Universität Bremen

Feldrand-Erkennung im Roboterfußball mittels Random Sample Set Consensus

Bachelorarbeit in der Informatik
Fachbereich 3
Universität Bremen

von

René Schröder

Betreuer:
Dr. Tim Laue
Prof. Dr. Rolf Drechsler

Tag der Anmeldung: 14. Juli 2017
Tag der Abgabe: 16. Oktober 2017

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

Bremen, den 16. Oktober 2017

Inhaltsverzeichnis

1. Einleitung

Dieses Kapitel behandelt die Motivation und die Ziele der Arbeit. Außerdem wird ein Überblick über die Arbeit gegeben.

1.1 Motivation

Die Feldgrenze (siehe Kapitel 4.1) spielt bei dem Roboterfußball eine sehr wichtige Rolle. Der Ball und alle interessanten Objekte, wie z.B. andere Roboter, befinden sich auf dem Spielfeld und somit auch zwangsläufig innerhalb der Feldgrenzen. Wenn sich die Feldgrenze also im momentanen Bild befindet, muss nur bis zu dieser das Bild durchsucht werden, wenn Objekte auf dem Feld gesucht werden. Das verhindert, dass Objekte – wie z.B. Zuschauer oder Werbebanden – für falsche Erkennungen sorgen. Außerdem wird dadurch Rechenzeit gespart, da unwichtige Bereiche des Bildes nicht durchsucht werden müssen. Wird die Feldgrenze in Feldkoordinaten transformiert, kann sie außerdem bei der Lokalisierung[1] der Roboter auf dem Spielfeld helfen. Eine falsche Erkennung der Feldgrenze kann dementsprechend dazu führen, dass eigentlich wichtige Bereiche des Bildes nicht durchsucht werden oder die Lokalisierung falsche Ergebnisse liefert. Die momentane Feldgrenze im System *B-Human* [1, Kapitel 4.1.11] setzt auf die *konvexe Hülle* [2] und berechnet diese iterativ. Dieser Ansatz führt dazu, dass die Feldgrenze auch unerwartete Formen annehmen kann. Nicht selten sorgen Roboter am Bildrand dafür, dass der berechnete Feldrand am Rand des Bildes stark abfällt. Die Implementierung einer *konvexen Hülle* ist allerdings sehr sparsam was die Rechenzeit angeht. Die Modellierung über eine *konvexe Hülle* ist nicht wirklich praktikabel für die Feldgrenze, da sich diese eigentlich nur durch Geraden zusammensetzt, welche orthogonal aufeinander stehen. Genau diese Annahme soll in dieser Entwicklung der Feldgrenzen-Erkennung genutzt werden. Die möglichen Punkte auf dem Feldrand werden gesucht, und anschließend werden über einen Algorithmus namens *RANSAC* (siehe Kapitel 4.2) Geraden in diesen Punkten gesucht. Diese Geraden müssen dann verschiedene Kriterien erfüllen, um als Feldgrenze akzeptiert zu werden. Die Erwartung ist, dass dieser Ansatz robuster gegen Roboter oder andere Objekte im Bild ist und somit u.a. zu einer besseren Lokalisierung führt. Außerdem

[1] Die Lokalisierung ist die Bestimmung der Position und Rotation des Roboters auf dem Spielfeld.

soll die Implementierung um ein *Tracking* erweitert werden, in welchem die letzten erkannten Feldgrenzen im Bild über eine längere Zeit mitgeführt werden – auch, wenn sich diese schon nicht mehr im Bild befinden. Dadurch erhält man bei einem laufenden Roboter eine Übersicht der letzten erkannten Feldgrenzen und kann auch Geraden und Ecken segmentieren, welche in einem gewissen Zeitintervall beobachtet wurden. Dies eröffnet eine Menge neuer Möglichkeiten für die Lokalisierung.

1.2 Ziele und Aufbau der Arbeit

Das Ziel dieser Arbeit ist die Implementierung eines *Moduls* (siehe Kapitel 4.5) im System *B-Human* (siehe Kapitel 3.3), welches die Feldgrenze in einem Bild erkennen kann. Außerdem sollen in diesem System die letzten erkannten Feldgrenzen über einen gewissen Zeitintervall gespeichert werden, um Feldgrenzen und Ecken außerhalb des Bildes erkennen zu können. Die Ausführung des gesamten *Moduls* soll dabei nicht länger als zwei Millisekunden dauern.

Zu Beginn werden einige verwandte Arbeiten vorgestellt (Kapitel 2). Danach werden die Hintergründe und der Rahmen erläutert, in dem diese Arbeit entstanden ist (Kapitel 3). Anschließend wird auf die Grundlagen eingegangen, welche wichtig für das Verständnis der Arbeit sind (Kapitel 4). Hier werden die wesentlichen Algorithmen vorgestellt. Danach wird in der Umsetzung (Kapitel 5) ausführlich auf die Implementierung eingegangen. Zum Ende folgen die Evaluation (Kapitel 6), in welcher das System getestet wird und letztendlich ein Fazit mit einem Ausblick gegeben wird (Kapitel 7).

2. Verwandte Arbeiten

Yongbo Qian und *Daniel D. Lee* [3] von der *University of Pennsylvania* detektierten die Feldgrenze mittels einer Kombination aus einer *Konvexen Hülle* und *RANSAC*. Nachdem mögliche Feldrand-Punkte detektiert wurden, wurde eine *Konvexe Hülle* berechnet um die gröbsten Ausreißer zu eliminieren, bevor eine *RANSAC*-Implementierung angewendet wurde um Linien zu finden. Die Entscheidung, ob eine zweite Feldgrenze gesucht werden soll, wurde hier aufgrund der Verteilung der verbliebenen Punkte getroffen.

Thomas Reinhardt von der *Hochschule für Technik, Wirtschaft und Kultur Leipzig* [4] detektierte die Feldgrenze auch mittels einer *Konvexen Hülle*. Um grobe Ausreißer zu eliminieren, entwickelte er anschließend eine Variation seiner *Konvexen Hülle*, welche den Einfluss eines jeden Punktes auf die berechnete Feldgrenze berücksichtigt.

Die momentan bei B-Human genutzte Implementierung [1, Kapitel 4.1.11] nutzt die *Konvexe Hülle* ähnlich der Varianten, wie sie auch *Thomas Reinhardt* [4] implementiert hat. Dabei werden iterativ die Punkte eliminiert, welche einen zu großen vertikalen Abstand zu ihren Nachbarpunkten aufweisen. Aus jeder Iteration entsteht eine neue *Konvexe Hülle*. Die Beste zeichnet sich dadurch aus, dass ihre Punkte alle dicht beieinander liegen. Die möglichen Feldrand-Punkte werden ermittelt, indem das Bild von unten nach oben mittels so genannten *Scanlines* [5, Kapitel 4.1.5] durchlaufen wird. Für jede *Scanline* wird eine Wertung mitgeführt, welche mit 0 initialisiert wird und bei grünen Pixeln erhöht wird und bei nicht-grünen Pixeln verringert wird. Diese Wertung wird durch einen größeren Fehler verringert, wenn der Pixel – projiziert in Feld-Koordinaten – einen zu großen Abstand zum Roboter hat. Dadurch wird verhindert, dass anliegende Felder für falsche Erkennungen sorgen.

Die Segmentierung des Horizonts ist in vielerlei Hinsicht eng verwandt zu der Problemstellung, den Feldrand eines Spielfeldes in der *SPL* zu segmentieren. Auch hier wird eine Gerade oder ein anderes Modell in einer Datenmenge gesucht. Der Horizont liegt, wie der Feldrand, fast immer horizontal im Bild und reicht von einem Bildrand bis zum anderen.

Sergiy Fefilatyev et al. von der *University of South Florida, USA* [6] detektierten den Horizont mittels *Machine Learning*-Algorithmen. Die Klassifikatoren *Support*

Vector Machine (SVM), *J48* und *Naive Bayes* wurden aufgrund ihrer Performance miteinander verglichen. Dabei wurden die Klassifikatoren dafür verwendet, das Bild in *Himmel-Punkte* und *Boden-Punkte* zu klassifizieren. Die letztendliche Bestimmung des Horizonts erfolgte durch einen *Expectation Maximization*-Algorithmus. Mit *SVM* konnte somit eine Genauigkeit von 98,19% erreicht werden.

Damien Dusha et al. [7] von der *Queensland University of Technology, Australia* nutzten einen detektierten Horizont um die Rotation eines unbemannten Flugobjekts (*UAV*) zu berechnen. Der Horizont wurde dabei mittels *Morphologie* und der *Hough-Transformation* ermittelt. Anschließend wurde dieser mit Hilfe eines *Extended Kalman Filters* (*EKF*) über mehrere Bilder geschätzt.

Touqeer Ahmad et al. [8] verfolgten ebenfalls einen *Machine Learning*-Ansatz, um den Horizont zu detektieren. Hier wurde zu Beginn ein Kantenbild mit dem *Canny edge detector* errechnet. Anschließend wurden mittels *SIFT* lokale Merkmale für jeden Kantenpunkt berechnet – nachdem Kantenpunkte, deren Werte unter einem Schwellwert liegen, verworfen wurden. Mittels dieser lokalen Merkmale pro Pixel wurde eine *SVM* trainiert, die die Pixel in *Horizont* und *nicht Horizont* unterteilt.

Nasim Sepehri Boroujeni et al. [9] haben ebenfalls den Horizont detektiert, um die Navigation eines unbemannten Flugobjekts zu verbessern. Sie nutzten dabei aus, dass auf Bildern oftmals ein starker Gradient in der Lichtstärke um den Horizont zu finden ist. Um diesen Gradienten zu detektieren wird ein *Clustering*-Algorithmus angewandt. Genauer werden die Algorithmen *k-Means* und *Intensity-Based Clustering* angewandt und miteinander verglichen. Dabei stellte sich heraus, dass das *Intensity-Based Clustering* um ein vielfaches schneller ist. Bei 160 Testbildern konnte somit eine Fehlerrate von 1,2% erreicht werden.

Obwohl *Machine Learning*-Algorithmen in den vorgestellten Arbeiten gute Ergebnisse erzielt haben, sind sie für dieses Anwendungsgebiet leider nicht praktikabel. Wie in Kapitel 3.2 beschrieben, verfügt der hier eingesetzte *NAO* über eine sehr schwache Hardware. Hinzu kommt die Anforderung, dass die benutzten Algorithmen in Echtzeit laufen müssen, um in einem richtigen Spiel einsetzbar zu sein. Alleine schon die Berechnung eines Kantenbildes mit dem *Canny Edge Detector* [8] dauert auf Grund der limitierten CPU so lange, dass beinahe keine Zeit mehr für weitere Berechnungen bleibt. Aufgrund dessen wurde von solchen Algorithmen abgesehen. Der Ansatz, der auf den folgenden Seiten beschrieben wird, orientiert sich somit an der Arbeit von *Yongbo Qian und Daniel D. Lee* [3], welche im selben Anwendungsgebiet entwickelt wurde und somit ihre Realisierbarkeit schon unter Beweis gestellt hat.

