprüft ob der Roboter sein OK gesendet hat. Wenn dieses OK nicht kam wurde auf das neue Bild gewartet und dann erneut abgefragt bis der Roboter sein OK gesendet hat. Dies wird im Bild 5.1 auf Seite 13 nochmal grafisch zur besseren Verständlichkeit dargestellt. Dadurch konnten wir zwar die Ausreißer stark reduzieren, aber es gab immer noch die hohe Latenzzeit von mindestens einer halben Sekunde. Am Ende haben wir uns dann gegen die Empfehlung der Python Dokumentation entschieden und nur am Anfang des Programmstartes einen Socket zu öffnen und diesen über die komplette Programmlaufzeit offen zu lassen. Wir konnten auch bei längeren Laufzeiten von einer halben Stunde und mehr keine Irregularitäten feststellen, weshalb wir entschieden haben, dies so zu belassen. Der einzige Nachteil war, dass wir die Sockets auf diese Art und Weise nicht ordentlich schließen konnten, denn es gibt in Python keinen Deconstructor, wie man ihn aus anderen Objektorientierten Programmiersprachen kennt. Deshalb muss man nach Abbruch des Programms 1-2 Minuten warten bis das Betriebssystem den Socket von alleine wieder schließt. Leider kann man ohne eine Referenz auf den Socket diesen nicht manuell schließen. Und da wir nicht jedes Mal den Programmcode umschreiben wollten um einen neuen Port einzutragen, mussten wir zwischen den Tests immer ein Weilchen warten.

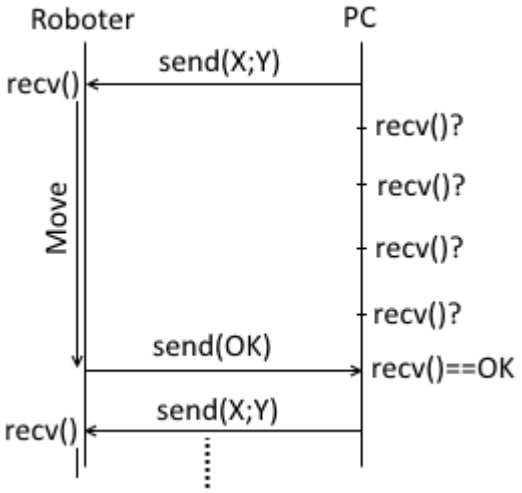

Abbildung 5.1: Veranschaulichung der Socketkommunikation

5.3 Umrechnung vom Bildkoordinatensystem und Roboterkoordinatensystem

Bilderkennung und Roboter nutzen unterschiedliche Koordinatensysteme. Das Bildkoordinatensystem enthält für X und Y Werte von 0 bis 1. Dieses geht von einer Ecke (0/0) bis in die entgegengesetzte Ecke (1/1). Es verwendet ein normalisiertes Koordinatensystem. Somit ist es abhängig, wie die Kamera auf das Spielfeld zeigt. Der Abstand zwischen 2 Punkten ist somit mit unterschiedlichen Kameraeinstellungen auch unterschiedlich. Das Roboterkoordinatensystem verwendet stattdessen mm für ihre Koordinaten. Ein Punkt (0/0) ist somit immer 10mm vom Punkt (10/0) entfernt. Hinzu kommt aber noch, das das Roboterkoordinatensystem einen kleinen Offset vom Bildkoordinatensystem hat. Der Punkt (0/0) im Roboterkoordinatensystem ist ein anderer Punkt im Bildkoordinatensystem. Das liegt daran, dass der Schläger einen gewissen Durchmesser hat und somit nicht exakt in die Ecke gefahren werden kann bei der Kalibrierung. Diesen Unterschied kann man auch gut in der Grafik 5.2 auf der Seite 14 erkennen. Für die Umrechnung haben wir folgende Gleichungen aufgestellt:

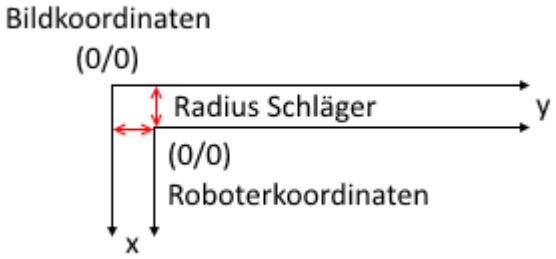

Abbildung 5.2: Veranschaulichung der Koordinatensystemunterschiede

$$x_{Roboter} = (x_{Bild} * Tischtiefe) - DurchmesserSchlaeger/2.0 + x_{Bild} * durchmesserSchlaeger$$

$$y_{Roboter} = (y_{Bild} * Tischbreite) - DurchmesserSchlaeger/2.0 + y_{Bild} * durchmesserSchlaeger$$

Nur in einem Fall nutzen wir die Rücktransformation für die Y-Koordinate. Dafür wurde die Gleichung folgendermaßen umgestellt:

$$y_{Bild} = (y_{Roboter} + durchmesserSchlaeger/2.0)/(Tischbreite + durchmesserSchlaeger)$$

6 Untersuchung der Bewegungen des Roboters

Um dem Roboter sinnvolle Steuerungsbefehle geben zu können, ist es nicht nur wichtig die Bewegung des Pucks richtig vorhersagen zu können. Vielmehr muss man, vor allem um bei einem Angriffsschlag den Puck zu treffen, auch wissen wie der Schläger sich bewegt. Aus diesem Grund haben wir eine ausführliche Untersuchung der Roboterbewegung durchgeführt, welche in diesem Kapitel beschrieben wird. Um die beschriebenen Bewegungen besser nachvollziehen zu können, ist in Abbildung 6.1 das zugrundeliegende Koordinatensystem dargestellt.

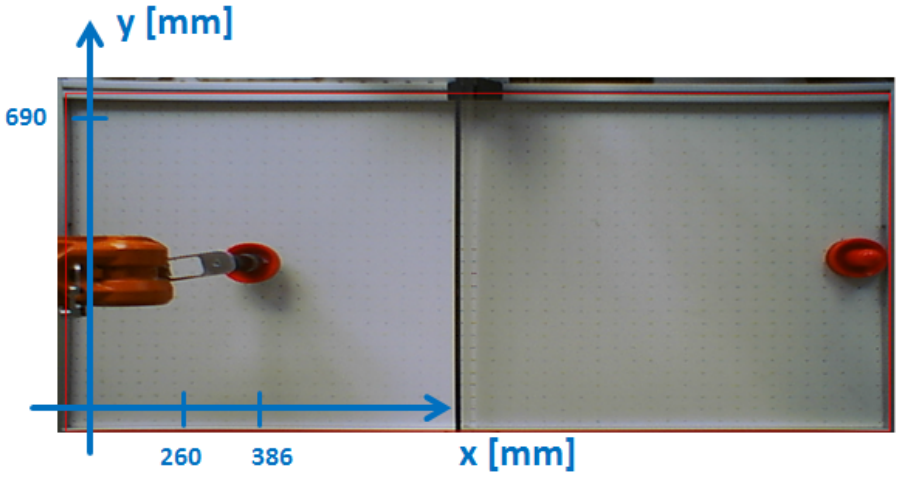

Abbildung 6.1: Koordinatensystem des Roboters

6.1 Versuchsdurchführung

Um die Bewegung des Roboters zu untersuchen haben wir das Python-Programm „Speed-Test.py“ geschrieben, welches sich in unserem GitHub-Repository zusammen mit den anderen Python-Dateien im Ordner „airhockey“ befindet.

Dieses besteht im wesentlichen aus der Funktion „versuch“ (siehe Listing 6.1. Diese hat drei Übergabeparameter: Der erste Übergabeparameter muss ein Objekt von der Klasse „RobotConnection“ sein. Die Definition dieser Klasse findet man in der Datei „Connection.py“. Der zweite Übergabewert ist die Startposition von der aus der Roboter

zur Endposition fahren soll, welche man mit dem dritten Übergabeparameter festlegen kann. Diese beiden Werte müssen als zweistelliges Tupel übergeben werden ([x-Koordinate, y-Koordinate]).

In der ersten While-Schleife wird solange versucht dem Roboter die Koordinaten des Startpunktes zu senden, bis dies erfolgreich war. Anschließend wird in der zweiten While-Schleife gewartet bis der Roboter seine Bewegung beendet hat und somit wieder neue Koordinaten entgegen nehmen kann. Ist dies der Fall, so wird zunächst die aktuelle Zeit in der Variable „start“ gespeichert. Danach werden die Koordinaten des Endpunktes gesendet und wieder gewartet bis der Roboter die Beendigung seiner Bewegung sendet. Dieser Zeitpunkt wird dann in der Variablen „end“ gespeichert. Zum Schluss werden die Messergebnisse noch protokolliert. Dazu werden der Startpunkt, der Endpunkt und die Zeit die zum Abfahren dieser Strecke benötigt wurde („end“ - „start“) in die Datei „speedtest.csv“ geschrieben.

```
1  def versuch(roboter, startpunkt, endpunkt):
2    while not roboter.SendKoordinatesToRoboter(startpunkt):
3      pass
4
5    while not roboter.canMove():
6      pass
7
8    start = time.time()
9
10   while not roboter.SendKoordinatesToRoboter(endpunkt):
11     pass
12
13   while not roboter.canMove():
14     pass
15
16   end =time.time()
17
18   protokoll.write(str(startpunkt[0]) + ";" + str(startpunkt[1]) + ";" +
       str(endpunkt[0]) + ";" + str(endpunkt[1]) + ";" + str(end - start)
        + "\n")
```

Listing 6.1: Python-Funktion für die Zeitmessung

Um in einem Durchlauf mehrere Strecken abfahren zu können, kann die Funktion „versuch“ dann beispielsweise in einer For-Schleife mehrmals aufgerufen werden. In Listing 6.2 wird zum Beispiel immer ausgehend vom Ursprung ([0,0]) in einem Abstand von zehn Millimetern alle Punkte der Grundlinie (x=0) abgefahren.

```
for y in range(0, 691, 10):
  versuch(roboter, [0, 0], [0, y])
```

Listing 6.2: Abfahren der Grundlinie

6.2 Versuchsauswertung

Dadurch dass die Messwerte als „Comma-Separated Values“ gespeichert werden, ist es einfach möglich diese Csv-Datei in einem Tabellenkalkulationsprogramm wie beispielsweise Excel zu öffnen.

