In der Regel haben wir einen zweizeiligen Bachelorthesistitel

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
Ausbildungsfirma	DHBW, Karlsruhe
Betreuer	Prof. Dr.-Ing. Clemens Reitze
Gutachter	Prof. Dr.-Ing. Clemens Reitze

Sperrvermerk

Die vorliegende Studienarbeit mit dem Titel *In der Regel haben wir einen zweizeiligen Bachelorthesistitel* enthält unternehmensinterne bzw. vertrauliche Informationen der DHBW, ist deshalb mit einem Sperrvermerk versehen und wird ausschließlich zu Prüfungszwecken am Studiengang Informationstechnik der Dualen Hochschule Baden-Württemberg Karlsruhe vorgelegt. Sie ist ausschließlich zur Einsicht durch den zugeteilten Gutachter, die Leitung des Studiengangs und ggf. den Prüfungsausschuss des Studiengangs bestimmt. Es ist untersagt,

- den Inhalt dieser Arbeit (einschließlich Daten, Abbildungen, Tabellen, Zeichnungen usw.) als Ganzes oder auszugsweise weiterzugeben,
- Kopien oder Abschriften dieser Arbeit (einschließlich Daten, Abbildungen, Tabellen, Zeichnungen usw.) als Ganzes oder in Auszügen anzufertigen,
- diese Arbeit zu veröffentlichen bzw. digital, elektronisch oder virtuell zur Verfügung zu stellen.

Jede anderweitige Einsichtnahme und Veröffentlichung – auch von Teilen der Arbeit – bedarf der vorherigen Zustimmung durch den Verfasser und DHBW.

Karlsruhe, Mai 2016

Claus Burkhart & René Glockan

Erklärung

Ich erkläre hiermit ehrenwörtlich:

1. dass ich unsere Studienarbeit mit dem Thema *In der Regel haben wir einen zweizeiligen Bachelorthesistitel* ohne fremde Hilfe angefertigt habe;
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe;
3. dass ich unsere Studienarbeit bei keiner anderen Prüfung vorgelegt habe;
4. dass die eingereichte elektronische Fassung exakt mit der eingereichten schriftlichen Fassung übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Karlsruhe, Mai 2016

Claus Burkhart & René Glockan

Abstract

Abstract normalerweise auf Englisch. Siehe: http://www.dhbw.de/fileadmin/user/public/Dokumente/Portal/Richtlinien_Praxismodule_Studien_und_Bachelorarbeiten_JG2011ff.pdf (8.3.1 Inhaltsverzeichnis)

Ein „Abstract“ ist eine prägnante Inhaltsangabe, ein Abriss ohne Interpretation und Wertung einer wissenschaftlichen Arbeit. In DIN 1426 wird das (oder auch der) Abstract als Kurzreferat zur Inhaltsangabe beschrieben.

Objektivität soll sich jeder persönlichen Wertung enthalten

Kürze soll so kurz wie möglich sein

Genauigkeit soll genau die Inhalte und die Meinung der Originalarbeit wiedergeben

Üblicherweise müssen wissenschaftliche Artikel einen Abstract enthalten, typischerweise von 100-150 Wörtern, ohne Bilder und Literaturzitate und in einem Absatz.

Quelle: http://de.wikipedia.org/wiki/Abstract Abgerufen 07.07.2011

Diese etwa einseitige Zusammenfassung soll es dem Leser ermöglichen, Inhalt der Arbeit und Vorgehensweise des Autors rasch zu überblicken. Gegenstand des Abstract sind insbesondere

- Problemstellung der Arbeit,
- im Rahmen der Arbeit geprüfte Hypothesen bzw. beantwortete Fragen,
- der Analyse zugrunde liegende Methode,
- wesentliche, im Rahmen der Arbeit gewonnene Erkenntnisse,
- Einschränkungen des Gültigkeitsbereichs (der Erkenntnisse) sowie nicht beantwortete Fragen.