3. Hintergrund

Hier wird eine Übersicht über den Kontext gegeben, in dem diese Arbeit entstanden ist.

3.1 RoboCup

Der *Robot World Cup* (kurz: *RoboCup*) ist ein jährlich ausgetragener Robotikwettbewerb, bei dem Teams aus der ganzen Welt gegeneinander antreten. Neben dem Roboterfußball gibt es weitere Ligen, wie *RoboCup Rescue*, *RoboCup@Home*, *RoboCup Industrial* und *RoboCupJunior*. Gegründet wurde der *RoboCup* 1992 in Tokio. Das Hauptziel ist, Mitte des 21. Jahrhunderts mit einem Team von rein autonomen, humanoiden Robotern den amtierenden FIFA Fußball-Weltmeister unter offiziellen FIFA-Spielregeln zu schlagen. Damit bietet der *RoboCup* einen Anreiz, um die KI-Forschung voran zu bringen. Außerdem wird der Austausch von wissenschaftlichen Fortschritten in den Gebieten der KI und der Robotik gefördert, weil viele Teams Teile ihres Codes offen zur Verfügung stellen. Neben dem offiziellen *RoboCup* gibt es auch kleinere Wettbewerbe, wie z.B. die *German Open* oder die *Japan Open* [10].

3.2 Standard Platform League und NAO

Die *Standard Platform League* (auch *SPL* genannt) ist eine Liga im *RoboCup*, bei der jedes Team mit dem selben Roboter antritt. Die Roboter agieren dabei voll autonom, das heißt, dass es keinerlei Steuerung von außen während der Spiele gibt. Seit 2009 ist der *NAO* von der Firma *Softbank Robotics* (ehemals Aldebaran) der einzige zugelassene Roboter für die *SPL*. Davor wurde mit dem *AIBO* von der Firma *Sony* gespielt [11]. Der *NAO* ist ein 58 cm hoher humanoider Roboter, welcher seit 2006 von der Firma *Softbank Robotics* entwickelt wird (siehe Abbildung 3.1). Dieser Roboter verfügt über einen 1,6 GHz Intel Atom Prozessor und einen Gigabyte RAM. Das Betriebssystem ist Linux. Über das Framework *NAOqi* haben Entwickler Zugriff auf die verschiedenen Sensoren und Gelenke mit insgesamt 25 Freiheitsgraden. Bei den Sensoren handelt es sich um Fußdruck-Sensoren, Inertial-Sensoren (Accelerometer und Gyroskop), Ultraschall und Infrarot. Außerdem verfügt der *NAO* über zwei Kameras, von denen sich eine im Mund befindet und eine zweite auf Stirnhöhe (siehe

Abbildung 3.2). Über die untere Kamera können somit Objekte im Nahbereich erfasst werden, wohingegen die obere Kamera Objekte in großer und mittlerer Weite erfasst. Die Kameras verfügen über eine Auflösung von 1280×960 und können maximal 30 Bilder pro Sekunde liefern [12, 13].

Abbildung 3.1: NAO-Roboter [13]

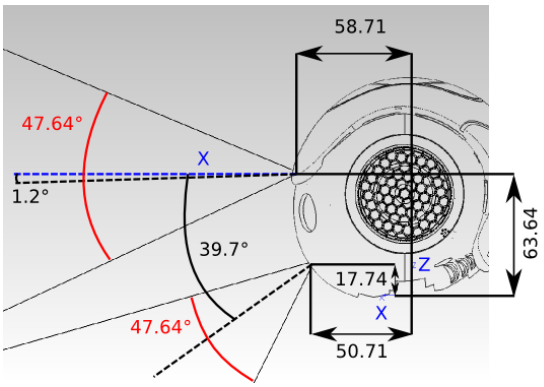


Abbildung 3.2: NAO-Kameras [14]

3.3 B-Human

B-Human ist ein Projekt des Fachbereichs 3 - Mathematik und Informatik der Universität Bremen und des Forschungsbereichs *Cyber-Physical Systems* des *DFKI*. Das Team nimmt seit 2008 an RoboCup-Turnieren in der *SPL* (siehe Kapitel 3.2) teil und konnte bislang achtmal die *German Open* und sechsmal die Weltmeisterschaft gewinnen [15] (siehe Abbildung 3.3).

Abbildung 3.3: Das Team von B-Human nach dem Sieg der RoboCup German Open 2017 [5]

4. Grundlagen

In diesem Kapitel werden einige Grundlagen erläutert, welche wichtig für das vollständige Verständnis der Arbeit sind.

4.1 Feldrand

Der Feldrand (oder auch: Feldgrenze) ist der äußerste Rand des Spielfeldes. Wie in Abbildung 4.1 zu sehen ist, kann der unverzerrte Feldrand durch vier orthogonal aufeinander stehende Geraden definiert werden. Ein Roboter hat nie den gesamten Feldrand im Bild, sondern nur einen Ausschnitt. In Abbildung 4.3 ist eine beispielhafte Detektion zu sehen. Bei der Form ist zu beachten, dass die Geraden nur in Roboter- und Feldkoordinaten orthogonal aufeinander stehen. In Bildkoordinaten wird die Optik dadurch verzerrt, dass auf eine zweidimensionale Ebene abgebildet wird – wie auch in den Abbildungen 4.2 und 4.3 zu sehen ist. Bei dieser Abbildung werden Geraden allerdings erhalten, weshalb die Berechnungen aus Kapitel 5 in Bildkoordinaten durchgeführt werden können.

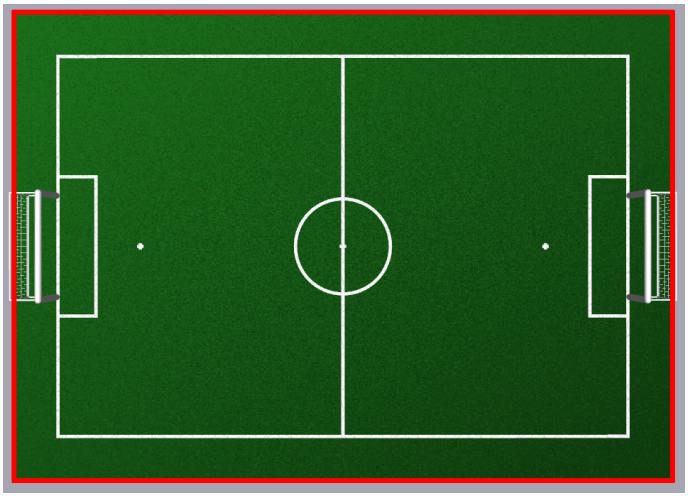


Abbildung 4.1: Spielfeld mit rot markierter Feldgrenze

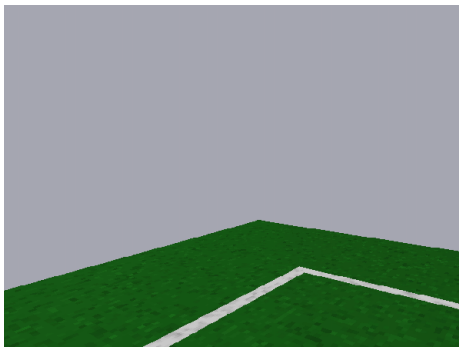


Abbildung 4.2: Feldgrenze im Bild

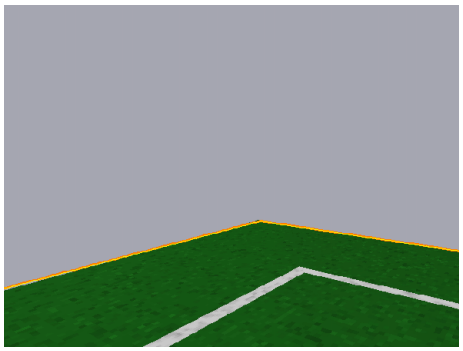


Abbildung 4.3: Detektierte Feldgrenze

4.2 RANSAC

Random Sample Consensus (oder auch: *RANSAC*) ist ein Algorithmus zur Parameterschätzung von Modellen in Eingabedaten. Im Gegensatz zu anderen Algorithmen zur Parameterschätzung, nutzt *RANSAC* nur die kleinst mögliche Anzahl von Daten um iterativ die Modellparameter zu bestimmen und dann zu berechnen, wie gut das errechnete Modell in die Daten passt, um anschließend gegebenenfalls eine Optimierung durchzuführen (siehe Abbildung 4.4). Diese Optimierung kann z.B. durch *Least Squares* stattfinden.

Algorithmus 1: *RANSAC*
1. Gegeben ein Modell, das minimal n Datenpunkte braucht um die freien Parameter zu bestimmen, und eine Menge von Daten P, sodass die Anzahl der Daten in P größer als n ist ($\#(P) > n$), bestimme zufällig eine Teilmenge $S1$ mit n Datenpunkten von P und berechne das Modell $M1$. Benutze das Modell $M1$ um eine Teilmenge $S1^*$ mit Punkten von P zu bestimmen, welche in einem gewissen Toleranzwert σ liegen. Die Teilmenge $S1^*$ heißt *Consensus Set* von $S1$.
2. Wenn $\#(S1^*)$ größer als ein gewisser Grenzwert t ist, benutze $S1^*$ um ein besseres Modell $M1^*$ zu berechnen.
3. Wenn $\#(S1^*)$ kleiner als t ist, wähle zufällig eine neue Teilmenge $S2$ und wiederhole den obigen Prozess. Wenn nach einer vordefinierten Anzahl von Iterationen N kein geeignetes *Consensus Set* gefunden wurde, berechne entweder das bisher größte Modell oder terminiere und melde Misserfolg.

Abbildung 4.4: *RANSAC* Pseudocode [16]

Der Algorithmus benötigt drei Parameter, welche vorher spezifiziert werden müssen. Diese sind die maximale Anzahl von Iterationen N, der Toleranzwert σ, welcher stark anwendungsspezifisch ist und entweder analytisch oder heuristisch bestimmt werden kann, und der Grenzwert t. Der Grenzwert t ist die Grundlage, um zu bestimmen, ob ein Modell einen Datensatz genügend gut repräsentiert. Wenn ein Algorithmus zur Optimierung des Modells angewendet werden soll, muss t außerdem so gewählt werden, dass genügend viele Daten zur Verfügung stehen. Eine Annäherung an die optimale Anzahl von Iterationen N kann über die Formel

$$N = \frac{1}{b} = w^{-n} \tag{4.1}$$

berechnet werden, wobei w die Wahrscheinlichkeit ist, dass ein beliebiger Datenpunkt im Toleranzbereich eines guten Modells liegt, n die Anzahl der benötigten Datenpunkte, um das Modell zu initialisieren und $b = w^n$ [16]. Dieser Wert wird üblicherweise noch verdoppelt oder verdreifacht, wenn vorher kein geeignetes Modell gefunden wurde.

Als letzte Optimierung für gefundene Modelle wird der Algorithmus *Least Squares* (Methode der kleinsten Quadrate) angewandt. Die Geradenparameter werden hier über den quadratischen Abstand aller relevanten Punkte optimiert. Der Algorithmus steht über das *B-Human*-Framework (siehe Kapitel 4.5) zur Verfügung.