Zur Auswertung wurde zunächst der Messtabelle eine weitere Spalte mit der jeweils zurückgelegten Strecke hinzugefügt. Diese wurde mit folgender Formel berechnet:

$$\Delta s = \sqrt{(x_{end} - x_{Start})^2 + (y_{end} - y_{Start})^2} \tag{6.1}$$

Die zurückgelegt Strecke wurde dann über der dafür jeweils benötigten Zeit in einem Diagramm eingetragen. Auf diese Weise wird einerseits der Zusammenhang der beiden Größen anschaulich dargestellt. Zum anderen ist es auch möglich eine Trendlinie durch die Punkte zu legen und sich deren Gleichung angeben zu lassen. Auf diese Weise gelangt man zu einer mathematischen Beschreibung, die den Zusammenhang zwischen Weg und Zeit approximiert. Wie gut eine solche Näherung ist, kann durch das Bestimmtheitsmaß (R^2) angegeben werden. Dabei handelt es sich um ein Maß aus der Statistik für den erklärten Anteil der Varianz einer abhängigen Variablen durch ein statistisches Modell. [6]

Auf diese Weise wurden nun unterschiedliche Bewegungen des Roboters ausgewertet. Dabei zeigte sich, dass sich die Bewegungen des Roboters in den meisten Fällen sehr gut mit einer linearen Trendlinie beschreiben lassen. So konnten die Bewegungen entlang der y-Achse (siehe 6.2) und entlang der x-Achse (siehe 6.3) durch eine lineare Trendlinie angenähert werden, deren Bestimmtheitsmaß über 99% liegt.

Bei Diagonal-Bewegungen fiel auf, dass es sich dabei zwar auch um einen linearen Zusammenhang zwischen Weg und Zeit handelt (siehe 6.4), dieser sich jedoch nicht ganz so genau mathematisch darstellen lässt ($R^2 < 0,99$). Des weiteren fiel auf, dass sich die Steigungen und y-Achsenabschnitte der Geraden-Gleichung sehr stark unterscheiden, je nachdem wo man den Startpunkt wählt und wie man die x- bzw. y- Koordinaten der Endpunkte bei einem Durchlauf variiert.

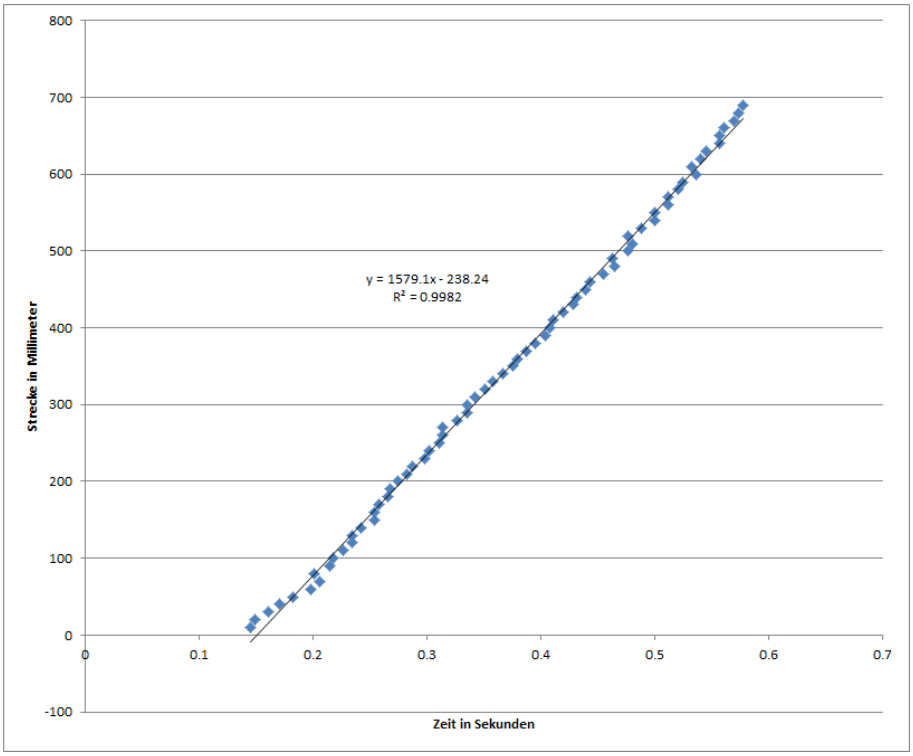

Abbildung 6.2: Bewegung entlang der y-Achse auf der Grundlinie

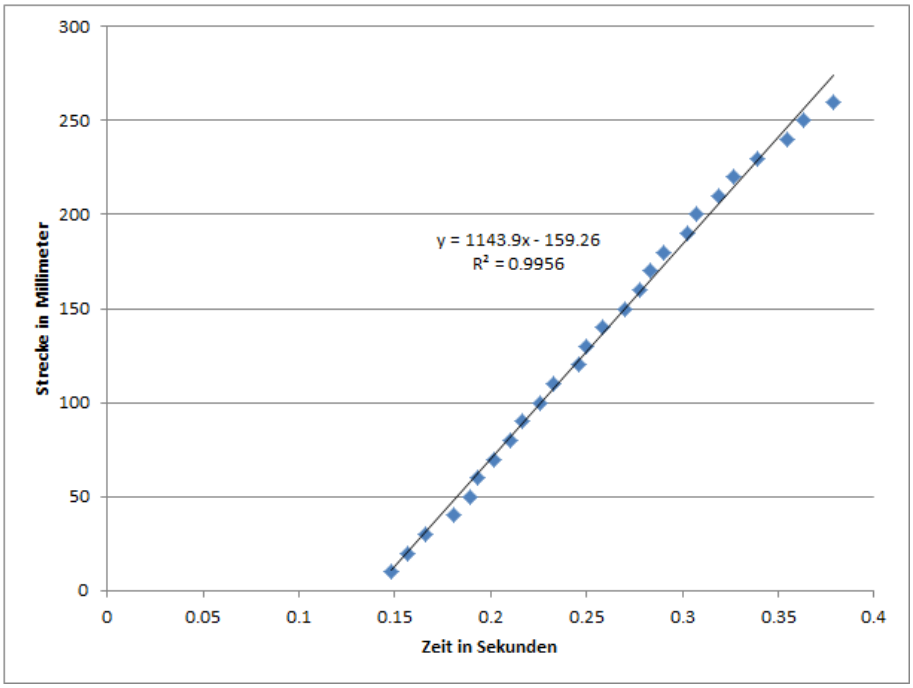

Abbildung 6.3: Bewegung entlang der x-Achse bei y = 570

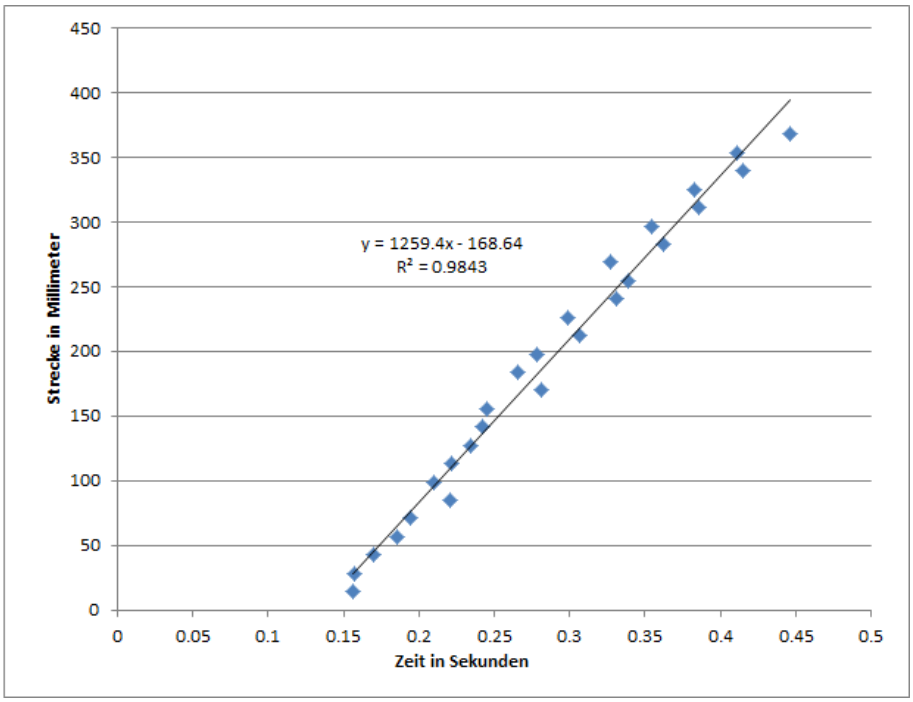

Abbildung 6.4: Diagonalbewegung

Wir haben uns deshalb dazu entschlossen, dass wir den Schläger nur auf festgelegten Bahnen fahren lassen. So wird eine Diagonal-Bewegung aufgeteilt in zwei Bewegungen: Zunächst eine Bewegung entlang der y-Achse auf der Grundlinie und anschließend eine Bewegung entlang der x-Achse.

Zur Berechnung der Zeit, die der Roboter benötigt, um eine gewisse Strecke Δy auf der Grundlinie zurückzulegen, kann die Gleichung der linearen Trendlinie verwendet werden (siehe Abb. 6.2). Diese muss nur etwas umgestellt werden und man erhält folgende Gleichung zur Berechnung von t_y bei gegebenem Δy:

$$t_y = \frac{\Delta y + 238,4}{1579,1} \tag{6.2}$$

Die Berechnung der Zeit, die der Roboter benötigt um eine Strecke entlang der x-Achse zurückzulegen, gestaltet sich etwas schwieriger. Es hat sich nämlich gezeigt, dass Steigung und y-Achsenabschnitt der unterschiedlichen Geradengleichungen abhängig von der y-Koordinate sind, an der die Bewegung ausgeführt wird. Aus diesem Grund wurde eine Messung durchgeführt, bei der für 25 verschiedene y-Koordinaten die Bewegung entlang der x-Achse untersucht wurde. Dabei wurde für jede y-Koordinate jeweils die Gleichung der linearen Trendlinie bestimmt. Daraufhin wurden die Steigungen und y-Achsenabschnitte dieser Geradengleichungen über ihren entsprechenden y-Koordinaten in Diagramme eingezeichnet (siehe Abb. 6.5 und Abb. 6.6). Dabei zeigte sich, dass man die Abhängigkeiten dieser beiden Größen zur y-Koordinate jeweils durch ein Polynom zweiten Grades annähern

kann. Mit den entsprechenden Trendlinien-Gleichungen kann dann die Steigung und der y-Achsenabschnitts bei vorgegebener y-Koordinate bestimmt werden.