Quelle: http://www.ib.dhbw-mannheim.de/fileadmin/ms/bwl-ib/Downloads_alt/Leitfaden_31.05.pdf, S. 49

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

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 Studenarbeit weiter. Unsere Vorgänger sind soweit gekommen, das sie die Bilderkennung soweit fertig hatten, dass der Puck auf dem Spielfeld mithilfe einer Kamera erkannt werden konnte. Zusätzlich konnte eine recht genaue Flugbahn des Puckes bereits berechnet werden. Diese Infors wurden alle auf einer grafischen Oberfäche 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.

2 Aufgabenstellung

Das Ziel dieser Studienarbeit ist es einem Roboter von ABB das Airhockey spielen beizubringen. Da dies eine Fortsetzung einer früheren Studienabreit 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 Ansteurung 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 einer der beiden Punkte denen wir uns in der Studienarbeit gewidmet haben. Der zweite und wichtigste Aufgabenteil war der Entwurf einer Spielstrategie.

3 Ablaufplanung

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 Defensivestrategie ausgearbeitet. Sie hat bis zum ende des 5. Semesters erstmal funktioniert. Diese hat allerdings mehr schlecht als recht funktioniert. Für das 6. Semester werden wir als erstes den kompletten Code von unseren Vorgängern refactorn und diesen übersichtlicher machen sowie mehr kommentieren um den Code für eventuelle zukünftige Generationen freundlicher zu gestalten. Sobald dies erledigt ist, wird 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, sofern dies nicht schon beim refactorn geschehen ist.

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: ([1] 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.

4.1 Robotersteuerung mit dem FlexPendant

4.1.1 Fernsteuerung des Roboters

4.1.2 Programm starten

4.2 Einführung in RobotStudio

4.2.1 Neues Projekt anlegen

4.2.2 Roboter bewegen

4.2.3 Objekte anlegen

4.2.4 Workobjekt anlegen

4.2.5 Datentransfer zum Controller des Roboters

4.3 Einführung in RAPID

4.3.1 Robtarget

4.3.2 Pfade abfahren

Abbildung 4.1: Schaltschrank

5 Kommunikation zwischen Bilderkennung und Roboter

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. Die 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 letztenendes 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.

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.

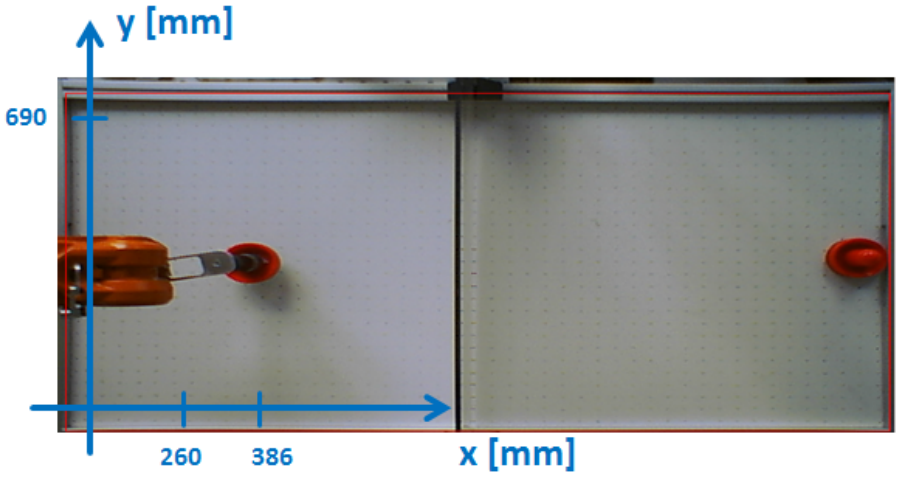

Abbildung 6.1: Koordinatensystem des Roboters

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 Ecxel 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. [2]

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 die Steigungen und y-Achsenabschnitte der Geraden-Gleichung sehr stark unterscheiden ja nachdem wo man den Startpunkt wählt und wie man die x- bzw. y- Koordinaten der Endpunkte bei einem Durchlauf variiert.