4.3 Hessesche Normalform

Eine bessere Lösung als die klassische Form $y = kx + d$ ist die *Hessesche Normalform*

$$\vec{x} \cdot \vec{n} = d, \tag{4.2}$$

wobei $\vec{x}$ ein Ortsvektor ist, $\vec{n}$ eine normalisierte, orientierte *Normale*, welche senkrecht auf der Geraden steht und d der Abstand der Geraden vom Ursprung $(0, 0)$ (siehe Abb. 4.5). Eine Gerade G besteht also aus allen Ortsvektoren $\vec{x}$, die die obige Formel oder die Formel

$$\vec{x} \cdot \vec{n} - d = 0 \tag{4.3}$$

erfüllen. Um den Abstand $d(\vec{p}, G)$ eines Punktes $\vec{p}$ zu einer Geraden G zu berechnen, muss $\vec{p}$ lediglich in die obige Formel eingesetzt werden [17]:

$$\vec{p} \cdot \vec{n} - d = d(\vec{p}, G) \tag{4.4}$$

Anders als die klassische Form, kann die *Hessesche Normalform* auch vertikale Geraden darstellen. Außerdem ist die Berechnung des Abstandes eines Punktes von einer Geraden in *Hessescher Normalform* wesentlich günstiger in der Berechnung, als es bei der klassischen Form der Fall ist.

Über das *B-Human*-Framework steht die Bibliothek *Eigen* zur Verfügung. Eine Gerade in der *Hesseschen Normalform* steht dort unter dem Namen *Hyperplane* zur Verfügung. Eine *Hyperplane* benötigt zum Initialisieren eine normalisierte, orientierte *Normale* und einen Punkt, welcher auf der Geraden liegt [18]. Es reichen also genau zwei Punkte, um eine Gerade in der *Hesseschen Normalform* im *B-Human*-Framework zu initialisieren.

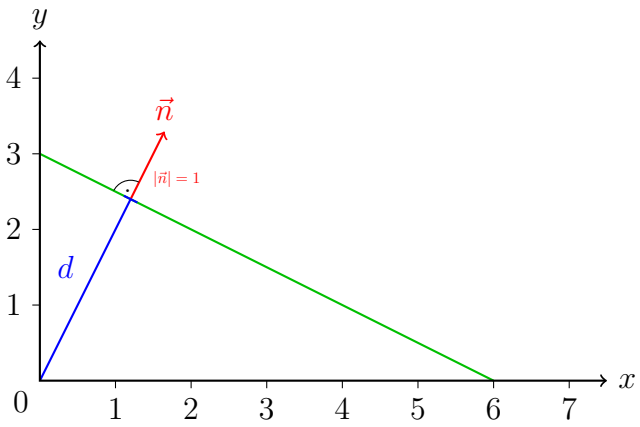


Abbildung 4.5: Hessesche Normalform Beispiel

4.4 Koordinatensysteme

Wenn Punkte dargestellt werden sollen, muss dieses immer in Relation zu einem Koordinatensystem passieren. Ein Koordinatensystem A in einem dreidimensionalen Raum besteht aus drei Achsen A_x, A_y, A_z und einem Ursprung A_W. Die Achsen stehen orthogonal aufeinander und bestehen aus freien Vektoren der Länge 1. Der Ursprung ist ein Punktvektor (siehe Abb. 4.6) [19].

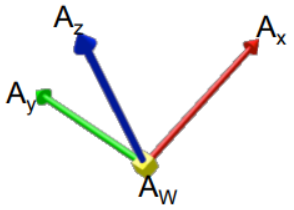


Abbildung 4.6: Beispiel für ein Koordinatensystem [19]

A kann als Matrix dargestellt werden, sodass für einen freien Vektor p und dessen vier Koordinaten p^A gilt:

$$p = (p^A)_x \cdot A_x + (p^A)_y \cdot A_y + (p^A)_z \cdot A_z + (p^A)_W \cdot A_W \tag{4.5}$$

p^A ist also die Darstellung von p relativ zum Koordinatensystem A. Wenn man nun zwei Koordinatensysteme A und B hat, kann eine Abbildung $T_{A \leftarrow B}$ konstruiert werden, sodass für einen Punkt p^B gilt:

$$p^A = T_{A \leftarrow B} \cdot p^B \tag{4.6}$$

Diese Abbildung wird im folgenden auch Transformation genannt.

4.5 B-Human Framework

Das *B-Human Framework* bietet verschiedene Werkzeuge, um Software für den *NAO* zu entwickeln. Das *Framework* wird seit vielen Jahren von dem *SPL*-Team *B-Human* entwickelt.

4.5.1 Modul-Framework und SimRobot

Das Herzstück ist der Simulator *SimRobot*, welcher die Möglichkeit bietet, Roboter in dreidimensionalen Räumen zu simulieren [20]. Außerdem kann man sich über diesen mit *NAO*-Robotern verbinden und direkt mit ihnen kommunizieren. Die vielen *Features* zum Debuggen waren auch bei der Entwicklung dieser Arbeit sehr nützlich. *B-Human* baut auf ein *Modul-Framework* auf, bei dem *Module* so genannte *Repräsentationen* bereitstellen und voraussetzen können. Wird eine *Repräsentation* von einem *Modul* vorausgesetzt, kann dieses die Daten aus der *Repräsentation* nutzen. Ein anderes *Modul* muss die *Repräsentation* mit Daten füllen, bevor diese benutzt werden kann. Über die *Repräsentationen* können also Daten im System kommuniziert werden. Das *Modul*, das im Laufe dieser Arbeit entstanden ist, wurde in diesem Framework entwickelt.

4.5.2 Koordinatensysteme in B-Human

In *B-Human* kann man mit verschiedenen Koordinatensystemen rechnen.

Bild-Koordinatensystem: In diesem Koordinatensystem werden üblicherweise Bildverarbeitungsalgorithmen ausgeführt, um z.B. Objekte in Bildern zu segmentieren. Die Berechnungen aus Kapitel 5 finden auch in diesem Koordinatensystem statt. Der Ursprung $(0, 0)$ liegt dabei links oben im Bild. Rechts unten befindet sich der Punkt $(h - 1, w - 1)$, wobei h die Höhe des Bildes ist und w die Breite. Die x-Achse zeigt also vom Ursprung nach rechts und die y-Achse zeigt vom Ursprung nach unten (siehe Abb. 4.7).

Roboter-Koordinatensystem: Dieses dreidimensionale Koordinatensystem beschreibt Feld-Koordinaten relativ zum Roboter. Der Ursprung liegt auf dem Boden zwischen den Füßen des Roboters, wobei die x-Achse vom Roboter aus nach vorne zeigt, die y-Achse nach links und die z-Achse nach oben. Zwischen diesem Koordinatensystem und dem Bild-Koordinatensystem kann man beliebig transformieren, wobei darauf geachtet werden muss, dass bei einer Transformation von Bild-Koordinaten nach Roboter-Koordinaten nur auf eine Ebene projiziert werden kann. Das heißt, dass für die z-Komponente stets ein Wert von 0 angenommen wird (siehe Abb. 4.8).

Feld-Koordinatensystem: Dieses dreidimensionale Koordinatensystem beschreibt Feldkoordinaten relativ zum Mittelpunkt des Spielfeldes. Der Ursprung liegt also auf dem Mittelpunkt, wobei die x-Achse vom eigenen Tor aus nach vorne zeigt, die y-Achse nach links und die z-Achse nach oben. Zu diesem Koordinatensystem kann man nur transformieren, wenn man die Position und Rotation des Roboters auf dem Spielfeld kennt (siehe Abb. 4.9).

Die Transformationen, um zwischen den verschiedenen Koordinatensystemen zu wechseln, werden von dem *B-Human*-Framework bereitgestellt.

Abbildung 4.7: Achsen des Bild-Koordinatensystems

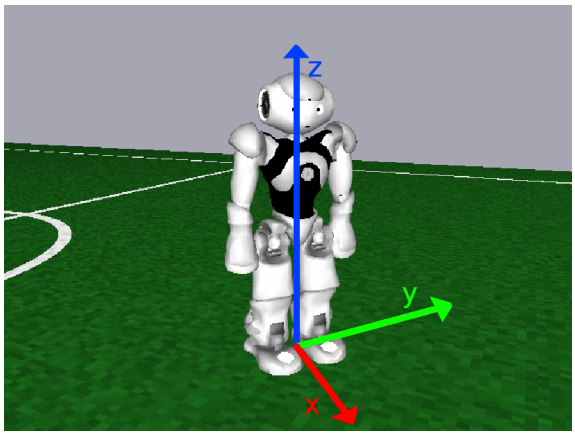


Abbildung 4.8: Achsen des Roboter-Koordinatensystems

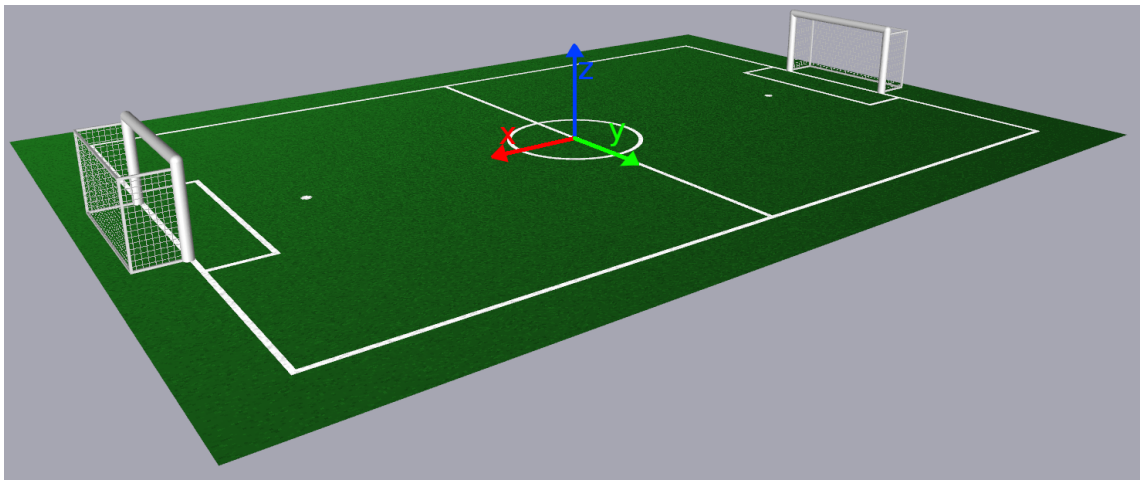


Abbildung 4.9: Achsen des Feld-Koordinatensystems

5. Umsetzung

In diesem Kapitel wird eine detaillierte Beschreibung der verwendeten Algorithmen gegeben. Nach einem Überblick über das Gesamtsystem werden die einzelnen Bestandteile ausführlich beschrieben.