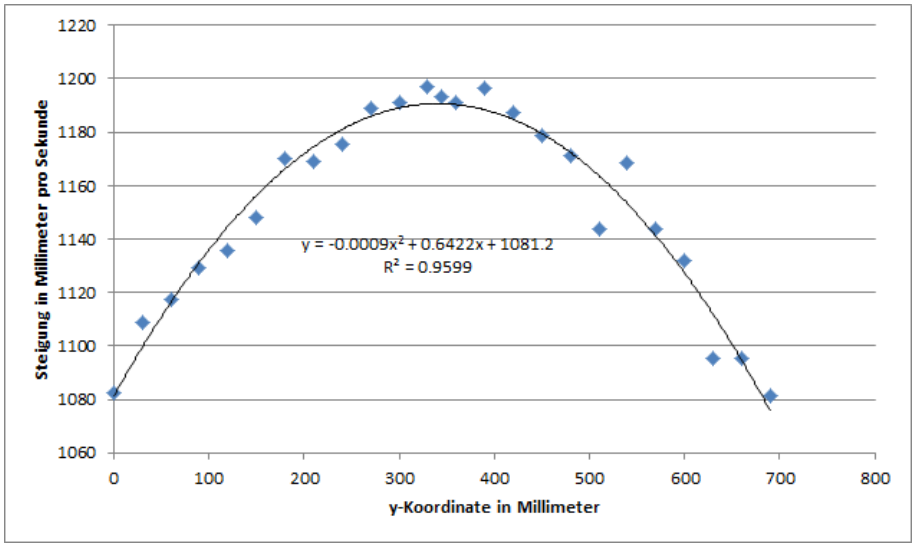

Abbildung 6.5: Zusammenhang zwischen y-Koordinate und Steigung der Trendlinie

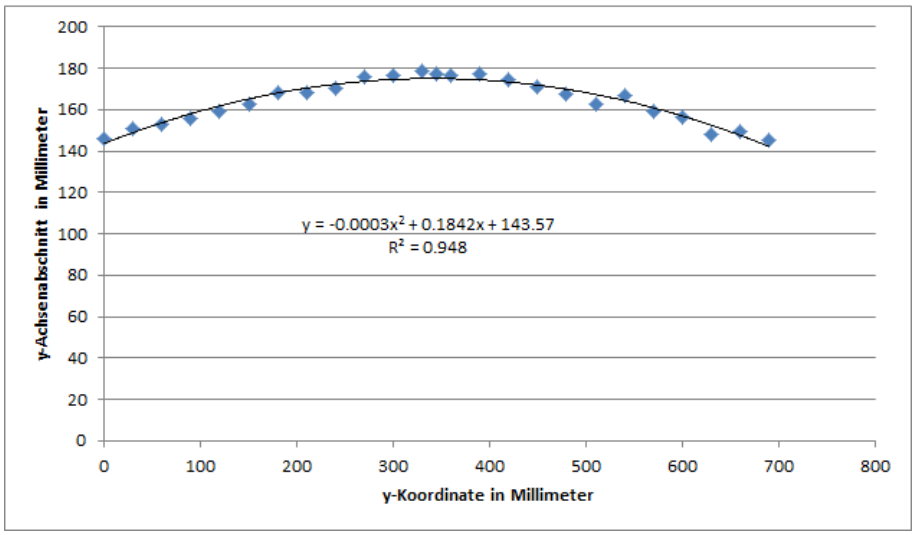

Abbildung 6.6: Zusammenhang zwischen y-Koordinate und y-Achsenabschnitt der Trendlinie

Beim Airhockey spielen fiel auf, dass der Puck häufig in eine Pendelbewegung zwischen den beiden langen Banden verfällt. Da die Reichweite des Roboter in y-Richtung jedoch nur 260 mm beträgt, konnte er den Puck oft nicht erreichen und man musste um weiterspielen zu können entweder lange warten, bis der Puck von alleine wieder in die Spielerhälfte glitt, oder man musste den Puck mit einem Zollstock anstupsen.

Da die Reichweite des Roboter in der Mitte des Spielfeldes (bei y = 345) um einiges höher ist als an den Rändern, nämlich 386 mm, wurde eine Strategie ergänzt um einen besseren Spielfluss zu erreichen. Diese sieht vor, dass sobald eine Pendelbewegung vorliegt der Schläger in die Mitte gefahren wird. Von da aus kann er sich dann weiter nach vorne

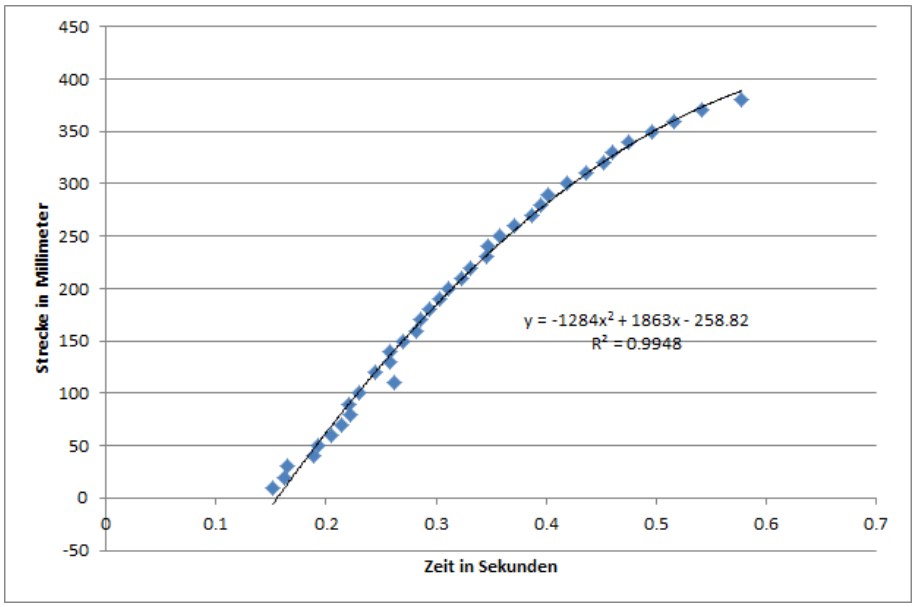

Abbildung 6.7: Bewegung entlang der x-Achse bei y = 345

bewegen als sonst. Bei der Untersuchung der Bewegung in x-Richtung an dieser Stelle (y=345) fiel auf, dass sich die Bewegung für diese längere Strecke nun nicht mehr gut genug durch eine lineare Trendlinie approximieren lässt (siehe Abb. 6.7). Vielmehr musste man hier ein Polynom zweiten Grades verwenden. Um an dieser Stellen nun für eine gegebene Strecke Δy die entsprechende Zeit t_y berechnen zu können, musste die Gleichung der Trendlinie noch entsprechend umgeformt werden. Dabei ergab sich folgende Gleichung:

$$t_y = \frac{-1863 + \sqrt{1893^2 - 4 * (-1284) * (-258,82 - \Delta y)}}{2 * (-1284)} \tag{6.3}$$

6.3 Schlussfolgerungen

Ohne diese Untersuchung der Roboterbewegung wäre es nicht möglich gewesen einen Angriffsschlag zu implementieren. Die Approximation die hierbei erarbeitet wurden, sind in sofern ausreichend, dass der Roboter meistens den Puck trifft sobald er einen Angriffsschlag durchführt.

Die Beschränkung der Roboterbewegungen auf Bewegungen entlang der x- und y-Achse ist jedoch sehr stark und lässt kaum ausgefeiltere Strategien zu. Außerdem fiel auf, dass der Roboter den Puck bei einer Pendelbewegung häufig verfehlt, obwohl extra für diesen Fall eine eigene Funktion zur Berechnung der benötigten Zeit erarbeitet wurde.

Zur Behebung dieser beiden Probleme könnten man anstatt kontinuierliche Werte für die Koordinaten zu verwenden nur diskrete Werte zulassen. Zum Beispiel für die x-Koordinate

nur die Werte 0, 30, 60, ... , 690 und für die y-Koordinate nur 0, 26, 52, ... , 260. Man könnte dann mit dem Schläger 254 verschieden Positionen anfahren. Misst man anschließend von jeder Position aus wie lange man benötigt um von hier aus zu allen anderen Positionen zu gelangen, könnten man die entsprechenden Zeiten in eine Lookup-Table eintragen. Mit einer solchen Implementierung wären dann auch Diagonal-Bewegungen ohne weiteres möglich. Die Erzeugung einer solchen Lookup-Table ist jedoch extrem aufwendig. Es wäre deshalb ratsam zunächst zu testen, ob man dadurch merkliche Verbesserungen erzielen kann. Anbieten würde sich dafür die Bewegung in x-Richtung in der Mitte des Tisches, da dafür sowieso schon einen eigene Funktion verwendet wird und man dabei eine Verbesserung auch deutlich merken sollte.

7 Implementierung der Spielstrategie

7.1 Spielstrategien

7.1.1 Defensive Spielstrategie

Bei der Defensiven Spielstrategie wird der Schläger nur auf einer geraden Linie parallel zum Tor bewegt. Bei dieser Strategie kann man grundsätzlich von 2 Situationen ausgehen. Die erste ist, der Puck bewegt sich auf den Roboter zu. In so einer Situation versucht der Roboter den Puck abzuwehren, erstmal ungeachtet ob der Puck sich überhaupt aufs Tor zu bewegt oder daran vorbeigehen würde. Bei der zweiten Situation bewegt sich der Puck vom Roboter weg. In diesem Fall sollte sich der Roboter zwischen dem Tor des Roboters und dem Puck positionieren und dem nächsten Angriff des Spielers zuvorzukommen.

Bei einer erweiterten Defensivstrategie könnte zum Beispiel zusätzlich überprüft werden ob der Puck überhaupt das Tor trifft oder in die Nähe des Tores kommt um unnötige Bewegungen zu vermeiden.

7.1.2 offensive Spielstrategie

Bei der Offensiven Strategie versucht der Roboter immer den Puck sofort zurückzuschlagen ohne an eine Verteidigung zu denken, frei nach dem Motto „Angriff ist die beste Verteidigung“. Dabei sollte angefangen werden, den Puck nur zurückzuschlagen ungeachtet der Richtung in welche dieser geschlagen wird. Sollte diese Taktik funktionieren, kann man dann ergänzen, dem Puck beim Zurückschlagen eine bestimme Richtung zu geben, damit der Roboter gezielter das gegnerische Tor trifft.

Später kann diese Strategie so erweitert werden, dass sich der Roboter nicht direkt auf den Puck zu bewegt, sondern sich vorher noch positioniert um eine bessere Angriffsposition zu haben.

7.1.3 Defensive Spielstrategie mit offensiven Ansätzen

Es wird weiterhin die selbe Taktik wie bei der rein defensiven Strategie genutzt. Zusätzlich wird ergänzt, das nach dem Blocken des Puckes, dieser mit der offensiven Strategie zurückgeschlagen wird.

7.1.4 ausbalancierte Spielstrategie

Bei der ausbalancierten Spielstrategie wird selbstständig vom Roboter entschieden, ob die Defensive oder die offensive Strategie verwendet wird. Bei der defensiven Strategie sollte die Verteidigungslinie auf einen Bereich unmittelbar vor dem Tor reduziert werden um unnötige Verteidigungsaktionen zu vermeiden.

7.2 Strategieimplementierung

Bei unserer Spielstrategie wird eine ausbalancierte Strategie verwendet. Um diese Strategie ausführen zu können, wurden vier unterschiedliche Befehle eingeführt. Diese kann man grundsätzlich in 2 Kategorien unterteilen, in defensive und offensive Befehle.