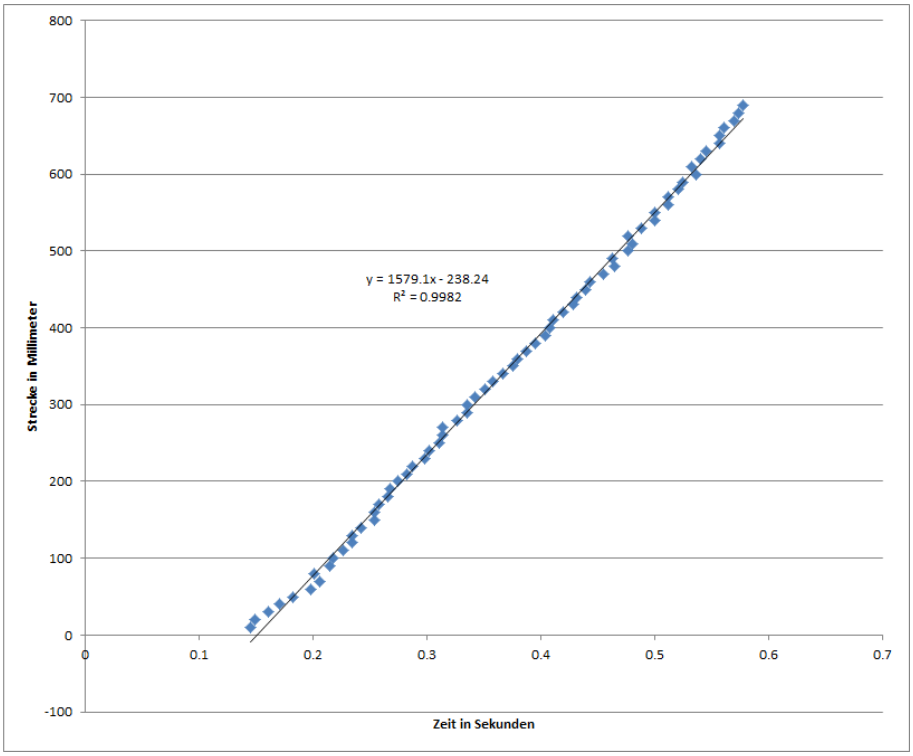

Abbildung 6.2: Bewegung entlang der y-Achse auf der Grundlinie

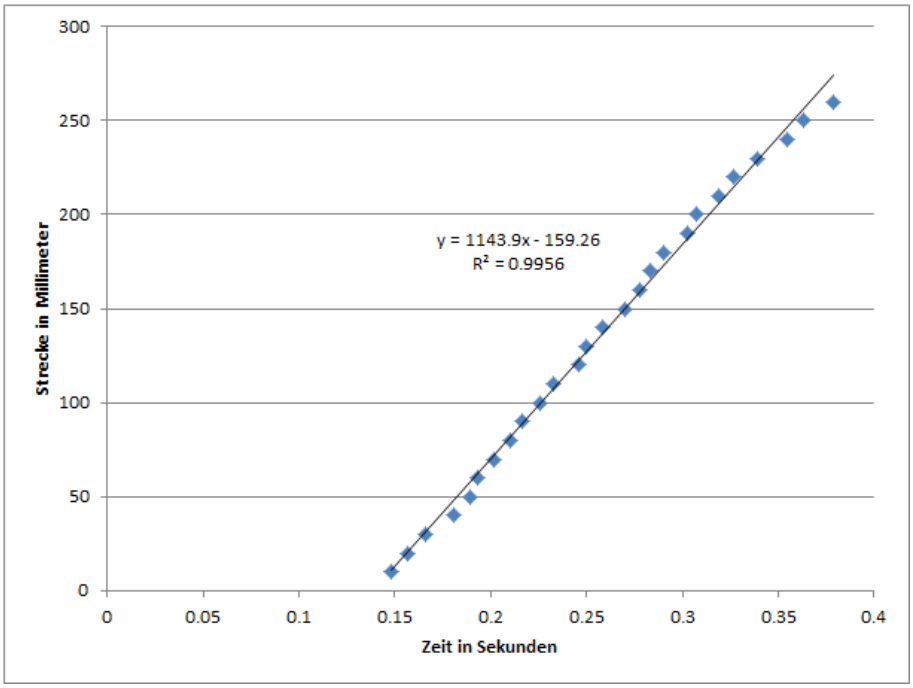

Abbildung 6.3: Bewegung entlang der x-Achse bei y = 570

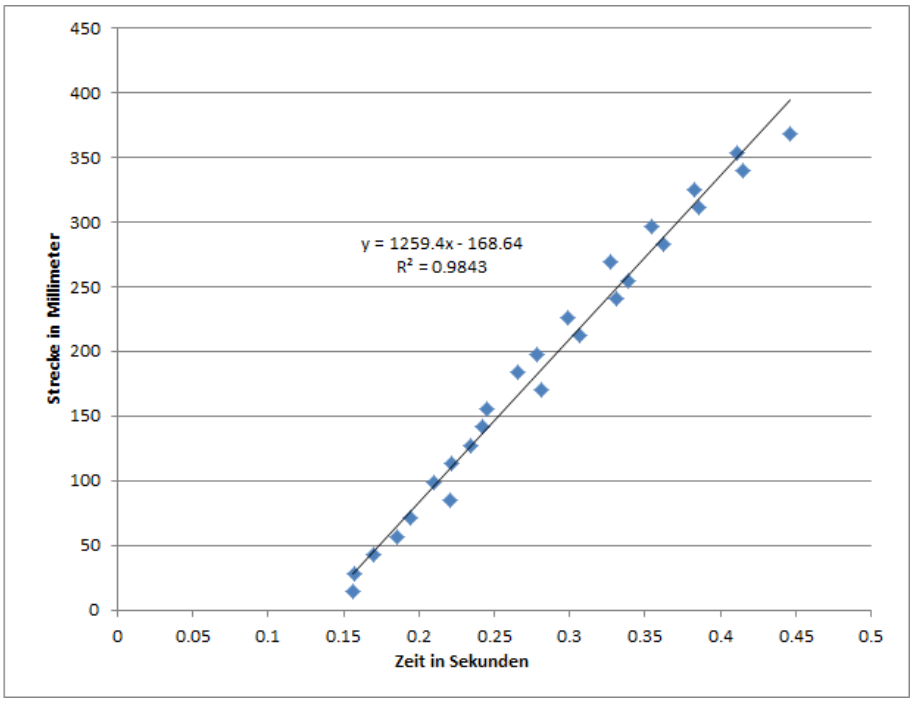

Abbildung 6.4: Diagonalbewegung

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 Δyauf 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 ist, 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 die Abhängigkeiten dieser beiden Größen zur y-Koordinate jeweils durch ein Polynom zweiten Grades annähern