5.1 Idee und Überblick

Wie schon in Kapitel 1 erwähnt, liegt die Idee darin, passende Geradenparameter für eine Menge von Punkten zu finden. Die daraus resultierenden Geraden werden im folgenden auch Modelle genannt. Die dafür benötigten Punkte werden so segmentiert, dass sie möglichst nah am sichtbaren Feldrand liegen. Bei der Suche der Modelle kann man sich zunutze machen, dass der unverzerrte Feldrand, wie in Abbildung 4.1 zu sehen, durch vier orthogonal aufeinander stehende Geraden beschrieben werden kann. Wenn nur ein Feldrand ohne Ecke im aktuellen Bild zu sehen ist, wird das längste Modell mit den wenigsten Punkten über ihm gesucht. Wenn eine oder mehrere Ecken im Bild sind, werden die Modelle gesucht, welche den besten Winkel – optimalerweise also genau 90° – zueinander aufweisen. Eine Ausführung des *Moduls* lässt sich in die folgenden Schritte unterteilen:

1. Suche möglicher Feldrand-Punkte: Hier werden Punkte auf dem Feldrand gesucht, aus denen später Modelle errechnet werden können.
2. Geraden-Findung: In diesem Schritt werden Geraden in den vorher gefundenen Feldrand-Punkten gesucht und bewertet.
3. Merken guter Punkte: Hier werden die gefundenen Modelle intern zusammen mit älteren Modellen gespeichert.

Wie schon in Kapitel 4 beschrieben, bilden *Module* und *Repräsentationen* die Grundlage für das *B-Human*-Framework. Im Laufe der Bearbeitung dieses Themas fiel die Entscheidung, eine neue *Repräsentation* in das System zu integrieren, welche die Daten enthalten soll, die mit diesem *Modul* berechnet werden. Wie genau die Ein- und Ausgaben des *Moduls* und der *Repräsentation* aussehen, wird im folgenden Abschnitt beschrieben.

5.2 Ein- und Ausgaben

Bei den Eingaben handelt es sich in diesem Fall um die *Repräsentationen*, welche im neuen *Modul* vorausgesetzt werden. Die *CameraInfo* enthält die intrinsischen Kamera-Parameter und Daten, welche Kamera das aktuelle Bild zurückgeliefert hat. Hier finden sich auch unter anderem die Brennweite, der Öffnungswinkel und die Höhe sowie die Breite des aktuellen Bildes. Zusammen mit der *CameraMatrix* können über das *B-Human*-Framework Transformationen zwischen den Koordinatensystemen (siehe Kapitel 4.5.2) durchgeführt werden. Eine *Repräsentation* namens *FrameInfo* wird benötigt, um auszurechnen, wie viel Zeit seit einem Zeitpunkt vergangen ist. Außerdem kann man über diese *Repräsentation* den aktuellen Zeitstempel des Systems abrufen, welcher ab Start des Systems inkrementiert wird. *ImageCoordinateSystem* ermöglicht Transformationen auf Bild-Koordinaten. Mit dieser *Repräsentation* können z.B. Verzerrungen von der Kamera auf eine Koordinate aufgerechnet werden. Über eine *Repräsentation* namens *Odometer* stehen sämtliche *Odometrie-Daten* zur Verfügung. Die *Odometrie* ist die Differenz zwischen aktueller Position und Rotation in Relation zur Position und Relation im letzten Durchgang. Das Bild steht über *ECImage* als Graustufenbild mit dem Wertebereich $[0, 255]$ und als farbsegmentiertes Bild bereit, bei dem jeder Pixel einer der Klassen *keine*, *weiss*, *schwarz* oder *feld* zugeordnet ist.

Die Ausgaben erfolgen ebenfalls in Form einer *Repräsentation*. Diese enthält die Punkte der letzten erkannten Feldgrenze in Bild-Koordinaten und in Koordinaten relativ zum Roboter.

Wie die Daten für die *Repräsentation* berechnet werden, wird in den folgenden Kapiteln beschrieben.

5.3 Suche möglicher Feldrand-Punkte

Um Geraden im aktuellen Bild detektieren zu können, müssen zuerst Punkte segmentiert werden. Der hier verwendete Algorithmus ähnelt dem von *Yongbo Qian et.al* [8]. Das Farbsegmentierte Bild (*ECImage*) wird von unten nach oben in einem gewissen Abstand x durchlaufen. Der Horizont wird in diesem Falle als oberste Grenze verwendet – diese steht durch die Benutzung des *B-Human*-Frameworks bereit. Für jeden Pixel wird nun ein *Score* mitgeführt, welcher um 1 verringert wird, wenn es sich um die Farbklasse *feld* handelt und ansonsten wird er um 1 erhöht. Der Pixel jeder Iteration mit dem geringsten *Score* wird als Feldrand-Punkt gewählt. x muss dabei so gewählt werden, dass auch kleinere Geraden bei Feldecken noch segmentiert werden können. Um Laufzeit zu sparen, wurde der Abstand in y-Richtung ebenfalls parametrisiert. Diese kleinen Optimierungen kann man hier anwenden, da die spätere Optimierung über *Least Squares* auch mit kleinen Abweichungen der Punkte in der Höhe gut funktioniert. Abbildung 5.1 zeigt die Suche von Feldrand-Punkten. Die abgetasteten Pixel wurden in Abbildung 5.1a rot markiert. Die gefundenen Punkte werden gelb in Abbildung 5.1b dargestellt.

5.4 Geraden-Findung

Das Ziel in diesem Abschnitt ist es, Geradenparameter zu finden, die möglichst gut in eine übergebene Menge von Punkten in Bild-Koordinaten passen.

(a) Suche von Punkten (b) Gefundene Feldrand-Punkte

Abbildung 5.1: Suche von Feldrand-Punkten

5.4.1 Ansatz

Hier kann man sich die Form des Feldrandes zu nutze machen (siehe Kapitel 4.1). Da ein Roboter mit der Kamera immer nur einen Ausschnitt des Feldrandes erfasst, kann man zwischen den folgenden Fällen unterscheiden (siehe auch Abb. 5.2):

- Es ist nur eine Gerade im Bild: Dies ist der häufigste, aber auch einfachste Fall. Hier muss nur die beste Gerade gefunden werden, welche in die übergebenen Punkte passt.
- Es ist eine Ecke im Bild: Dies ist der zweit häufigste Fall. Hier müssen zwei Geraden gefunden werden, welche die kleinste Abweichung zu einem 90° Winkel aufweisen.
- Es sind zwei Ecken im Bild, die durch eine Gerade verbunden werden: Dies ist der seltenste Fall. Hier müssen drei Geraden gefunden werden, wobei anliegende Geraden einen 90° Winkel und gegenüberliegende Geraden einen 180° Winkel aufweisen müssen.

Es hat sich bewährt, die Fälle nacheinander abzuarbeiten. Im Detail heißt das, dass zuerst eine erste Gerade gesucht wird – wird diese gefunden, wird eine zweite gesucht, welche orthogonal auf der ersten steht. Wird auch eine zweite gefunden, wird eine dritte gesucht, die entweder orthogonal auf der ersten oder der zweiten Gerade steht und zu der anderen einen 180° Winkel aufweist. Weitere Geraden werden außerdem nur gesucht, wenn noch genügend Punkte zur Verfügung stehen, welche noch nicht mit einem vorherigen Modell in Verbindung gebracht wurden.

Dabei ist zu beachten, dass in einem Zyklus bei der ersten Suche die gesamten segmentierten *Feldrand-Punkte* aus dem aktuellen Bild mit den Punkten der Modelle aus den letzten Bildern verwendet werden (siehe auch Kapitel 5.5). In den darauf folgenden Suchen werden allerdings nur noch die Punkte verwendet, welche in den vorherigen Durchgängen nicht in ein *Consensus Set* gehörten – also so genannte *Ausreißer* waren. Dadurch soll verhindert werden, dass die gleichen Modelle mehrmals gefunden werden.

5.4.2 RANSAC

Dieser Abschnitt beschreibt die Umsetzung des *RANSAC*-Algorithmus' (siehe Kapitel 4.2), über den iterativ Geraden in einer Menge von Punkten gefunden werden können.

Wähle zufällige Punkte aus

Hier werden zufällig zwei Punkte aus den gegebenen Punkten gewählt, mit denen später das Modell berechnet werden soll. Um zu verhindern, dass schlechte Punkte mehrmals verwendet werden, wird ohne Zurücklegen gewählt. Die Anzahl der Punkte ergibt sich aus der minimalen Anzahl, die vorhanden sein muss, um das Modell zu berechnen. Folglich kann dieser Schritt nur ausgeführt werden, wenn mehr als zwei Punkte vorhanden sind (siehe Kapitel 4.3). Die Punkte werden dabei abwechselnd aus dem vorderen, mittleren und hinteren Bereich gewählt, um bei maximal drei möglichen Geraden alle erfassen zu können. Die Punkte müssen dafür nach ihren x-Koordinaten sortiert sein.

Berechne das Modell mit den gewählten Punkten

Aus den gewählten Punkten wird nun ein Modell berechnet. Das Modell ist in diesem Falle eine Gerade, welche in der *Hesseschen Normalform* vorliegt (siehe Kapitel 4.3). Bei der Erstellung der Modelle muss darauf geachtet werden, dass die *Normalen* konsistent in die selbe Richtung zeigen. In Bildkoordinaten werden alle *Normalen* in entgegengesetzter Richtung des Roboters gedreht, um in einem der folgenden Schritte die Fehler richtig berechnen zu können. Um das Modell zu erstellen, wird eine *Normale* $\vec{n}$ der Länge 1 benötigt. Diese wird aus den zwei Ortsvektoren berechnet, welche im vorherigen Schritt gewählt wurden – hier als $\vec{a}$ und $\vec{b}$ dargestellt. Die Normalisierung der *Normalen* auf die Länge 1 erfolgt durch eine Multiplikation mit $\frac{1}{|\vec{n}|}$, welche hier durch *norm* dargestellt wird. $\vec{a}_x$ bzw. $\vec{b}_x$ beschreibt die x-Koordinate des jeweiligen Punktes. Bevor die *Normale* ausgerechnet werden kann, muss eine Richtung der Geraden bestimmt werden:

$$\vec{dir} = \begin{cases} \text{norm}(\vec{a} - \vec{b}) & \text{wenn } \vec{a}_x < \vec{b}_x \\ \text{norm}(\vec{b} - \vec{a}) & \text{sonst} \end{cases} \tag{5.1}$$

Damit die *Normale* die richtige Richtung hat, muss sie um 90° gegen den Uhrzeigersinn gedreht werden. Rotationen von Vektoren gegen den Uhrzeigersinn um einen Winkel θ können durch eine Rotationsmatrix *rot* verwirklicht werden:

$$rot_\theta = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \tag{5.2}$$

Die *Normale* $\vec{n}$ wird dann also wie folgt berechnet:

$$\vec{n} = \text{rot}_{90} \cdot \vec{dir} \tag{5.3}$$

In *Eigen* [18] kann die Gerade nun mittels der *Normalen* und eines Punktes auf der Geraden berechnet werden, z.B. a.