Bei vieler dieser Funktionen und auch der Entscheidung wird eine Funktion zur Schnittpunktberechnung benutzt. Deshalb wird diese jetzt gleich vorab erklärt wie sie funktioniert. Die Funktion für die Schnittpunktberechnung ist im Quellcode 7.1 auf Seite 24 zu sehen. Hier wird nur die Schnittpunkberechnung für eine X-Achse erläutert. Dieselbe Methode kann aber leicht umgeändert auch für Schnittpunkte mit einer Y-Achse verwendet werden. Dieser Funktion können 2-4 Parameter übergeben werden. Der erste Parameter ist der *bag*. Der *bag* enthält die Daten der Bilderkennung für die Puckposition, dessen Bewegungsrichtung und Geschwindigkeit. Der zweite Parameter ist xLine. Dies ist ein Wert zwischen 0 und 1, der den X-Wert der zur X-Achse parallelen Linie im Bildkoordinatensystem enthält. Der dritte und vierte Parameter sind optional. Werden diese beiden Parameter übergeben, werden sie für die Schnittpunkberechnung als Ausgangswerte genutzt anstatt der Werte aus dem *bag*.

```
1 def CrossXLine(self,bag, xLine, startposition = None, direction = None):
2   if startposition == None:
3     startposition = bag.puck.position
4   if direction == None:
5     direction = bag.puck.direction
6   if direction[0] == 0:
7     return None
```

```
8   timeToCrossXLine = (xLine - startposition[0])/(direction[0] * bag.puck
        .velocity)
9   yPosition = startposition[1] + timeToCrossXLine * direction[1] * bag.
        puck.velocity
10
11  #Wenn der Puck ausserhalb der Spielfeldbegrenzung liegt, wird dieser
        vor dem zurueckgeben noch an der Spielfeldgranze gespiegelt
12  koordinates = vector.mirror_point_into_field([xLine, yPosition])
13  return (koordinates[0], koordinates[1], timeToCrossXLine)
```

Listing 7.1: Python-Funktion für Schnittpunktberechnung

Man kann mithilfe eines Punktes und eines Bewegungsvektors eine Gerade darstellen. Die Geradengleichung im 2-Dimensionalen Raum sieht in dann so aus:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_P \\ y_P \end{pmatrix} + k * \begin{pmatrix} x_R \\ y_R \end{pmatrix}.$$

X und Y sind dabei die Koordinaten jeden beliebigen Punktes auf der Geraden. X_P und Y_P sind die Koordinaten eines beliebigen Punktes auf dieser Geraden. In unserem Fall ist dies die Position des Puckes. X_R und Y_R sind Teile des Richtungsvektors. Der Skalar k ist eine reelle Zahl, die den Richtungsvektor streckt oder staucht, um jeden beliebigen Punkt auf der Geraden zu erreichen. Im Bild 7.1 auf Seite 25 ist dies nochmal grafisch dargestellt worden. Wir haben die Position des Puckes und die Bewegungsrichtung des Puckes. Wenn wir nun den Bewegungsvektor des Puckes entsprechend strecken oder stauchen, schneiden wir die gewünschte X-Achse. Wir könnten nun für die gewünschte X-Achse auch eine Geradengleichung aufstellen und dann die beiden Gleichungen in einem Gleichungssystem gleichsetzen, aber da wir bei der X-Achse einen festen X-Wert haben, überspringen wir diesen Teil einfach. Wir haben also nun folgende Gleichung, die es zu lösen gilt:

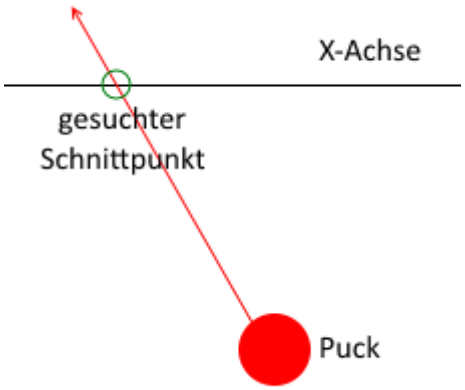

Abbildung 7.1: Veranschaulichung der Schnittpunktberechnung

$$\begin{pmatrix} xLine \\ y_S \end{pmatrix} = \begin{pmatrix} x_P \\ y_P \end{pmatrix} + k * \begin{pmatrix} x_R \\ y_R \end{pmatrix}.$$

Dies kann man nun in einem Gleichungssystem folgendermaßen dargestellt werden:

$$\begin{aligned} xLine &= x_P + k * x_R \\ y_S &= y_P + k * y_R \end{aligned}$$

Wir haben somit die die Y-Koordinate des Schnittpunktes Y_S und den Skalar k als unbekannte Variablen. Die X-Koordinate des Schnittpunktes *xLine* wird als Variable an die Funktion übergeben und ist dadurch bekannt. Die erste Gleichung enthält damit nur die unbekannte k und kann nach k umgestellt werden. Wir erhalten damit die Gleichung:

$$k = \frac{xLine - x_P}{x_R}$$

Da in dieser Gleichung eine Division stattfindet, muss überprüft werden, ob der X-Anteil des Bewegungsvektors X_R ungleich null ist. Dies geschieht im Python Code in Zeile 6 und 7. Direction[0] ist dabei dieses X_R. Dies wird so dargestellt, da Vektoren in Python als Array behandelt werden. Vektoren sind somit 1-Dimensionale Array mit in unserem Fall 2 Werten. Der erste Wert, also direction[0] ist der X-Anteil und der zweite Wert direction[1] ist der Y-Anteil. Der Richtungsvektor der sich in dem Programm hinter bag.puck.direction verbirgt ist ein Einheitsvektor, das heißt dass dieser Vektor immer die Länge 1 hat. Deshalb multiplizieren wir zum Richtungsvektor in unserem Code noch die Geschwindigkeit auf. Dadurch erhalten wir einen Richtungsvektor, den der Puck in einer Sekunde beschreitet. Somit kann das errechnete k, oder wie wir es in unserem Code bezeichnen *timeToCrossXLine*, auch für die benötigte Zeit genutzt werden, die später für die Angriffsstrategien wichtig ist. Anhand dieses Skalares kann man auch erkennen ob sich der Puck auf den Schnittpunkt zubewegt oder weg bewegt. Denn wenn $k > 0$, dann bewegt sich der Puck auf die Schnittachse zu, ansonsten von ihr weg.

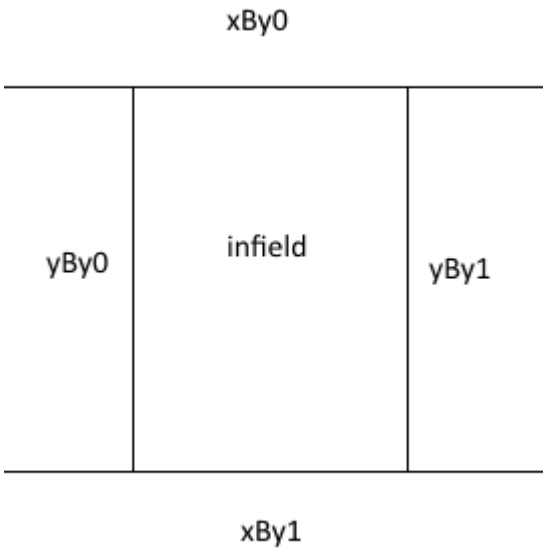

Abbildung 7.2: Quadranten bei der Spiegelung

Mit dem errechneten k, kann man nun auch die 2. Gleichung ausrechnen um die Y-Koordinate des Schnittpunktes zu erhalten. Dieser errechnete Wert kann aber unter Umständen außerhalb des Spielfeldes liegen und muss deshalb noch an den Banden in das Spielfeld gespiegelt werden. Dies übernimmt die Funktion *mirror_point_into_field*. Zu sehen im Quellcode 7.2 auf Seite 27. Diese Funktion nutzt dafür 2 andere Funktionen. Zum ersten die Funktion *check_for_out_of_field* um den Quadranten zu berechnen in dem sich der Puck befindet. Dabei werden 5 Quadranten unterschieden, wie im Bild 7.2 auf Seite 26 zu sehen ist. Der Quadrant 0, in unserem Quellcode *infield* genannt, bedeutet, dass sich der Puck innerhalb des Spielfeldes befindet. Quadrant 1 und 2 bedeutet das sich der Puck in X-Richtung außerhalb des Spielfeldes befindet. Wir haben diese Quadranten *xBy0* und *xBy1*. Punkte in diesem Quadranten müssen an den X-Achsen gespiegelt werden. Quadrant 3 und 4 bedeutet, dass sich der Puck in Y-Richtung außerhalb des Spielfeldes befindet.

Diese wurden respektabel *yBy0* und *yBy1* genannt. Diese Punkte werden an den Grenzen der Y-Achsen gespiegelt. Die unterschiedliche Größe der Quadranten kommt durch die Reihenfolge der Berechnungen zu Stande. Die Spiegelung erfolgt im Bildkoordinatensystem bei dem die Koordinaten von (0/0) bis (1/1) gehen. Die Quadranten wurden deshalb nach der Spielfeldgrenzen benannt, an denen sie angrenzen und die Punkte in diesen Quadranten gespiegelt werden müssen. Die zweite Funktion, die genutzt wird, ist die *mirror_point_at_border* Funktion. Dieser Funktion wird der zuvor berechnete Quadrant übergeben und berechnet dann den neuen gespiegelten Punkt. Um den neuen Punkt zu berechnen wird der Abstand des Punktes zur nahesten Spielfeldgrenze errechnet und dieser Abstand wird dann von der Spielfeldgrenze subtrahiert. Dies wird solange in einer Schleife wiederholt bis der Punkt im Spielfeld liegt.

```
1  def mirror_point_into_field(point):
2      bordervalue = check_for_out_of_field(point)
3      while bordervalue > 0:
4      point = mirror_point_at_border(point, bordervalue)
5      bordervalue = check_for_out_of_field(point)
6      return point
7
8  def mirror_point_at_border(point, bordervalue):
9      yPosition = point[1]
10     xPosition = point[0]
11     if bordervalue == border.border.xBy0:
12       xPosition = const.CONST.xBorderBy0 - (xPosition - const.CONST.
             xBorderBy0)
13     if bordervalue == border.border.xBy1:
14     xPosition = const.CONST.xBorderBy1 - (xPosition - const.Const.
           xBorderBy1)
15     if bordervalue == border.border.yBy0:
16       yPosition = const.CONST.yBorderBy0 -(yPosition - const.CONST.
             yBorderBy0)
17     if bordervalue == border.border.yBy1:
18       yPosition = const.CONST.yBorderBy1 -(yPosition - const.CONST.
             yBorderBy1)
19     return (xPosition, yPosition)
20
21
22 def check_for_out_of_field(point):
23   if point[0] < const.CONST.xBorderBy0:
24     return border.border.xBy0
25   if point[0] > const.CONST.xBorderBy1:
26     return border.border.xBy1
27   if point[1] < const.CONST.yBorderBy0:
28     return border.border.yBy0
29   if point[1] > const.CONST.yBorderBy1:
30     return border.border.yBy1
```

```
31    return border.border.infield
```

Listing 7.2: Python-Funktion für Punktspiegelung

7.2.1 defensive Strategien

Der einfachste dieser Befehle ist die „*Fahre die Home-Position an*“. Bei diesem Befehl platziert sich der Roboter direkt mittig vors Tor. In dieser Position ist es nur noch schwer möglich das Tor zu treffen, aber immer noch machbar.