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

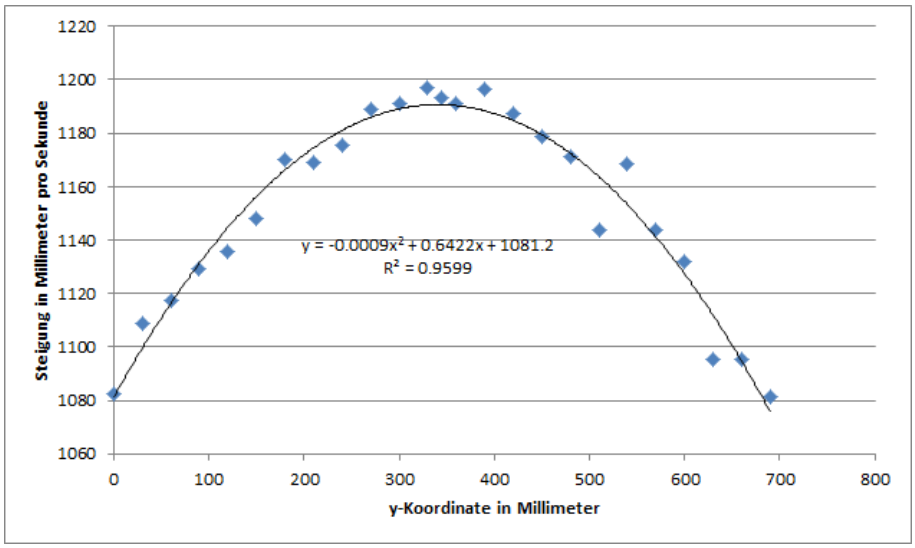

Abbildung 6.5: Zusammenhang zwischen y-Koordinate und Steigung der Trendlinie

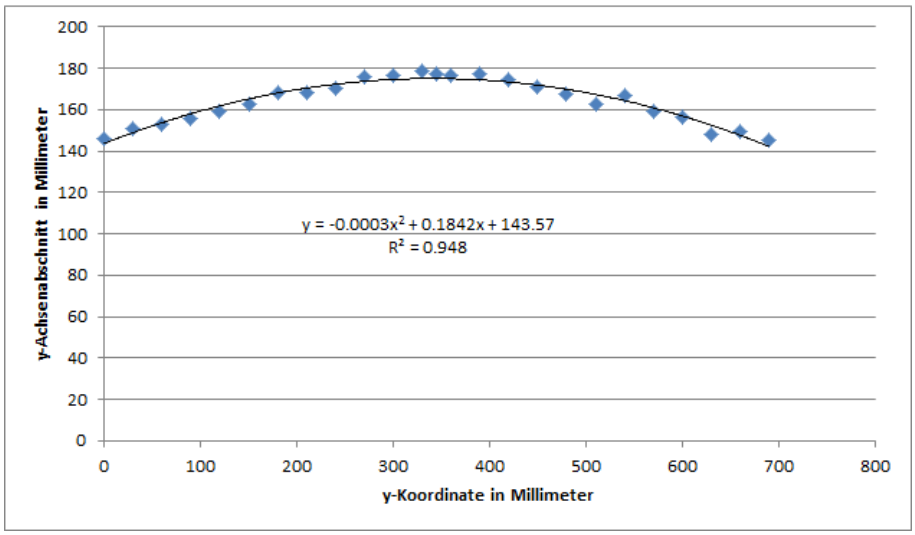

Abbildung 6.6: Zusammenhang zwischen y-Koordinate und y-Achsenabschnitt der Trendlinie

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 unsere Hälfte 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 bewegen als sonst. Bei der Untersuchung der Bewegung in x-Richtung an dieser Stelle (y=345) fiel auf, dass sich die Bewegung für diese

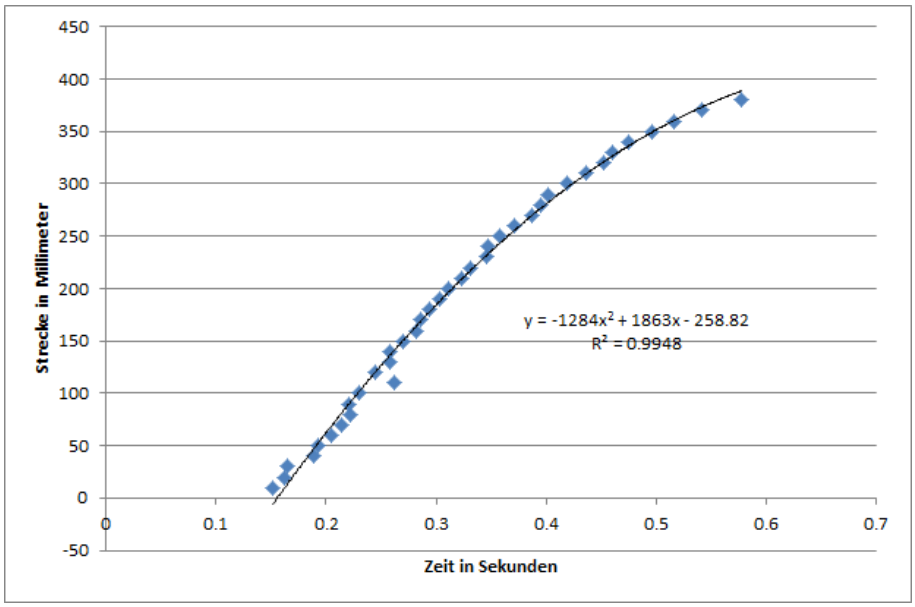

Abbildung 6.7: Bewegung entlang der x-Achse bei y = 345

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 Erweiterung des Bilderkennungsprogramms um Kommunikationsmodul

Die Kommunikation zwischen PC und Roboter erfolgt wie in Kapitel 5 auf Seite 7 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 Übertratung oder eine kleine Folge von Übertragungen genutzt wird und anschließend wieder geschlossen wird.