Bestimmung des *Consensus Set*

Hier wird das in Kapitel 4.2 erwähnte *Consensus Set* berechnet. Dieses enthält Punkte aus den übergebenen Punkten, welche einen gewissen Abstand σ zum berechneten Modell unterschreiten – dieses also so gesehen unterstützen. Der Abstand wird hier wie in Formel 4.3 berechnet. Die Ausreißer – also Punkte welche den Abstand σ überschreiten – werden auch gespeichert, da diese zum Suchen weiterer Modelle benötigt werden. Das *Consensus Set* enthält also alle Punkte $\vec{p}$ aus den übergebenen Punkten, welche die folgende Formel erfüllen:

$$d(\vec{p}, G) <= \sigma \tag{5.4}$$

Ist der Abstand kleiner als 0, wird negiert, da hier nur der absolute Wert wichtig ist.

Bewertung des Modells und des *Consensus Set*

Zu Beginn wird der Fehler vom Modell berechnet (siehe Kapitel 5.4.4). Ein Modell kann nur zugelassen werden, wenn der Fehler kleiner ist, als der des bisher besten Modells. Außerdem wurde für den Fehler eine Obergrenze eingebaut, um sehr schlechte Modelle zu verwerfen. Wie in Kapitel 4.2 beschrieben, muss die Anzahl der Punkte im *Consensus Set* einen Schwellwert t überschreiten. Für den Fall, dass mehrere Modelle gesucht werden, werden außerdem die Winkel zwischen den letzten gefundenen Modellen (M_1, M_2) und dem Modell der aktuellen Iteration M_0 verglichen. Wenn ein zweites Modell gesucht wird – also ein M_1 aus einer letzten Suche bereits vorhanden ist, wird auf einen 90°-Winkel getestet. Wenn es die dritte Suche ist – also zwei Modelle M_1 und M_2 bereits vorhanden sind, muss das neue Modell M_0 zu einem der Modelle einen 90°-Winkel aufweisen und zu dem anderen einen 180°-Winkel. Für einen Überblick über die möglichen Winkel zwischen Modellen (siehe Abbildung 5.2). Den Winkel zwischen zwei Modellen A und B kann man als Winkel zwischen ihren Normalen definieren:

$$\alpha_{A,B} = acos(A_{normal} \cdot B_{normal}), \tag{5.5}$$

wobei A_{normal} bzw. B_{normal} die *Normale* des jeweiligen Modells ist und "$\cdot$" das *Skalarprodukt*. Die Winkel-Berechnungen werden dabei mit einem Toleranzwert durchgeführt, in dem die Modelle trotzdem zugelassen werden.

5.4.3 Optimale Anzahl der Iterationen

Wie schon in Kapitel 4.2 erwähnt, kann man diesen Parameter mittels Formel 4.1 berechnen. Wenn man als w von einer Wahrscheinlichkeit von 50% ausgeht, und bei einer Gerade in *Hessescher Normalform* gilt $n = 2$, ergibt sich für die Anzahl der Iterationen $N = \frac{1}{0{,}5^2} = 4{,}0$. Nach Fischler [16] sollte man den errechneten Wert nochmal verdoppeln oder verdreifachen. Hier wird dieser auf 12 verdreifacht, wodurch sich die endgültige Anzahl für die Iterationen ergibt: $N = 12$. Dieser Wert ist allerdings stark gekoppelt an den Algorithmus, welcher bei der Wahl der Punkte angewendet wird. Wählt man die Punkte rein zufällig, sind bis zu 20 Iterationen nötig, um ein gutes Modell zu finden.

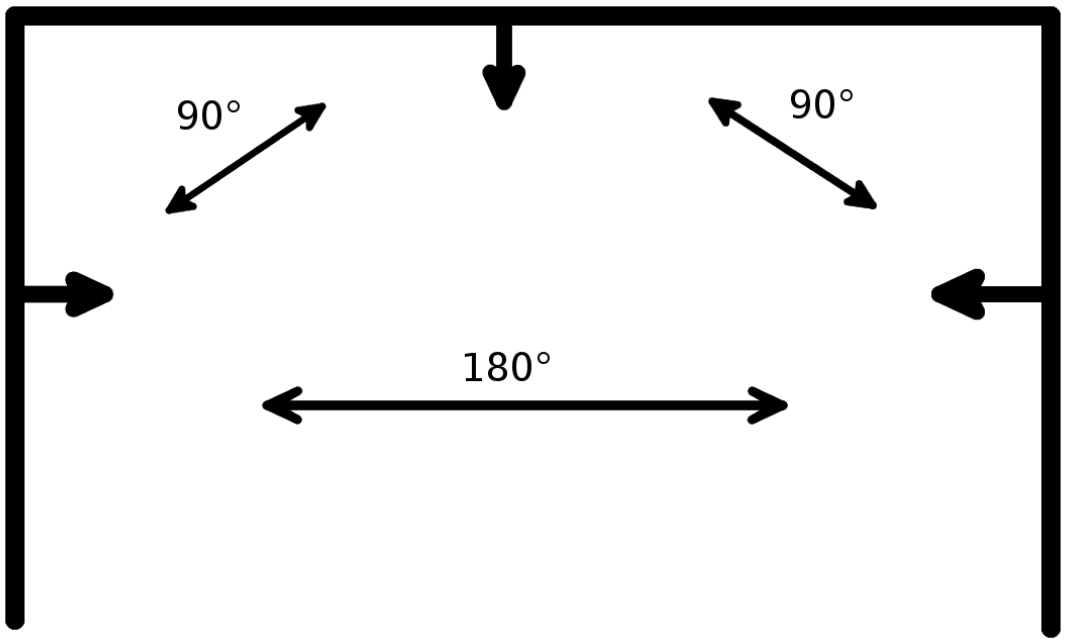


Abbildung 5.2: Winkel zwischen mehreren Feldgrenzen im Bild

5.4.4 Auswahl der Fehlerfunktion

Die Fehlerfunktion gilt als Hauptkriterium, um die Güte eines Modells zu bewerten. Wichtig ist, dass Punkte über der Linie für einen größeren Fehler sorgen, da die Gerade gesucht wird, welche die wenigsten Punkte über ihr aufweist. Dafür wird ein extra Fehler σ mit aufgerechnet. Für Punkte, welche unter der Geraden liegen, wird ein konstanter Fehler e aufgerechnet um zu verhindern, dass Punkte am unteren Bildrand den Fehler zu stark beeinflussen. Die Fehlerfunktion für ein Modell M und eine Menge von Ortsvektoren X mit der Länge n sieht also wie folgt aus:

$$E(X, M) = \sum_{i=1}^{n} \begin{cases} \sigma * \text{dis}(X_i, M) & \text{wenn dis}(X_i, M) > 0 \\ e & \text{sonst} \end{cases} \tag{5.6}$$

5.4.5 Berechnen der Punkte auf den Modellen

Abschließend müssen Punkte auf den Geraden berechnet werden, weil diese in anderen *Modulen* verwendet werden. Um diese zu berechnen, wird das Bild von links nach rechts in x-Richtung durchlaufen. Dabei werden die Schnittpunkte aller Modelle mit einer Geraden berechnet, welche den Ursprung $(x, 0)$ hat und genau vertikal im Bild steht. Der Schnittpunkt mit der größten y-Koordinate ist dann jeweils der Feldrand-Punkt für diese x-Koordinate. Ein Beispiel findet sich in Abbildung 5.3 und 5.4. Die gewählten Punkte werden außerdem über das *B-Human*-Framework in Roboter-Koordinaten transformiert. Die Punkte werden anschließend in die *Repräsentation* geschrieben, welche das *Modul* bereitstellt, damit sie für andere *Module* zur Verfügung stehen.

Abbildung 5.5 zeigt eine Feldgrenze, welche durch den in diesem Kapitel beschriebenen Algorithmus detektiert wurde.

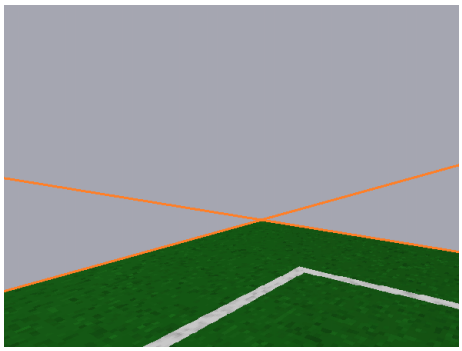


Abbildung 5.3: Mehrere Modelle im Bild

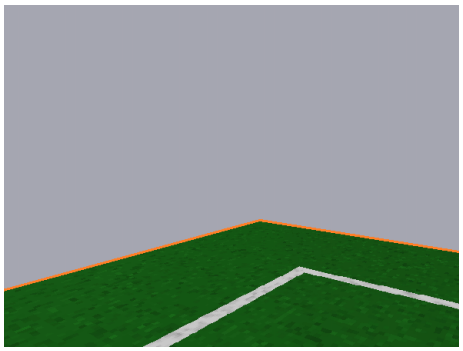


Abbildung 5.4: Die berechneten Punkte

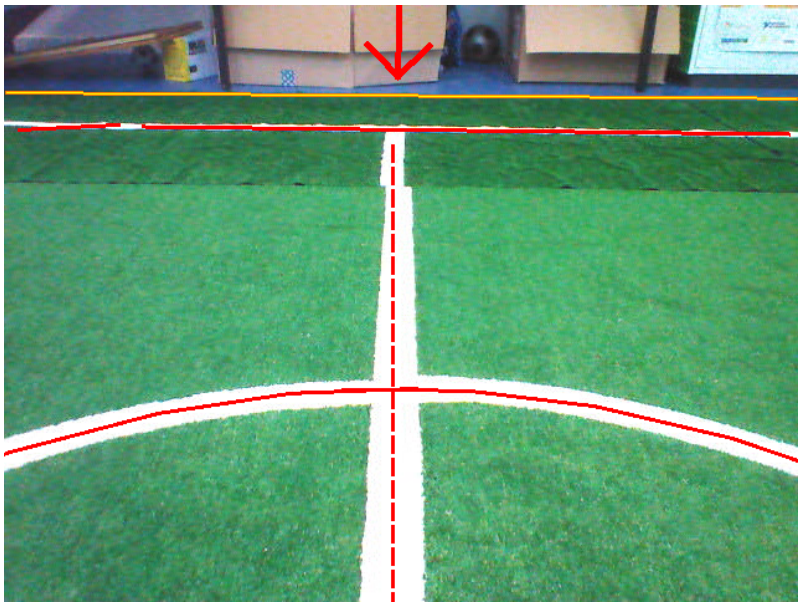


Abbildung 5.5: Eine detektierte Feldgrenze (gelb). Feldlinien sind in rot dargestellt. Der rote Pfeil ist Teil des Horizonts, welcher in das Bild gezeichnet wurde.

5.5 Tracking

Das Ziel des Trackings ist es, Feldgrenzen und Feldecken zu detektieren, welche sich schon nicht mehr im Bild befinden. Da das restliche System darauf ausgelegt ist, Geraden in einer Menge von Punkten (in Bildkoordinaten) zu suchen und es keinen Unterschied macht, ob sich die Punkte tatsächlich im sichtbaren Bereich befinden[1], müssen lediglich Punkte aus den letzten Erkennungen gespeichert werden, mit denen die alten Modelle wieder detektiert werden können. Zu diesem Zwecke werden die Punkte aus dem *Consensus Set* der letzten erkannten Modelle gespeichert. Der Container für die Punkte wird im folgenden Algorithmus *trackedPoints* genannt. Dieser enthält entweder Punkte aus den vorherigen Durchgängen oder ist leer ($\emptyset$).