Der zweite Befehl ist der *Verteidige* Befehl. Bei diesem Befehl bewegt sich der Roboter vor dem Tor auf die Bewegungsbahn des Puckes, damit dieser nicht in das Tor gelangt. Der Python Code für diesen Befehl sieht man in 7.3 auf Seite 28.

```
1  def defend(self, bag):
2    koordinates = self.CrossXLine(bag, 0.0)
3    robKoordinates = self.transformToRobotkoordinates(koordinates)
4    #Position auf unmittelbar vors Tor beschraenken
5    yMin = (const.CONST.RobYMax - const.CONST.Torgroesse) / 2.0
6    yMax = (const.CONST.RobYMax + const.CONST.Torgroesse) / 2.0
7    if robKoordinates[1] < yMin or robKoordinates[1] > yMax:
8      return None
9    if abs(self.oldRobotKoordinates[1] - robKoordinates[1]) < const.CONST.
         minimumMovement:
10     return None
11   self.lastMove = self.DEFEND
12   #return explizit 0, da CrossXLine die 0 im Bildkoordinatensystem
         verwendet, die eine andere 0 als im Roboterkoordinatensystem ist
13   return (0,robKoordinates[1])
```

Listing 7.3: python-funktion für Defensivstrategie

Dazu wird zuerst der Schnittpunkt mit der Nulllinie der X-Achse berechnet. Diese liegt direkt an der Bande des Spielfeldes. Damit diese auch für den Roboter genutzt werden kann müssen die errechneten Koordinaten noch vom Bildkoordinatensystem in das Roboterkoordinatensystem umgerechnet werden. Um die Verteidigungsbewegung auch wirklich auszuführen, müssen noch 2 zusätzliche Kriterien erfüllt werden. Das erste Kriterium ist, das der Puck sich auch wirklich auf das Tor zu bewegt und nicht irgendwo neben dem Tor gegen die Bande prallt. Dies wird im Code in den Zeilen 5-8 entschieden indem zuerst die Torbegrenzungen *yMin* und *yMax* berechnet werden. sollte der errechnete Schnittpunkt außerhalb liegen, wird die Funktion abgebrochen. Sollte sich der Schläger bereits in einem gewissen Toleranzbereich vom errechneten Punkt befinden, soll sich der Roboter auch nicht erneut bewegen, da dieser dann für eine kurze Zeit nicht mehr reagieren kann.

Dieser Toleranzbereich ist mit der Konstante *minimumMovement* festgelegt und beträgt 25 mm. Sollten diese Fälle nicht eintreffen, merkt sich das Programm das es zuletzt diese Funktion ausgeführt hat und gibt die Koordinaten zu denen sich der Roboter bewegen soll zurück. Dabei wird für die X-Koordinate explizit 0 verwendet, da die X-Koordinate von der CrossXLine Funktion im Bildkoordinatensystem 0 ist, was nicht ganz exakt 0 im Roboterkoordinatensystem ist. Ansonsten gäbe es eine Verschiebung von etwa 3mm, die immer recht merkwürdig aussah wenn sie auftrat.

7.2.2 offensive Strategien

Es gibt 2 offensive Strategien. Die Strategie *attack1* ist dabei die Vorbereitung auf den Angriff und *attack2* ist Ausführung des Angriffes. Bei der Vorbereitung zum Angriff bewegt sich der Puck auf der Nulllinie der X-Achse, so dass dieser dann nur noch eine Vorwärtsbewegung für den Angriff machen muss. Der Angriff ist dann quasi die Vorwärtsbewegung um den Puck zurückzuschlagen. Den Sourcecode zu beiden Funktionen sieht man im Quellcode 7.4 auf Seite 31. Der Funktion *attack1* werden direkt die Koordinaten des errechneten Schnittpunktes übergeben. Es wird nur überprüft ob diese Koordinaten zu nah an den Außenbanden liegen um zu verhindern, dass der Roboter den Puck zwischen Bande und Schläger einquetscht. Sollte dies nicht der Fall sein, merkt sich das Programm das *attack1* zuletzt ausgeführt wurde und gibt die Koordinaten zu denen sich der Roboter bewegen soll zurück. Bei *attack2* muss als erstes die aktuelle Position des Roboters vom Roboterkoordinatensystem ins Bildkoordinatensystem umgerechnet werden, da diese für die Schnittpunktberechnung benötigt wird. Für die Schnittpunktberechnung werden bei dieser Funktion je nach Situation unterschiedliche Schnittachsen verwendet. Sollte sich der Puck in einem spitzen Winkel von etwa 18° der Y-Achse nähern, wird der Schnittpunkt zur X-Achse etwa 20cm vor dem Tor berechnet. Sollte dieser Fall eintreffen, wird noch zusätzlich überprüft ob der Schlag nicht zu schräg wird, da dann unsere Berechnung zur benötigten Zeit des Schlages nicht mehr stimmt und um wieder zu verhindern. dass der Puck zwischen Bande und Schläger eingequetscht wird. Dieser Fall sollte zwar nicht eintreffen, da dieser vorher bereits abgefangen wird, ist aber aus Sicherheitsgründen hier immer noch enthalten. Sollte der Winkel zur Y-Achse größer sein, wird der Schnittpunkt mit der Y-Achse berechnet, wobei dann der Y-Wert der aktuellen Roboterposition verwendet wird. Dies wird so gelöst, da die Berechnungen mit der Y-Achse als Schnittachse genauer sind, da hier nur eine senkrechte Vorwärtsbewegung ausgeführt werden muss, anstatt einer leicht schrägen. Allerdings erhält man bei Spitzen Winkeln zur Y-Achse häufig nicht nutzbare Werte dadurch. Sobald man den Schnittpunkt hat, wird noch überprüft ob dieser vom Roboter überhaupt erreicht werden kann. Dies sollte nur auftreten wenn der Puck während einer Pendelbewegung angegriffen wird und der Puck zu weit vom Roboter weg ist. Dies

wird im Kapitel 7.3 auf Seite 31 etwas genauer erklärt. Wenn der Roboter den errechneten Schnittpunkt in seiner Bewegungszeit mit Abweichung einer Framezeit erreichen kann, wird der Angriffsschlag ausgeführt, ansonsten wird gewartet bis er den Schnittpunkt in der errechneten Zeit erreichen kann.

```
def attack1(self, robKoordinates):
  if robKoordinates[1] < const.CONST.durchmesserPuck:
    return None
  if robKoordinates[1] > const.CONST.RobYMax - const.CONST.
      durchmesserPuck:
    return None
  self.lastMove = self.ANGRIFF1
  return (0, robKoordinates[1])

def attack2(self, bag):
  #alte Roboterkoordinaten ins Bildsystem umrechnen
  y = (self.oldRobotKoordinates[1] + const.CONST.durchmesserSchlaeger /
      2.0) / (const.CONST.tableWidth +  const.CONST.durchmesserSchlaeger
      )
  # sollte der Puck sich dem Schlaeger in einem Winkel kleiner 18 Grad
      naehern,
  # wird der schnittpunkt mit der x - Achse verwendet ansonsten der
      Schnittpunkt mit der y-Achse
  if bag.puck.direction[0] < -0.95:
    koordinates = self.CrossXLine(bag, 0.18)
    # Wenn der Schlag zu schraeg wird, Angriff abbrechen
    if abs(koordinates[1] - y) > 0.2:
      print("Angriff 2 abgebrochen")
      return None
  else:
    koordinates = self.CrossYLine(bag, y)
  robKoordinates = self.transformToRobotkoordinates(koordinates)
  if not self.roboter.robCanReachPoint(robKoordinates):
    return None
  if abs(self.calculateTimeToXPoint(robKoordinates) - koordinates[2]) <
      1.0 / const.CONST.FPS:
    self.lastMove = self.ANGRIFF2
    return robKoordinates
  return None
```

Listing 7.4: python-funktion für Offensivstrategien

7.3 Strategieentscheidung

Für die Auswahl der richtigen Strategie haben wir einen komplexen Entscheidungsbaum entworfen, der auf verschiedene Parameter zurückgreift. Zu diesen Parametern zählen die aktuelle Position, die Bewegungsrichtung und Bewegungsgeschwindigkeit des Puckes, die zuletzt befahrene Position des Roboters und die zuletzt ausgeführte Aktion des Roboters.

Im Bild 7.3 auf Seite 33 sehen sie das Flussdiagramm dieses Entscheidungsbaumes. Wenn der Roboter als letzte Bewegung einen Angriff ausgeführt hat um den Puck zurück zu schlagen, steht in den meisten Fällen das Tor komplett frei. Um einen Konter entgegenzukommen, bewegt sich der Roboter anschließend sofort vors Tor. Denn so wird der Weg zum Verteidigen erheblich verkürzt. Wenn sich der Puck in der gegnerischen Hälfte befindet, werden nur defensive Strategien in Betracht gezogen, da sich der Roboter bei offensiven Strategien vom Tor entfernt und der Gegner die Bahn des Puckes immer noch beeinflussen kann. Sollte sich der Puck in der gegnerischen Hälfte vom Roboter wegbewegen, bewegt sich der Roboter zurück in die Homeposition um sich alle Möglichkeiten offen zu lassen, da der Puck vom Gegenspieler auf jeden Fall noch beeinflusst wird. Bewegt sich der Puck stattdessen schon auf den Roboter zu, soll dieser die *Defend* Funktion aufrufen um einen möglichen Angriff bereits entgegenzuwirken.

Beim nächsten Schritt wird ermittelt ob der Puck auf der Spielfeldseite des Roboters pendelt. Als Pendeln haben wir festgelegt, wenn sich der Puck in einem Winkel kleiner 20° entlang der Y-Achse bewegt. Dies ist der Fall, wenn der Betrag des normalisierten Bewegungsvektors in X-Richtung kleiner als 0,35 ist, denn $\arccos 0,35 \approx 70°$ in X-Richtung, also etwa 20∘ in Y-Richtung. Sollte sich der Puck in so einer Pendelbewegung befinden, fährt der Roboter mittig vors Tor oder wenn dies schon geschehen ist, macht der Roboter einen Angriffsschlag. Der Grund wieso der Roboter erst mittig vors Tor fährt ist folgender. Mit voll ausgestreckten Arm macht der Roboter eine Kreisförmige Bewegung, dadurch kann er unterschiedlich weit in das Spielfeld hineinreichen. Am Rand des Spielfeldes erreicht dieser eine Maximalreichweite von etwa 26cm, wohingegen er in der Mitte etwa 38cm schafft. Dadurch kann dieser fast 1,5x so weit reinreichen wie am Rande des Spielfeldes. Aber diese 38 cm sind immer noch nur etwa 55% seiner eigenen Spielfeldhälfte, wodurch dieser nicht immer den Puck erreichen kann und der Spieler warten muss, bis der Puck dann irgendwann mal auf seine eigene Hälfte zurück pendelt.