> 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).[1] Da auch der Socket auf der Serverseite geschlossen wurde, 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[2] 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

[1] biber referenz für https://docs.python.org/2/howto/sockets.html
[2] Bruch?

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 Test hat dann häufig bis zu 3 Werte hintereinander geschickt, bevor es gewartet hat, das der Roboter einen neuen Serversocket geöffnet hat. Wir führten darauf hin einen Two-Way-Handshake ein. Ursprünglich hatten wir dann solange gewartet bis der Roboter sein OK geschickt hat. Dabei hatten wir nur ein nicht beachtet und zwar dass das Programm sequentiell abläuft und für die 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. Dadurch konnten wir zwar die Aussreißer 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 Amfang 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 dann so zu lassen. 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 jedesmal den Programmcode umschreiben wollten um einen neuen Port einzutragen, mussten wir zwischen den Tests immer ein weilchen warten.

7.2 Implementierung von Spielstrategien

7.2.1 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. Der Roboter versucht 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. Dabei sollte sich der Puck zwischen dem Tor des Roboters und dem Puck positionieren und dem nächsten Angriff des Spielers zuvorzukommen.

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

7.2.2 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. 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.

7.2.3 Defensive Spielstrategie mit offensiven Ansätzen

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

7.2.4 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 Halbkreis um das Tor reduziert werden um unnötige Verteidigungsaktionen zu vermeiden.

7.3 Spielstrategie entscheiden

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.

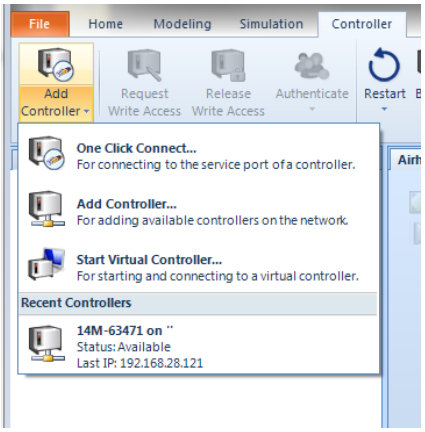

Abbildung 8.1: Controller hinzufügen

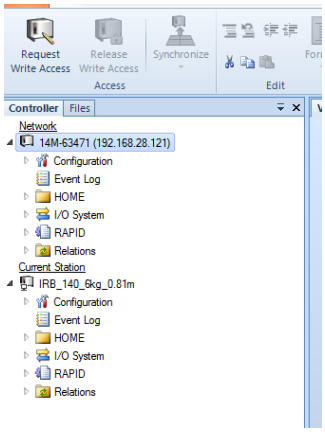

Abbildung 8.2: Verzeichnisstruktur

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.