[1] Also $x \geq 0$ und $y \geq 0$ für Bild-Koordinaten muss nicht gegeben sein.

Bei der Auswahl der Punkte wird wie folgt vorgegangen:

```
Data: trackedPoints
Result: trackedPoints updated with current points
if trackedPoints = ∅ then
    write all points from every Consensus Set to trackedPoints;
else
    calculate the maximum and minimum x value from trackedPoints in
     image coordinates;
    for every model m do
        for every point p in the Consensus Set of m do
            if the x coordinate of p is less than the minimum value or greater
             than the maximum value of trackedPoints then
                add p to trackedPoints;
            end
        end
    end
end
```

Algorithm 1: Tracking von Modellen

Abbildung 5.6 zeigt einige Punkte, welche von vorherigen Geraden gespeichert wurden. Diese können sich auch außerhalb des aktuellen Bildes befinden, was die Erkennung von Geraden und Ecken ermöglicht, obwohl diese nicht im Bild zu sehen sind.

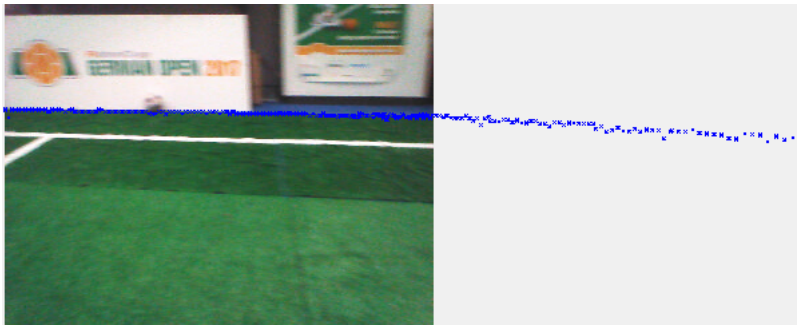


Abbildung 5.6: Gespeicherte Punkte außerhalb des aktuellen Bildes

Die Punkte werden dabei zusammen mit einem Zeitstempel (aus *FrameInfo*) gespeichert und nach einer festgelegten Zeit wieder entfernt. Der daraus resultierende Container wird an den vorher beschriebenen Algorithmus – zusammen mit den möglichen Feldrand-Punkten aus dem aktuellen Bild – zur Geraden-Suche übergeben (siehe 5.4).

6. Evaluation

In den folgenden Experimenten wird die Robustheit des neu entwickelten Moduls getestet. Zu diesem Zweck wird der Roboter auf einer bekannten Feldposition platziert und in Richtung der Feldbegrenzung gedreht (siehe z.B. Abb. 6.1a und 6.1b). Die – in Feldkoordinaten projizierten – detektierten Punkte auf der Feldgrenze werden anschließend über einen längeren Zeitraum kontrolliert und die Abweichung relativ zu den erwarteten Positionen wird berechnet. Über die *Repräsentation FieldDimensions* stehen die erwarteten x und y Positionen der Feldgrenze zur Verfügung. Es sind allerdings nur Abweichungen in der Höhe relevant, weshalb immer nur einer der Werte für die Berechnungen benötigt wird. Nach dem Feld-Koordinatensystem (siehe Kapitel 4.4) sind an den Feldseiten also die y-Koordinaten relevant um Abweichungen nach oben und unten zu berechnen, da die y-Achse in diese Richtung zeigt. An den Seiten, wo sich die Tore befinden, sind die x-Koordinaten relevant. Die z-Komponenten der Feld-Koordinaten werden als nicht relevant angesehen, da diese in diesem Fall keinen Informationsgewinn bieten. Um einen Maßstab zu haben, werden die selben Experimente auch mit der aktuell in *B-Human* genutzten Feldgrenze durchgeführt (siehe [1, Kapitel 4.1.11]). Der Fehler wurde als Durchschnittswert der letzten n projizierten Punkte definiert:

$$E(X) = \frac{(\sum_{i=1}^{n} X_i - Y)}{n}, \tag{6.1}$$

wobei X hier die projizierten Punkte und Y die bekannte erwartete Position beschreibt (also entweder einen x oder einen y Wert). Es wird immer der absolute Wert genommen, damit sich positive und negative Werte nicht gegenseitig aufheben. Es werden jeweils 40 *Samples* in einem Abstand von 500 ms gesammelt – also über einen gesamten Zeitraum von 20 Sekunden. Die durchschnittlichen Modullaufzeiten für jedes Experiment werden ebenfalls notiert. Ein *Sample* beschreibt den Durchschnittswert aller vorherigen Punkte nach Gleichung 6.1 in einem maximalen zeitlichen Abstand von 500 ms. Als Parameter in den Experimenten wird der Abstand zur Feldgrenze und dessen Kopfbewegungen (geradeaus schauend, und abwechselnd nach links und rechts schauend) gewählt, um verschiedene Spielsituationen zu simulieren.

Die Ergebnisse werden in Graphen (z.B. 6.2) dargestellt. Die x-Achse beschreibt das momentane Sample (also $0 \leq x < 40$ für 40 Samples). Die y-Achse beschreibt

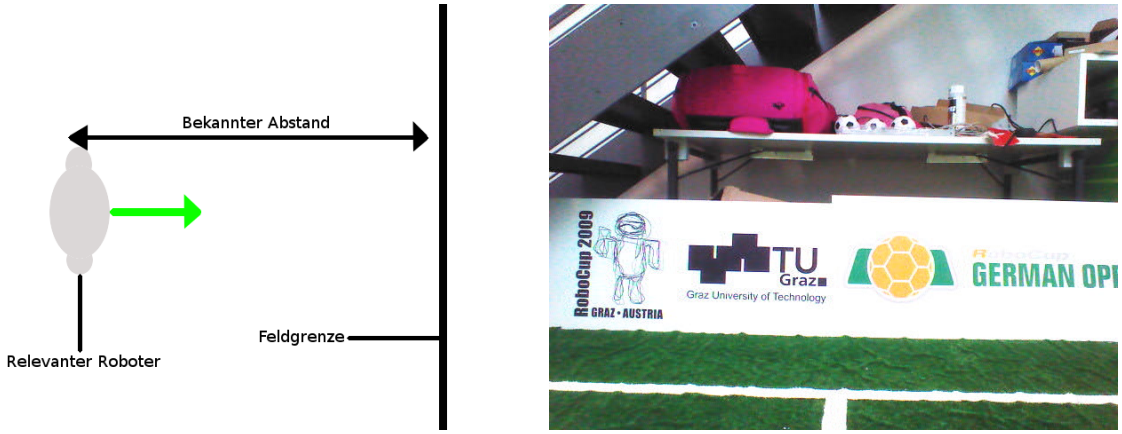
den Abstand zur realen Feldgrenze nach Gleichung 6.1. Ein kleiner Wert auf der y-Achse ist folglich besser, als ein Großer. Die Daten des alten Moduls werden in rot dargestellt und die Daten des neuen Moduls in blau.

6.1 Erkennung von Geraden

In den folgenden zwei Experimenten wird die Erkennung von Geraden getestet. In *Experiment 1* wird die Erkennung in einem statischen Bild getestet und in *Experiment 2* während einer aktiven Kopfbewegung. Der Roboter dreht seinen Kopf dabei immer wieder von links nach rechts. Dies sorgt für eine Bewegungsunschärfe, was die Erkennung stark beeinflussen kann. Bei dem neu entwickelten Modul wurde für diese Tests das *Tracking* deaktiviert. Es wurden also nur die im aktuellen Bild segmentierten Punkte verarbeitet.

6.1.1 Experiment 1

Der Roboter wird hier auf eine definierte Position auf dem Feld platziert und zur Feldgrenze gedreht. Die Position dieser ist bekannt. Die Abbildungen 6.1a und 6.1b zeigen den Versuchsaufbau exemplarisch für eine Position auf dem Mittelkreis. Die Feldgrenze ist dabei der Übergang zu den weißen Banden.

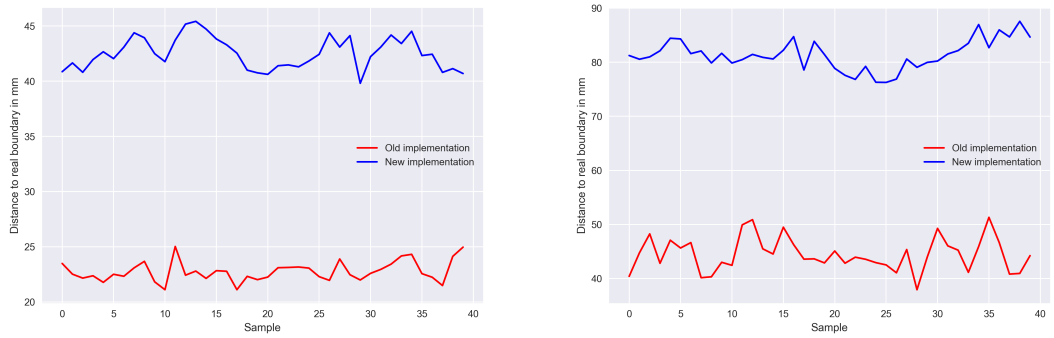


(a) Skizze des Aufbaus (b) Kamerabild

Abbildung 6.1: Versuchsaufbau für Experiment 1

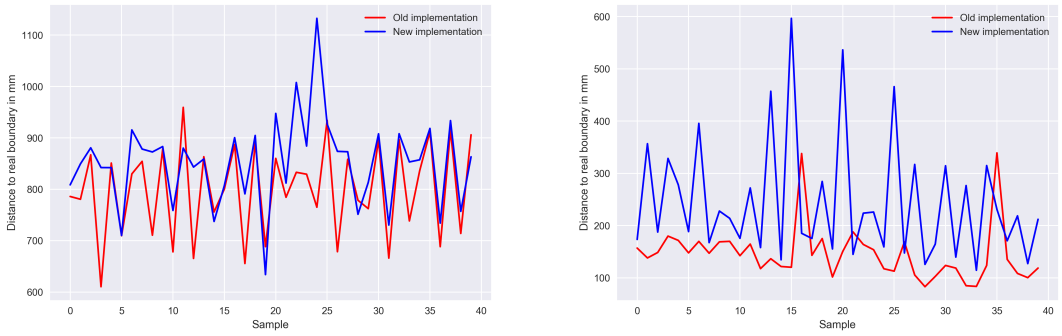
Die aufgenommenen Daten sind den Abbildungen 6.2 und 6.3 zu entnehmen. Dabei fällt sofort auf, dass das neue Modul in Abbildung 6.2 schlechtere Werte aufweist, als das Alte. Auch der Durchschnittswert (Tabelle 6.1) ist in allen Fällen folglich deutlich größer. Das lässt sich darauf zurückführen, dass Fehler bei einer Geraden nicht ausgeglichen werden können, indem sie einen Knick macht. Auch gibt es einen kleinen Spielraum bei der Suche der Geraden. Es kann durchaus mehrere Geraden geben, welche die gegebenen Datenpunkte beschreiben, und sich leicht in ihrem Winkel unterscheiden. Dieser kleine Fehler macht sich hier bemerkbar, wobei es hier um einen Fehler von 40 Millimetern bei mehreren hundert Punkten geht, was weniger als 0,5 Millimeter pro Punkt sind. Bei einer *Konvexen Hülle* kann es auch Ausreißer geben, aber wenn die anderen gewählten Punkte wieder genauer auf der Feldgrenze liegen, kann sie trotzdem sehr gute Ergebnisse in dieser Testreihe erzielen. Die Werte sind dabei bei einem Roboter auf dem Mittelkreis (Abbildung 6.2a) viel

kleiner als bei einem Roboter auf dem Mittelpunkt (Abbildung 6.2b). Das ist auf die Transformation von Kamera-Koordinaten in Bild-Koordinaten zurückzuführen, welche sich bei weiter entfernten Punkten viel stärker auf das Ergebnis auswirkt, da dort kleine Abweichungen schon zu großen Projektionsfehlern führen können. Abgesehen davon bewegen sich die Werte im kleinen Millimeter Bereich. Das neue Modul ist in beiden Fällen nicht so gut wie das Alte, aber trotzdem kann man die Werte nicht als Misserfolg deuten, da keine großen Ausreißer jeglicher Art vorhanden sind. Man kann die Erkennung von Geraden in einem statischen Bild also als robust ansehen.