Sollten diese Entscheidungen nicht die richtigen gewesen sein, wird nun der Schnittpunkt mit einer X-Achse, die sich etwa 20cm vor dem Tor befindet berechnet. Wenn die letzte Bewegung eine Vorbereitung auf den Angriff war, also *attack1* ausgeführt wurde, wird überprüft ob die zuletzt berechnete Position nicht weiter als 2,5cm von der aktuell berechneten Position entfernt liegt. Diese 2,5cm sind die selbe Konstante *minimumMovement*, wie sie auch im Defendalghorithmus auf Seite 28 verwendet wird. Sollte dies der Fall sein, wird die Funktion *attack2* ausgeführt. Sollte als letztes nicht *attack1* ausgeführt worden sein oder die neu berechnete Position weicht zu sehr von der alten ab, wird überprüft ob der Roboter mit eine Bewegung in Y-Richtung und einer anschließenden Bewegung in X-Richtung den Puck erreichen kann. Sollte die benötigte Zeit nicht ausreichen, also der Puck kommt vor dem Schläger an, wird die *defend* Funktion ausgeführt. Sollte der Roboter aber genug Zeit haben, wird der Angriffsschlag mit der *attack1* Funktion vorbereitet.

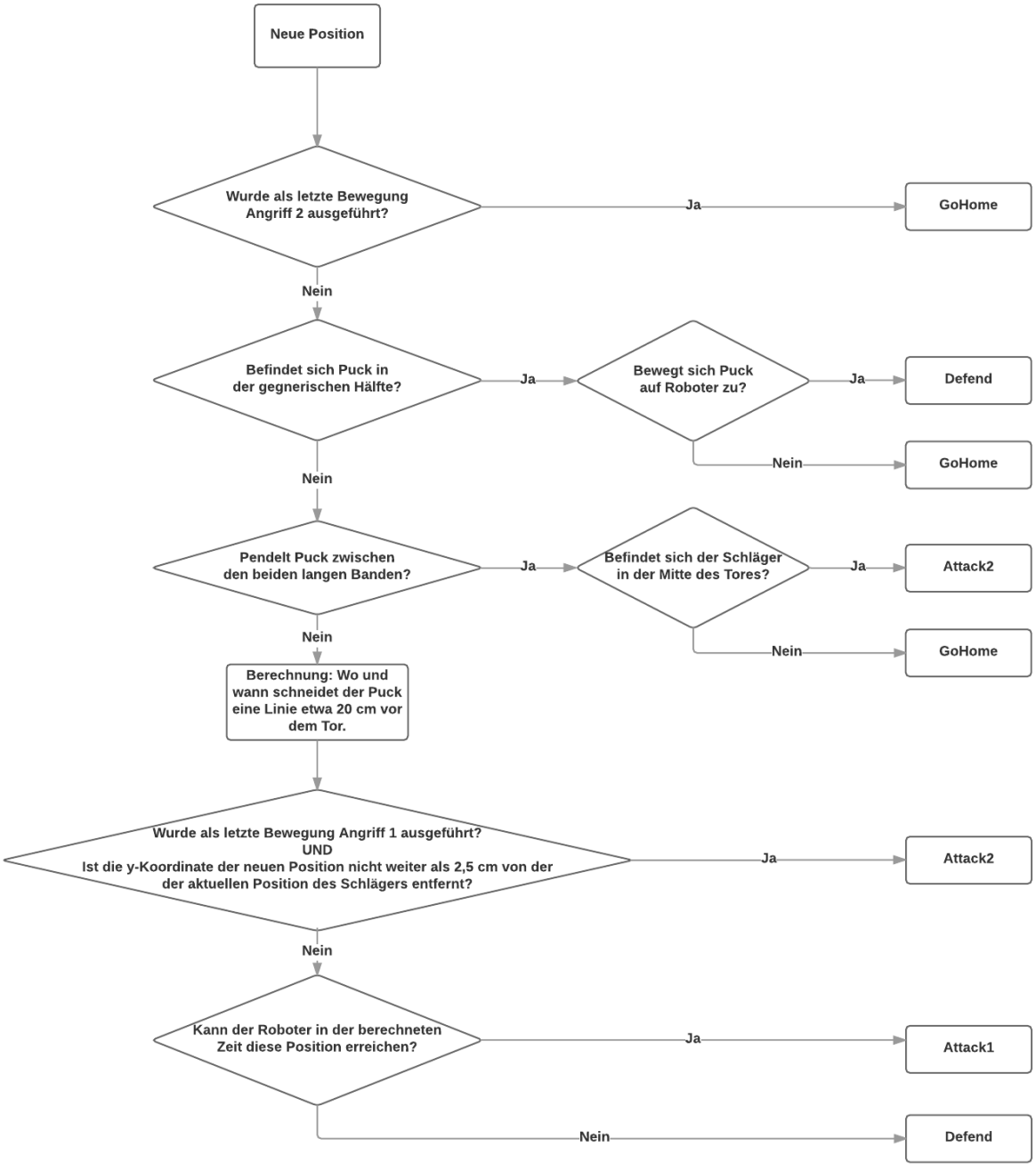

Abbildung 7.3: Entscheidungsbaum für die Strategieauswahl

8 Manual

In diesem Kapitel werden zusammenfassend alle Schritte erklärt die nötig sind um das Projekt von Grund auf in den Stand zu bringen, wie es gegen Ende unserer Studienarbeit war.

Sofern am Roboter und den beiden Rechnern nichts verändert wurde, können Sie die ersten beiden Abschnitte dieses Kapitels überspringen und direkt ein neues Spiel starten (siehe Abschnitt 8.3).

8.1 Einrichtung der Produktivumgebungen

Das Projekt besteht aus zwei Teilen. Zum einen Bilderkennung und -verarbeitung, welche durch ein Python-Programm realisiert wird. Dafür benötigt man einen PC, auf dem einen Linux-Distribution installiert ist. Des weiteren aus einem RAPID- Programm zur Steuerung des ABB-Roboters. Zum einfachen Starten und Editieren dieses Programms auf dem Controller braucht man einen Windows-PC auf dem die Software RobotStudio installiert ist.

Alle nötigen Dateien finden Sie im GitHub-Repositorie https://github.com/clush/Airhockey/.

8.1.1 Produktivumgebung für Bilderkennung einrichten

Alle nötigen Dateien für die Bilderkennung befinden sich im Ordner „aihockey“ in unserem GitHub-Repository. Um das Programm starten zu können, muss OpenCV und Python auf dem Rechner installiert sein. Für die Installation von OpenCV können Sie das Shell-Script „install-opencv.sh“ verwenden, welches ebenfalls in unserem GitHub-Repository zu finden ist. Dieses haben wir auf einem Debian-System getestet.

Sollte es wider erwarten zu Problemen kommen, finden Sie in der Studienarbeit „*Roboter lernt Air-Hockey spielen*“ von Sebastian Bezold, Christian Füller und Fabian Nagel in Abschnitt 3.2 eine ausführlichere Beschreibung zur Einrichtung der Produktivumgebung für die Bilderkennung.

8.1.2 RAPID-Programm auf Controller des Roboters laden

Melden Sie sich auf dem Windows-PC mit ihrem DHBW-Account an. Öffnen Sie anschließend RobotStudio. Nun müssen Sie zunächst unser RobotStudio-Projekt entpacken. Klicken Sie dazu in der linken Menüleiste auf „Share“ und dann auf „Unpack and Work“. Navigieren Sie anschließend zum Projektordner und wählen die Datei „Airhockey2.0.rspag“ aus. Legen Sie nun einen Speicherort für die das Projekt aus und merken sich diesen, damit es zu einem späteren Zeitpunkt von dort öffnen können. Nach erfolgreichem Entpacken öffnet sich die Station automatisch.

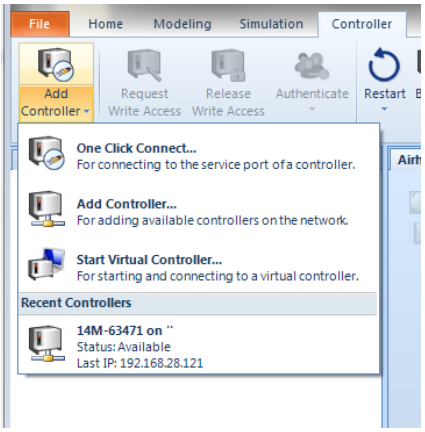

Abbildung 8.1: Controller hinzufügen

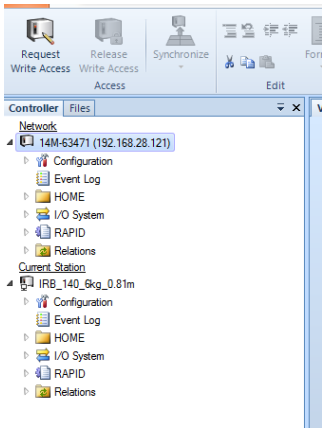

Abbildung 8.2: Verzeichnisstruktur

Als nächstes müssen Sie den Roboter einbinden. Stellen Sie zunächst sicher, dass er eingeschaltet ist. Gehen Sie dann unter den Reiter „Controller“ und dann auf „Add Controller“. Der Roboter-Controller hat die IP-Adresse 192.168.28.121 und sollte ganz unten im Drop-Down-Menü zu finden sein (siehe Abb. 8.1). Oft muss man um ihn auswählen zu können, das Menü nochmal einklappen und dann erneut auf „Add Controller“ klicken.

Nach erfolgreichem Einbinden können Sie den Roboter in der linken Verzeichnisstruktur finden (vgl. Abb. 8.2). In seinem Unterverzeichnis RAPID finden Sie alle Programme, die momentan auf ihm aufgespielt sind. Hier sollten sich nach diesem Schritt die Dateien „CalibData.mod“ und „positionenAbfahren“ befinden. Um diese von „Current Station“ auf den Controller zu übertragen müssen Sie eine Verbindung zwischen beiden anlegen. Dies können Sie durch die Funktionalität „Create Relation“, welche Sie unter der Registerkarte „Controller“ ganz rechts finden.

Achten Sie darauf, dass Sie bei der Datenübertragung nur die beiden nötigen Dateien „positionenAbfahren“ und „CalibData.mod“ ausgewählt haben und nicht versehentlich System-Module überschreiben (siehe Abb. 8.3). Bevor Sie die Dateien kopieren können, müssen Sie Schreibzugriff auf den Controller anfordern. Im Automatik-Modus können Sie dies direkt über RobotStudio erlauben. Ist er auf Handbetrieb eingestellt müssen Sie dies zunächst auf dem FlexPendant bestätigen.