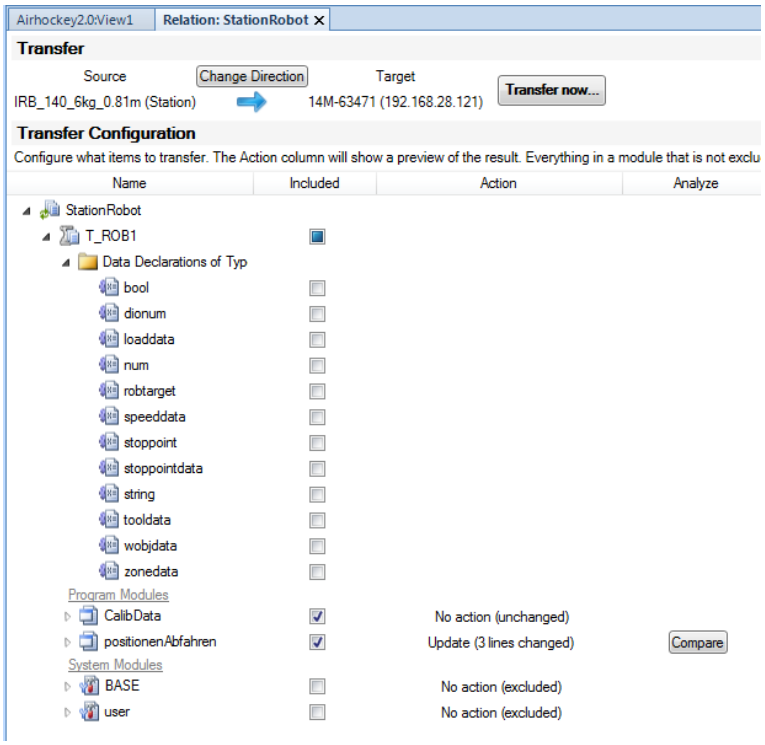

Abbildung 8.3: Datentransfer auf den Roboter-Controller

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 die folgenden drei Schritte durchgeführt 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]];
```

```
6  CONST robtarget Target_40:=[[xMax,yMin,30],[0,1,0,0],[-1,0,0,0],[9E9,9E9
       ,9E9,9E9,9E9,9E9]];
7
8  ! Die Routine Kalibrierungstest aus der Datei positionenAbfahren
9
10 PROC Kalibrierungstest()
11   MoveL Target_10,v50,fine,tool0\WObj:=Workobject_Table;
12     MoveL Target_20,v50,fine,tool0\WObj:=Workobject_Table;
13     MoveL Target_30,v50,fine,tool0\WObj:=Workobject_Table;
14     MoveL Target_40,v50,fine,tool0\WObj:=Workobject_Table;
15
16 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 Workobjekt „wobj0“ eingestellt ist. Bei der Montage des Schlägers muss man anschließend darauf achten, dass der Schläger möglichst parallel zur Roboterachse nach vorne zeigt.

```
1  !Definitionen der Home-Position aus der Datei CalibData
2  CONST robtarget Target_Home:=[[453,26,610],[0,1,0,0],[-1,0,0,0],[9E9,9E9
       ,9E9,9E9,9E9,9E9]];
3
4  ! Die Routine Jump_Home aus der Datei positionenAbfahren
5
6  PROC Jump_Home()
7      MoveL Target_Home,v50,fine,tool0\WObj:=wobj0;
8  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 Workobjekt „wobj0“ eingestellen. Danach könne 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. Dabei sollte man bei „Position 1“ möglichst genau in die Ecke fahren, da dies der Ursprung des Werkobjekt-Koordinatensystems ist.