(a) Auf Mittelkreis (b) Auf Mittelpunkt

Abbildung 6.2: Daten für Experiment 1 - Ohne Kopfbewegung



(a) Auf Mittelkreis (b) Auf Mittelpunkt

Abbildung 6.3: Daten für Experiment 1 - Mit Kopfbewegung

Wenn man sich Abbildung 6.3 ansieht, fällt auf, dass beide Module allgemein mit den selben Problemen zu kämpfen haben. Die Ausschläge in die positive Richtung sind die Zeitpunkte, zu denen eine Feldecke im Bild war. Bei einem Roboter, der auf dem Mittelkreis steht (siehe Abbildung 6.3a), verhalten sich beide Module sehr ähnlich, wobei das alte Modul allgemein bessere Werte liefert, was sich auch im Durchschnittswert (Tabelle 6.1) widerspiegelt. An einigen Stellen weisen beide Module den nahezu gleichen Fehler auf. Die starken Schwankungen sprechen dafür, dass bei Ecken wohl häufiger falsch erkannte Feldgrenzen auftreten, welche bei einem Roboter auf dem Mittelkreis bei beiden Modulen ähnlich schlimm ausfallen. Wenn man sich auf die Tiefpunkte konzentriert, also die Zeitpunkte, zu denen eine Gerade im Bild war,

liegen die beiden Module allerdings nicht allzu weit auseinander. Bei einem Roboter auf dem Mittelkreis liegen beide Module an diesen Punkten oftmals gleichauf (siehe Abbildung 6.3a). Das neue Modul neigt aber eher zu Ausreißern in der negativen Richtung, wie z.B. an der Stelle $x = 24$. Bei einem Roboter auf dem Mittelpunkt (siehe Abbildung 6.3b) verhält es sich ähnlich, wenngleich die Ausschläge des neuen Moduls hier viel stärker sind. An den Tiefpunkten weisen aber auch hier beide Module sehr ähnliche Werte auf. Das neue Modul ist bei beiden Experimenten langsamer gewesen (siehe Tabelle 6.2). Außerdem sollte erwähnt werden, dass in Abbildung 6.3 die Werte bei einem Roboter auf dem Mittelkreis viel kleiner sind, als bei einem Roboter auf dem Mittelpunkt. Auf den ersten Blick steht das im Widerspruch zu den Werten aus Graph 6.2, aber das Rauschen und die Projektionsfehler scheinen keine große Rolle mehr zu spielen, wenn genügend Punkte auf der Feldgrenze zur Verfügung stehen. Bei einer weiteren Entfernung ist folglich ein größerer Teil der Feldgrenze im Bild, wodurch Fehler leichter kompensiert werden können. Gerade dem *Least Squares* Algorithmus sollte das zugutekommen.

	Altes Modul	**Neues Modul**
Nach vorne schauend		
Auf Mittelkreis	22,7574	42,5355
Auf Mittelpunkt	44,4754	81,3599
Mit bewegendem Kopf		
Auf Mittelkreis	799,39	852,102
Auf Mittelpunkt	146,147	243,631

Tabelle 6.1: Durchschnittswerte für Experiment 1 in mm

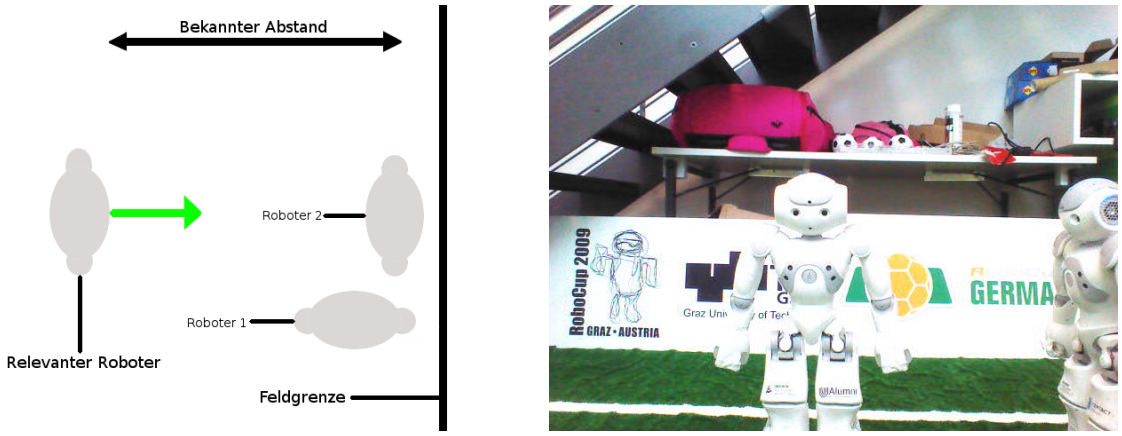
	Altes Modul	**Neues Modul**
Nach vorne schauend		
Auf Mittelkreis	0,2	0,8
Auf Mittelpunkt	0,22	0,78
Mit bewegendem Kopf		
Auf Mittelkreis	0,24	0,75
Auf Mittelpunkt	0,25	0,77

Tabelle 6.2: Modullaufzeiten für Experiment 1 in ms

6.1.2 Experiment 2

Der Roboter wird hier wie in *Experiment 1* platziert. Außerdem werden zwei Roboter so auf dem Feld platziert, dass sie wichtige Bereiche der Feldgrenze verdecken. Dies beinhält die rechte Seite und einen Großteil der Gerade. Die Abbildungen 6.4a und 6.4b zeigen den Versuchsaufbau exemplarisch für eine Position auf dem Mittelkreis. Die Feldgrenze ist dabei der Übergang zu den weißen Banden.

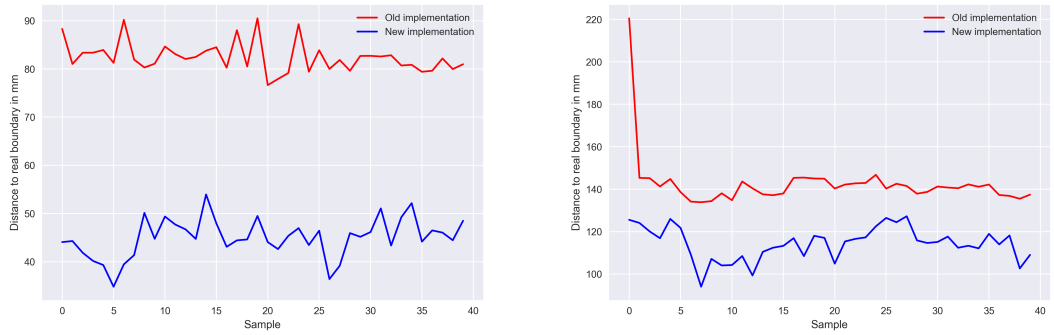
Die aufgenommenen Daten sind den Abbildungen 6.5 und 6.6 zu entnehmen. Wenn man sich Abbildung 6.5 ansieht, fällt gleich auf, dass das neue Modul hier besser abschneidet als das Alte. Das ist darauf zurückzuführen, dass das alte Modul bei Robotern am Bildrand erhebliche Probleme hat, wobei sie bei dem neuen Modul kein großes Problem darstellen. Auch Roboter in der Bildmitte sind kein großes Problem.



(a) Skizze des Aufbaus (b) Kamerabild

Abbildung 6.4: Versuchsaufbau für Experiment 2 - Ohne Kopfbewegung

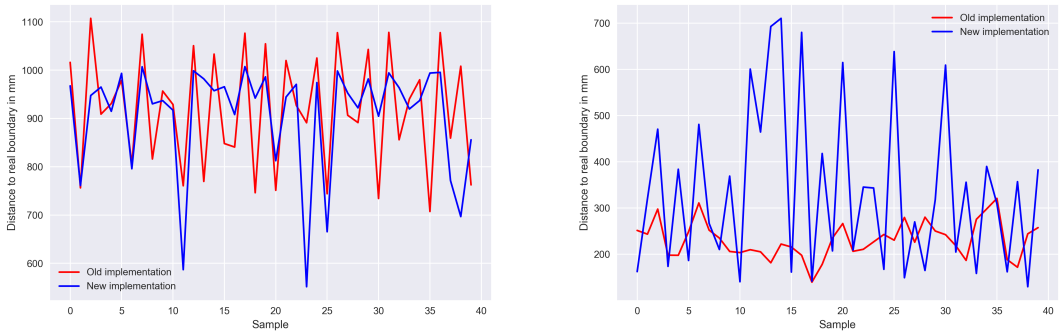
Folglich weist das neue Modul auch bei den Durchschnittswerten (siehe Tabelle 6.3) bessere Werte auf. Bei einem Roboter auf dem Mittelkreis ist es um den Faktor 2 besser. Bei einem Roboter auf dem Mittelpunkt ist das neue Modul immerhin noch um den Faktor 1,3 besser. Allerdings ist auch hier das neue Modul wieder langsamer als das Alte (siehe Tabelle 6.4).



(a) Auf Mittelkreis (b) Auf Mittelpunkt

Abbildung 6.5: Daten für Experiment 2 - Ohne Kopfbewegung

Bei Kopfbewegungen (siehe Abbildung 6.6) leidet das neue Modul unter den selben Problemen wie im vorherigen Experiment. Bei einem Roboter auf dem Mittelkreis fallen diese allerdings in Relation zum alten Modul nicht allzu stark auf (siehe Abbildung 6.6a). Hier schneidet es mit einem Durchschnittswert von 906,81 sogar leicht besser ab als das Alte mit 918,393 (siehe Tabelle 6.3). Aber trotz allem handelt es sich dabei um einen sehr großen Fehler. Bei einem Roboter auf dem Mittelpunkt (siehe Abbildung 6.6b) wird die Problematik mit den Feldecken wieder deutlich. Hier ist der Durchschnittswert folglich viel schlechter. Bei der Modullaufzeit (siehe Tabelle 6.4) hat sich nicht viel geändert. Das neue Modul ist durchgehend langsamer, als das Alte. Schaut man sich aber nur die Tiefpunkte an, ist das neue Modul fast immer genau so gut und manchmal sogar besser, als das Alte. Die großen Probleme bei den Feldecken trüben dabei die eigentlich guten Werte bei den Geraden.