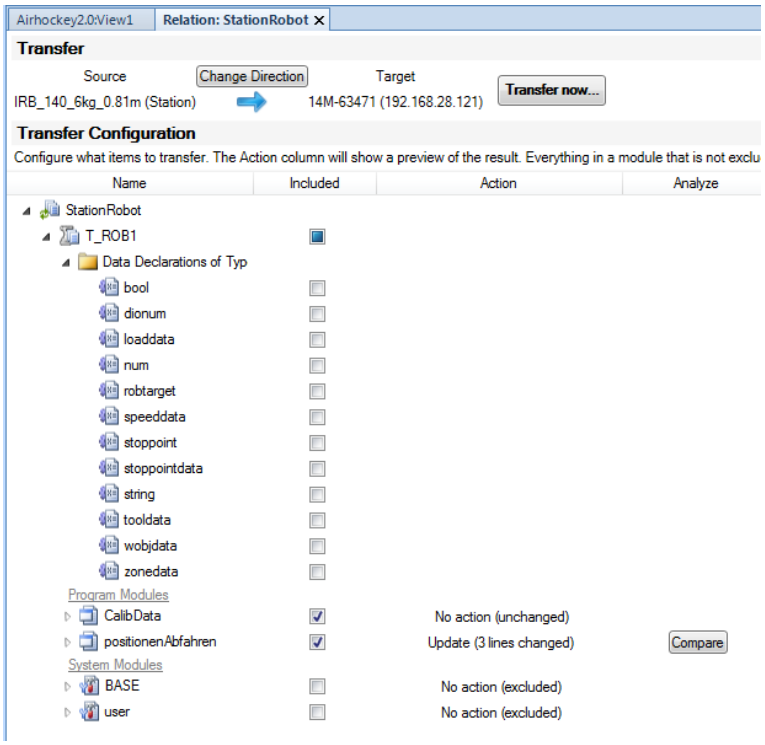

Abbildung 8.3: Datentransfer auf den Roboter-Controller

8.2 Kalibrierung des Roboters

Wurde der Roboter längere Zeit nicht mehr benutzt, sollte vor der Benutzung getestet werden, ob der Roboter noch richtig kalibriert ist. Zu diesem Zweck gibt es die Routine „Kalibrierungstest“. Wird diese ausgeführt, werden mit langsamer Geschwindigkeit die vier Eckpunkte des Bereiches abgefahren in dem er sich bewegen kann, wobei sich Schlägerunterkante 30 mm über der Tischplatte befindet (siehe Listing 8.1). Dabei sollte zum einen überprüft werden, ob die Schlägerunterkante überall 3 cm über dem Tisch ist. Dazu das Programm stoppen und an verschiedenen Stellen nachmessen. Außerdem sollte der Schläger genau in die beiden Ecken des Spielfeldes fahren und sich parallel zu den Banden bewegen. Sollte dies nicht der Fall sein, oder man möchte gar einen anderen Schläger verwenden, muss eine Neukalibrierung vorgenommen werden. Dazu müssen drei Schritte durchgeführt werden, die in den folgenden Abschnitten beschreiben werden.

```
1 !Definitionen der Targets aus der Datei CalibData
2
3 CONST robtarget Target_10:=[[xMin,yMin,30],[0,1,0,0],[-1,0,0,0],[9E9,9E9
      ,9E9,9E9,9E9,9E9]];
4 CONST robtarget Target_20:=[[xMin,yMax,30],[0,1,0,0],[-1,0,0,0],[9E9,9E9
      ,9E9,9E9,9E9,9E9]];
5 CONST robtarget Target_30:=[[xMax,yMax,30],[0,1,0,0],[-1,0,0,0],[9E9,9E9
      ,9E9,9E9,9E9,9E9]];
```

```
CONST robtarget Target_40:=[[xMax,yMin,30],[0,1,0,0],[-1,0,0,0],[9E9,9E9
    ,9E9,9E9,9E9,9E9]];

! Die Routine Kalibrierungstest aus der Datei positionenAbfahren

PROC Kalibrierungstest()
  MoveL Target_10,v50,fine,tool0\WObj:=Workobject_Table;
    MoveL Target_20,v50,fine,tool0\WObj:=Workobject_Table;
    MoveL Target_30,v50,fine,tool0\WObj:=Workobject_Table;
    MoveL Target_40,v50,fine,tool0\WObj:=Workobject_Table;

ENDPROC
```

Listing 8.1: RAPID-Routine: Kalibrierungstest

8.2.1 Schläger montieren

Wurde der Schläger demontiert oder möchte man einen neuen Schläger montieren, so muss man den Roboterarm zunächst auf die vordefinierte Home-Position fahren. In dieser Position kann man den Schläger bequem ab- und anschrauben. Viel wichtiger ist jedoch, dass an dieser Position die Orientierung des Werkzeughalters (als Quaternion ausgedrückt) und die Achsenkonfiguration genau die gleichen sind wie bei den Positionsangaben, die dem Roboter später übergeben werden. Man kann den Roboter auf unterschiedliche Arten auf diese Position bringen. Zum einen durch ausführen der Routine „Jump-Home“ in RobotStudio (Automatik-Modus) oder über das FlexPendant (Handbetrieb) (siehe Listing 8.2). Über das FlexPendant kann die Position („Target_Home“) auch direkt angefahren werden, wobei man dabei darauf achten muss, dass als Koordinatensystem „Welt“, als Werkzeug „tool0“ und als Workobject „wobj0“ eingestellt ist. Bei der Montage des Schlägers muss man anschließend darauf achten, dass der Schläger möglichst parallel zur Roboterachse nach vorne zeigt.

```
!Definitionen der Home-Position aus der Datei CalibData
CONST robtarget Target_Home:=[[453,26,610],[0,1,0,0],[-1,0,0,0],[9E9,9E9
    ,9E9,9E9,9E9,9E9]];

! Die Routine Jump_Home aus der Datei positionenAbfahren

PROC Jump_Home()
    MoveL Target_Home,v50,fine,tool0\WObj:=wobj0;
ENDPROC
```

Listing 8.2: RAPID-Routine: Jump-Home

8.2.2 Eckpunkte abfahren

Bei diesem Schritt muss man zunächst auf dem FlexPendant als Koordinatensystem „Welt“, als Werkzeug „tool0“ und als Workobject „wobj0“ einstellen. Danach können wie in Abbildung 8.4 dargestellt, die drei Positionen abgefahren und jeweils die X-,Y- und Z-Koordinaten notiert werden. Dabei darf der Roboter nur lineare bewegt werden. Außerdem sollte man bei „Position 1“ möglichst genau in die Ecke fahren, da dies später der Ursprung des Werkobjekt-Koordinatensystems sein wird.

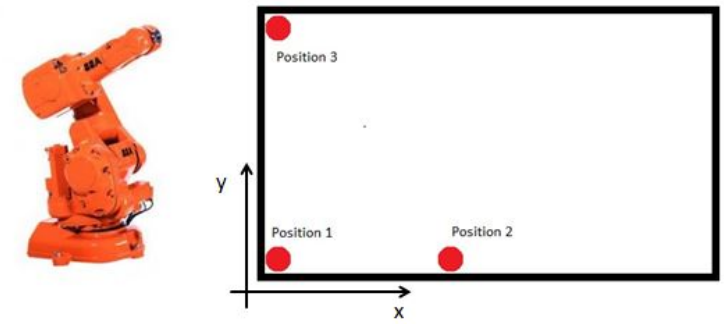

Abbildung 8.4: Die drei Positionen zur Kalibrierung des Roboters

8.2.3 Workobject erzeugen

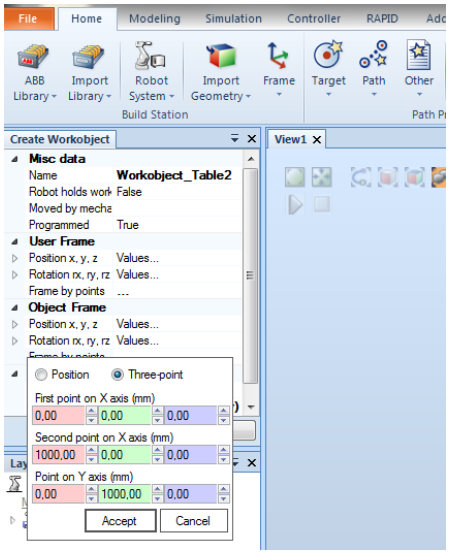

Abbildung 8.5: Workobject über drei Punkte definieren

Mit Hilfe der notierten Koordinaten, kann in Robotstudio ein neues Workobject angelegt werden. Dazu muss auch hier zunächst sichergestellt werden, dass als Koordinatensystem „Welt“, als Werkzeug „tool0“ und als Workobject „wobj0“ eingestellt ist. Zum erzeugen eines neuen Workobjects klickt man unter dem Reiter „Home“ auf „Other“ und anschließend auf „Creat Workobject“. In der linken Leiste öffnet sich ein entsprechendes Konfigurationmenü. Hier muss man nun das Feld „Frame by points“ ausklappen und den Radio-Button „Three-point“ auswählen (siehe Abb. 8.5). Nun können hier für die drei Punkte die notierten Koordinaten in der vorgegebenen Reihenfolge eingegeben werden.

Die Definition diese Workobjects findet man anschließend in der Datei „CalibData“ der „Current-Station“

im Verzeichnis „RAPID“. Um diese Datei zu aktualisieren, muss man zunächst eine Synchronisation mit der Station vornehmen. Dafür einfach im Reiter „RAPID“ in der oberen Menüleiste auf „Sychronize“ und dann auf „Synchronize to RAPID“ klicken.

Übertragen Sie anschließend die neuen Daten auf den Roboter. Dies können Sie wieder über die „Relation“ machen, oder wenn der Roboter sich im Automatik-Modus befindet auch durch Editieren der Datei „CalibData“ auf dem Controller des Roboters. Ersetzen Sie dazu die alte Definition von „wobj_Table“ durch die soeben erstellte.

Wie weit sich der Roboter entlang der x- bzw. y- Achse bewegen kann, können Sie über die Variablen „xMax“ bzw. „yMax“ einstellen. Wie weit er sich in der Mitte (bei y = 345) nach vorne bewegen kann über die Variable „xMaxMitte“. So wie wir den Roboter konfiguriert hatten, ergab sich ein Koordinatensystem für den Roboter wie es in Abbildung 6.1 auf Seite 15 dargestellt ist.

8.3 Ein Spiel starten

In diesem Abschnitt wird Schritt für Schritt beschrieben, wie man ein Spiel startet, wenn an den beiden Rechnern und dem Roboter seit Beendigung unserer Studienarbeit nichts mehr verändert wurde, oder bereits die Anweisungen der beiden vorhergehenden Abschnitte erfolgreich umgesetzt wurden.