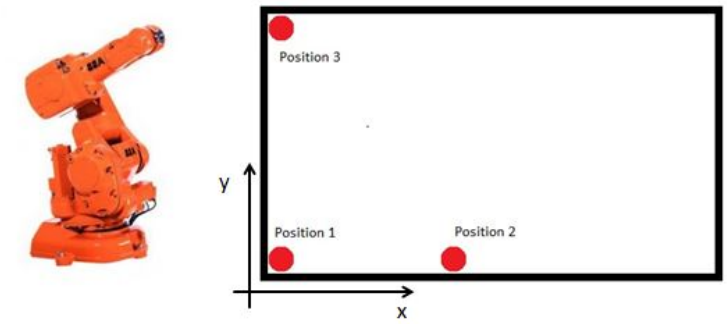

Abbildung 8.4: Die drei Positionen zur Kalibrierung des Roboters

8.2.3 Workobjekt erzeugen

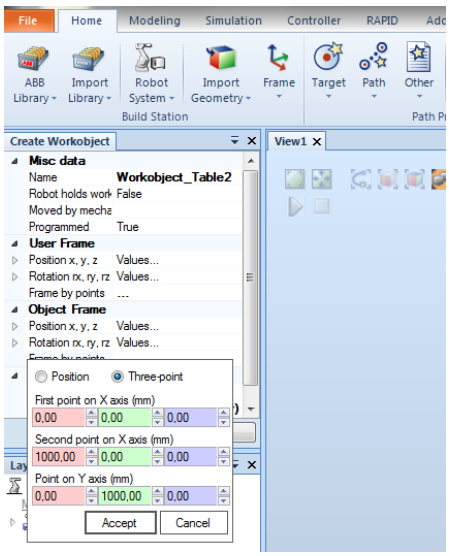

Abbildung 8.5: Workobjekt über drei Punkte definieren

Mit Hilfe der notierten Koordinaten, kann in Robotstudio ein neues Workobjekt angelegt werden. Dazu muss auch hier zunächst sichergestellt werden, dass als Koordinatensystem „Welt“, als Werkzeug „tool0“ und als Workobjekt „wobj0“ eingestellt ist. Zum erzeugen eines neuen Workobjekts klickt man unter dem Reiter „Home“ auf „Other“ und anschließend auf „Creat Workobjekt“. 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 Workobjekts 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 x = 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 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.

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 er eingeschaltet ist und auf Automatik-Betrieb eingestellt ist. 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).

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“

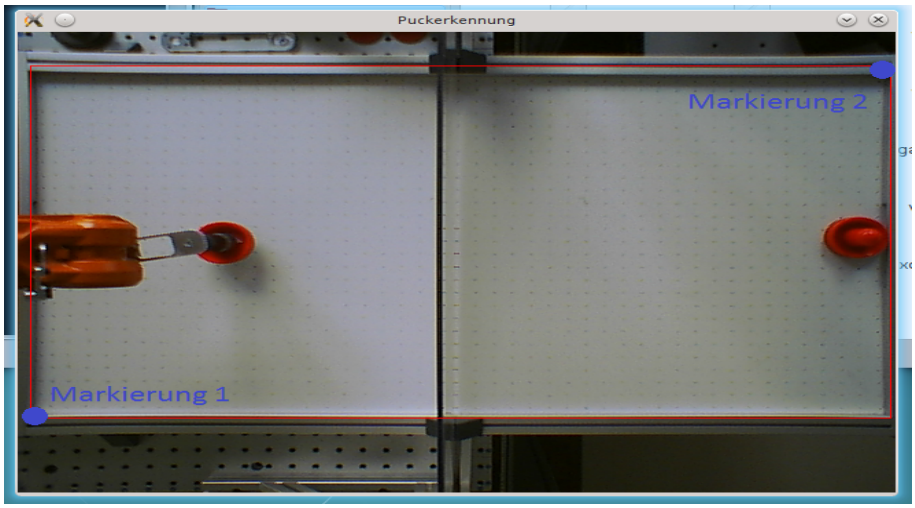

Abbildung 8.6: Einzeichnen des Spielfeldes

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 Framerate sollten 30 Bilder pro Sekunde genutzt werden. Der Befehl zum Starten des Skripts ergibt sich damit wie folgt:

```
sh table_setup 0 30
```

Listing 8.3: Terminal-Befahl zum starten der Bilderkennung

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 den 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 in im Terminal „Strg + C“ eingeben.

9 Fazit und Ausblick

Literatur

[1] *Produktspezifikation : Steuerung IRC5 mit FlexPendant*. SE-721 68 VstersSchweden, 2004.

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

Anhang

(Beispielhafter Anhang)

A. Assignment

B. List of CD Contents

C. CD

B. List of CD Contents

- **Literature/**
  - **Citavi-Project(incl pdfs)/** ⇒ *Citavi (bibliography software) project with almost all found sources relating to this report. The PDFs linked to bibliography items therein are in the sub-directory 'CitaviFiles'*
    - bibliography.bib ⇒ *Exported Bibliography file with all sources*
    - Studienarbeit.ctv4 ⇒ *Citavi Project file*
    - **CitaviCovers/** ⇒ *Images of bibliography cover pages*
    - **CitaviFiles/** ⇒ *Cited and most other found PDF resources*
  - **eBooks/**
  - **JournalArticles/**
  - **Standards/**
  - **Websites/**
- **Presentation/**
  - presentation.pptx
  - presentation.pdf
- **Report/**
  - Aufgabenstellung.pdf
  - Studienarbeit2.pdf
  - **Latex-Files/** ⇒ *editable LaTeX files and other included files for this report*
    - **ads/** ⇒ *Front- and Backmatter*
    - **content/** ⇒ *Main part*
    - **images/** ⇒ *All used images*
    - **lang/** ⇒ *Language files for LaTeX template*