Falls Sie zum ersten Mal ein Spiel starten, vergewissern Sie sich zunächst, ob sich die IP-Adressen geändert haben. Der PC mit der Bilderkennung hatte die IP-Adresse „192.168.28.222“. Hat sich diese geändert, müssen die Datei „positionenAbfahren“ auf dem Controller des Roboters und bei der Bilderkennung die Funktion „___init___“ in der Datei „Connection.py“ entsprechend geändert werden.

8.3.1 Roboter-Programm starten

Zuerst muss das RAPID-Programm auf dem Roboter-Controller gestartet werden. Dazu müssen folgende sechs Schritte durchgeführt werden:

1. Öffnen Sie RobotStudio auf dem Windows-PC und öffnen Sie die Datei „Airhockey2.0.rssln“. Sofern Sie noch den selben Rechner verwenden finden Sie diese Datei auf Laufwerk C im Ordner „Airhockey2.0“. Andernfalls in dem Verzeichnis, indem Sie das Projekt entpackt haben.

2. Stellen Sie zunächst sicher, dass der Roboter eingeschaltet ist und auf Automatik-Betrieb eingestellt ist (Schalter 4 in Abb. 4.1 nach links drehen). Gehen Sie dann in RobotStudio unter dem Reiter „Controller“ auf „Add Controller“. Der Roboter-Controller hat die IP-Adresse 192.168.28.121 und sollte ganz unten im Drop-Down-Menü zu finden sein (siehe Abb. 8.1). Oft muss man um ihn auswählen zu können, das Menü nochmal einklappen und dann erneut auf „Add Controller“ klicken.

3. Wechseln Sie in den Reiter „RAPID“ und öffnen Sie die Datei „positioinenAbfahren“ im Verzeichnis „RAPID“ aus dem Roboter-Controller (14M-63471(192.168.28.121))(siehe Abb. 8.2). Fordern Sie nun Schreibzugriff auf den Roboter-Controller an, indem Sie auf den Knopf „Request Write Acces“ drücken.

4. Starten Sie die Motoren des Roboters durch drücken von Schalter 3 am Schaltschrank(siehe Abb. 4.1 auf Seite 5).

5. Setzen Sie als nächstes den Programm-Zeiger zurück. Klicken Sie dazu auf „Program Pointer“ und wählen Sie dann „Set Program Pointer to Main in all tasks“. Diesen Schritt müssen auch vor jedem weiteren Start des Programmes durchführen, da zu Beginn immer erst eine Verbindung mit der Bilderkennung aufgebaut werden muss.

6. Als letztes betätigen Sie in RobotStudio den Start-Knopf. Der Schläger wird dann sofort in die Mitte des Tores gefahren, bevor versucht wird eine Verbindung mit der Bilderkennung aufzubauen.

8.3.2 Bildverarbeitung starten

1. Stellen Sie zunächst sicher, dass die Kamera an den Linux-Rechner angeschlossen ist.

2. Öffnen Sie das Terminal und navigieren Sie ins Verzeichnis „Schreibtisch/Airhockey/airhockey“

3. Starten Sie die Bilderkennung, indem Sie hier das Skript „table_setup.sh“ ausführen. Dieses erwartet zwei Interger-Werte als Übergabeparameter. Zum einen den Kameraanschluss und zum anderen die Framerate, mit der die Kamera aufnehmen soll. Den Kameraanschluss findet man im Verzeichnis „/dev“ heraus. Suchen Sie hier nach Ordnern, die mit „video“ beginnen. Ist nur eine Kamera angeschlossen, sollte ein Ordner „video0“ existieren. Der erste Übergabewert wäre somit 0. Als Framerrate sollten 30 Bilder pro Sekunde genutzt werden. Der Befehl zum Starten des Skripts ergibt sich damit wie folgt:

```
1 sh table_setup 0 30
```

Listing 8.3: Terminal-Befahl zum starten der Bilderkennung

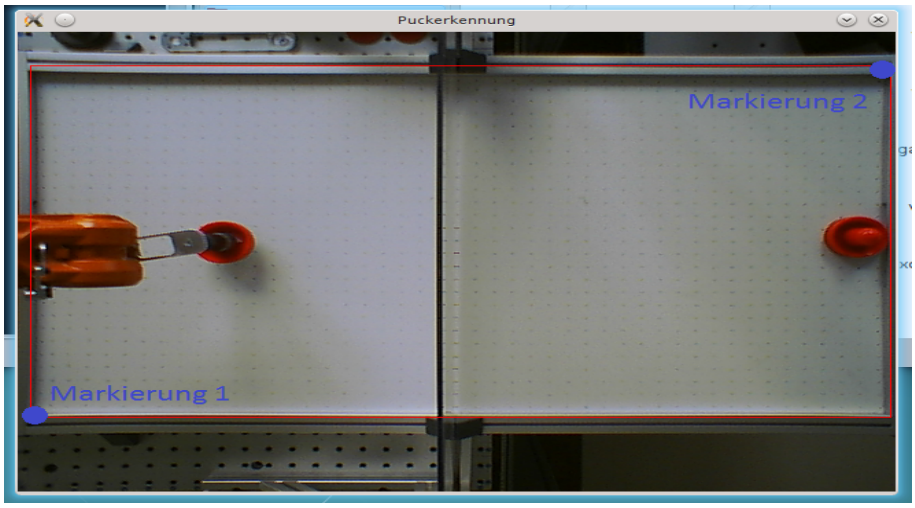

Abbildung 8.6: Einzeichnen des Spielfeldes

4. Das Programm versucht nun zunächst eine Verbindung mit dem Roboter-Controller auf zu bauen. Erst wenn dies erfolgreich war, öffnet sich ein Fenster, in dem ein Standbild des Tisches zu sehen ist. Darin muss nun zunächst das Spielfeld eingezeichnet werden. Dazu müssen Sie zunächst die untere linke Ecke und anschließend die obere rechte Ecke des Spielfeldes mit einem Doppelklick markieren (siehe Abb. 8.6). Dabei ist sehr wichtig, dass die Punkte genau in dieser Reihenfolge markiert werden, da ansonsten die Bilderkennung ein falsches Koordinatensystem zur Berechnung verwendet.

5. Haben Sie alle vorherigen Schritte erfolgreich abgearbeitet, können Sie nun Air-Hockey gegen den ABB-Roboter spielen. Wir wünschen Ihnen viel Spaß dabei. Zum Beenden des Bilderkennung einfach im Terminal „Strg + C“ eingeben.

9 Fazit und Ausblick

Ziel dieser Studienarbeit war es aufbauend auf zwei anderen Studienarbeiten eine Software zu entwickeln, welche es einem Roboter ermöglicht, gegen einen menschlichen Gegenspieler Air-Hockey zu spielen. Diese Ziel wurde in weiten Teilen erfüllt.

Am Anfang des Projekts wurde einiges an Zeit dafür benötigt, sich in die Bedienkonzept des Roboters einzuarbeiten und um die Bildverarbeitungssoftware der Vorgänger-Gruppe auf einem neu aufgesetzten Linux-Rechner wieder zum laufen zu bringen. Letzterer Punkt ist auch der Grund dafür, warum wir in Kapitel 8 eine detaillierte und hoffentlich auch verständliche Anleitung geschrieben haben, welche Schritte notwendig sind, um das Projekt wieder neu aufzusetzen.

Im Anschluss an diese Einarbeitungsphase konnte eine Kommunikation zwischen Roboter-Controller und Bildverarbeitung realisiert werden und Roboterarm so die ersten Bewegungsbefehle übermittelt werden.

Bei der Implementierung einer offensiven Spielstrategie, bei der Roboter den Puck zurückschlagen soll, zeigte sich, dass es einige Faktoren gibt, die die Fähigkeiten des Roboters Air-Hockey zu spielen stark einschränken. Manche dieser Probleme konnten auf Grund mangelnder Zeit nicht mehr in dieser Studienarbeit behoben werden. Im folgenden werden die größten noch bestehenden Problem aufgelistet und mögliche Lösungsansätze vorgestellt, mit denen es nachfolgenden Gruppen vielleicht möglich ist das Air-Hockey-Spiel des ABB-Roboters zu verbessern.

1. **Ungenaue Bewegungsvorhersage des Schlägers:**

   Dazu wurde bereits in Abschnitt 6.3 auf Seite 21 ein Verfahren beschrieben, bei dem man anstatt kontinuierliche Positionswerte nur diskrete Werte benutzt. Eine weitere Verbesserung des Spielverhaltens könnte man dadurch erreichen, dass man bei der MoveL-Instruktion den Übergabeparameter Zeit einsetzt (siehe Abschnitt 4.2.2 auf Seite 8). Anstatt bei einer Bewegung in x-Richtung auf einen passenden Zeitpunkt zu warten, könnte man damit den Schläger mit variabler Geschwindigkeit zur Zielposition fahren.

2. **Ungenaue Bewegungsvorhersage des Pucks:**

   Es fiel auf, dass die berechnete Bewegung des Pucks bei einer Reflexion an einer Bande oft nicht mit der tatsächlichen Bewegung übereinstimmt. Dies liegt daran, dass bei den

Berechnungen eine ideale Reflexion angenommen wurde, bei der also Ausfallswinkel gleich Einfallswinkel ist. Um den realen Stoßprozess besser approximieren zu können, bedarf es weitere Untersuchungen um geeignete Stoßzahlen zu ermitteln und diese in den entsprechenden Gleichungen einfügen zu können.

3. **Hohe Reaktionszeit und zu geringe Geschwindigkeit:**

   Es fiel auf, dass des Roboter-Controller ein gewisse Zeit braucht, um aus den Positionsdaten entsprechende Bewegungsbefehle zu berechnen. Des weiteren, dass vor allem die Bewegung entlang der x-Achse ziemlich langsam ist. Beides könnte durch eine direkte Ansteuerung der Roboterachsen verbessert werden.

Literatur

[1] *Bedienanleitung: Einführung in RAPID*. ABB, SE-721 68 Västerås Schweden, 2007.

[2] P. Hofmann. *Freie Rotation im Raum: Quaternionen und Matrizen*. 2009. URL: https://www.uninformativ.de/bin/SpaceSim-2401fee.pdf.

[3] *Produktspezifikation: Steuerung IRC5 mit FlexPendant*. ABB, SE-721 68 Västerås Schweden, 2004.

[4] Python Software Foundation. *Socket Programming HOWTO*. 2016. URL: https://docs.python.org/2/howto/sockets.html.

[5] *Technisches Referenzhandbuch: RAPID Instruktionen, Funktionen und Datentypen*. ABB, SE-721 68 Västerås Schweden, 2010.

[6] Wikipedia. *Bestimmtheitsmaß*. 2016. URL: https://de.wikipedia.org/wiki/Bestimmtheitsma%C3%9F.