(a) Auf Mittelkreis (b) Auf Mittelpunkt

Abbildung 6.6: Daten für Experiment 2 - Mit Kopfbewegung

	Altes Modul	**Neues Modul**
Nach vorne schauend		
Auf Mittelkreis	82,4226	44,9828
Auf Mittelpunkt	142,482	114,39
Mit bewegendem Kopf		
Auf Mittelkreis	918,393	906,81
Auf Mittelpunkt	231,071	337,807

Tabelle 6.3: Durchschnittswerte für Experiment 2 in mm

	Altes Modul	**Neues Modul**
Nach vorne schauend		
Auf Mittelkreis	0,23	0,65
Auf Mittelpunkt	0,22	0,74
Mit bewegendem Kopf		
Auf Mittelkreis	0,23	0,77
Auf Mittelpunkt	0,24	0,78

Tabelle 6.4: Modullaufzeiten für Experiment 2 in ms

Abschließend für die zwei durchgeführten Experimente lässt sich sagen, dass die Geraden-Erkennung des neu entwickelten Moduls bei Hindernissen im Bild robuster ist, als die des alten Moduls. Die Ergebnisse aus *Experiment 1* bewegen sich in einem so kleinen Bereich, dass die Unterschiede beinahe schon vernachlässigbar sind. In *Experiment 2* wurde die Robustheit bei Hindernissen im Bild nachgewiesen. Hier trübten die Fehler bei der Ecken-Erkennung die eigentlich guten Ergebnisse.

6.2 Erkennung von Feldecken

In den folgenden Experimenten wird die Erkennung von Ecken getestet. In *Experiment 3* wird die Erkennung anhand eines statischen Bildes getestet. In *Experiment 4* wurden Hindernisse in Form von Robotern hinzugefügt. Auch diese Experimente wurden ein mal mit Kopfbewegungen und ein mal ohne durchgeführt. Bei dem neu entwickelten Modul wurde das *Tracking* wieder deaktiviert.

6.2.1 Experiment 3

Hier wurde der Roboter ähnlich wie in *Experiment 1* (6.1.1) vor der Feldgrenze auf einer bekannten Position platziert. Anders als in *Experiment 1* handelt es sich aber dieses mal um eine Feldecke. Abbildung 6.7 zeigt ein Beispielbild für die kurze Entfernung, für die ein Roboter circa 1,70 m von der Feldecke entfernt platziert wurde. Für die zweite Testreihe wurde der Roboter auf einer Entfernung von circa 2,50 m platziert.

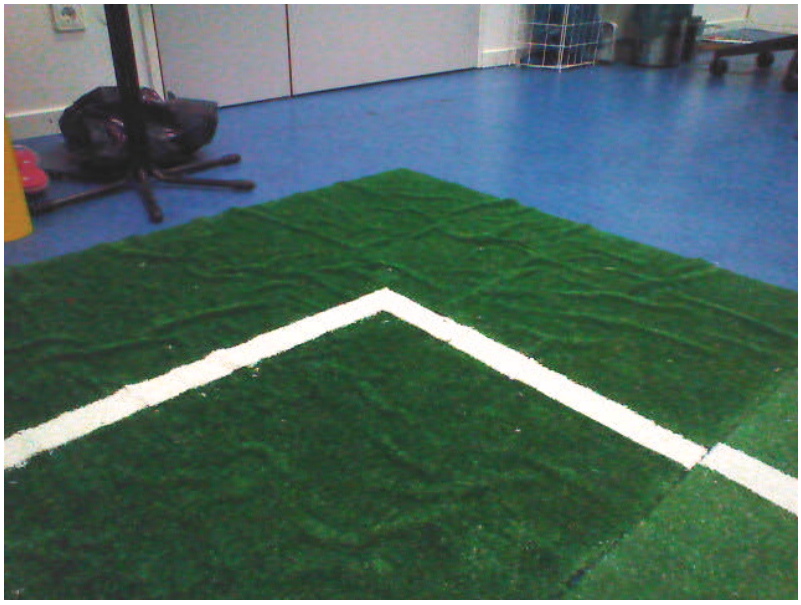
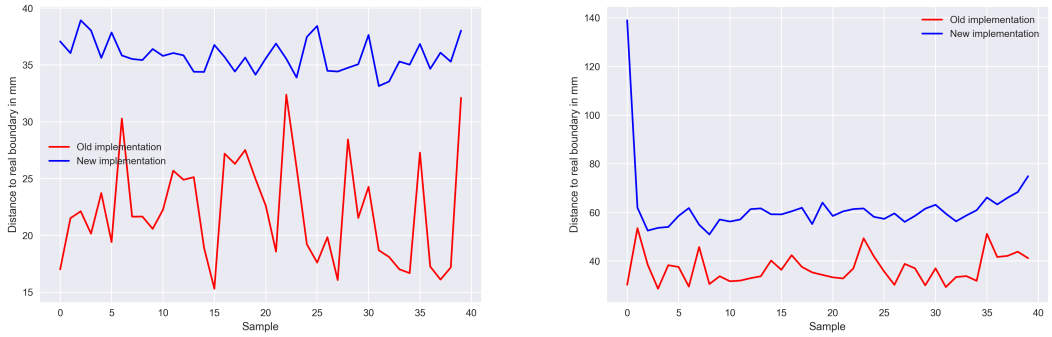


Abbildung 6.7: Beispielbild für Experiment 3 - Ohne Kopfbewegung

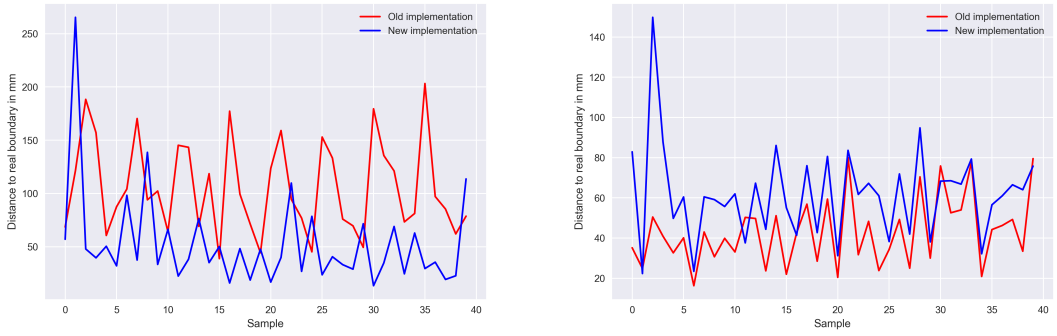
Die aufgenommenen Daten sind den Abbildungen 6.8 und 6.9 zu entnehmen. Wenn man sich Abbildung 6.8 ansieht, fällt auf, dass das neue Modul in diesem Experiment wieder anscheinend schlechter abschneidet, als das Alte. Was aber auch auffällt, ist, dass die Fehler des neuen Moduls deutlich konstanter sind. Der höhere Fehler lässt sich wieder auf den Spielraum bei der Geraden-Findung zurückführen. Zu Beginn in Graph 6.8b ist ein Ausschlag in positiver Richtung bei den Daten des neuen Moduls vorhanden. Anscheinend konnte an dieser Stelle durch eine schlechte erste Gerade keine gute zweite Gerade gefunden werden, oder die zweite hatte keinen optimalen Winkel, was zu einem größeren Fehler führt. Trotz allem kann man die Erkennung des neuen Moduls hier als besser einstufen, da die Abstände in den Fehlern von 20 mm vernachlässigbar sind, und in diesem Falle konstante Werte wichtiger sind, als ein geringer Fehler. Eine große Varianz in den Daten steht auch für große Änderungen in der Erkennung der Feldgrenze. Diese Änderungen wirken sich auch auf Module aus, welche die Feldgrenze verwenden, da hier deswegen gegebenenfalls Spezialfälle behandelt werden müssen. Folglich möchte man eine möglichst kleine Varianz haben.



(a) Auf kleiner Entfernung (b) Auf großer Entfernung

Abbildung 6.8: Daten für Experiment 3 - Ohne Kopfbewegung

Dieses Bild führt sich auch in Abbildung 6.9a fort. Abgesehen von einem stärkeren Ausschlag zu Beginn ist der Fehler des neuen Moduls hier beinahe durchgängig geringer, als der des alten Moduls. Außerdem sind die Fehler des neuen Moduls durchgehend konstanter, was für wenige Falsch-Erkennungen spricht. Der Fehler des alten Moduls variiert hingegen durchgehend sehr stark. Diese Beobachtungen spiegeln sich auch in den Durchschnittswerten wieder (siehe Tabelle 6.5). Auf größerer Entfernung (siehe Abbildung 6.9b) liefern beide Module die nahezu selben Fehler. Trotz der guten Ergebnisse ist die Modullaufzeit des neuen Moduls leider größer, als die des Alten (siehe Tabelle 6.6).



(a) Auf kleiner Entfernung (b) Auf großer Entfernung

Abbildung 6.9: Daten für Experiment 3 - Mit Kopfbewegung

Zusammenfassend für diese Testreihe kann man sagen, dass beide Module Feldecken gut erkennen. Die Erkennung des neu entwickelten Moduls kann man aber als leicht besser ansehen, da die Fehlerwerte durchgehend konstanter waren. Falscherkennungen sorgen allerdings für größere Fehler, wie z.B. Graph 6.8b zeigt. Diese halten sich allerdings in Grenzen.

	Altes Modul	Neues Modul
Nach vorne schauend		
Kleine Entfernung	22,0804	35,7995
Große Entfernung	36,8588	61,7789
Mit bewegendem Kopf		
Kleine Entfernung	105,584	52,8459
Große Entfernung	42,9445	61,8273

Tabelle 6.5: Durchschnittswerte für Experiment 3 in mm

	Altes Modul	Neues Modul
Nach vorne schauend		
Kleine Entfernung	0,3	0,9
Große Entfernung	0,39	0,89
Mit bewegendem Kopf		
Kleine Entfernung	0,22	0,5
Große Entfernung	0,23	0,78

Tabelle 6.6: Modullaufzeiten für Experiment 3 in ms

6.2.2 Experiment 4

Hier wurde der Roboter wie in *Experiment 3* (6.2.1) vor der Feldecke auf einer bekannten Position platziert. Als Hindernisse wurden Roboter so platziert, dass sie wichtige Bereiche der Feldecke abdecken, wie den Randbereich und die Spitze der Ecke. Abbildung 6.10 zeigt ein Beispielbild für einen Roboter, der circa 1,70 m von der Feldecke entfernt platziert wurde.

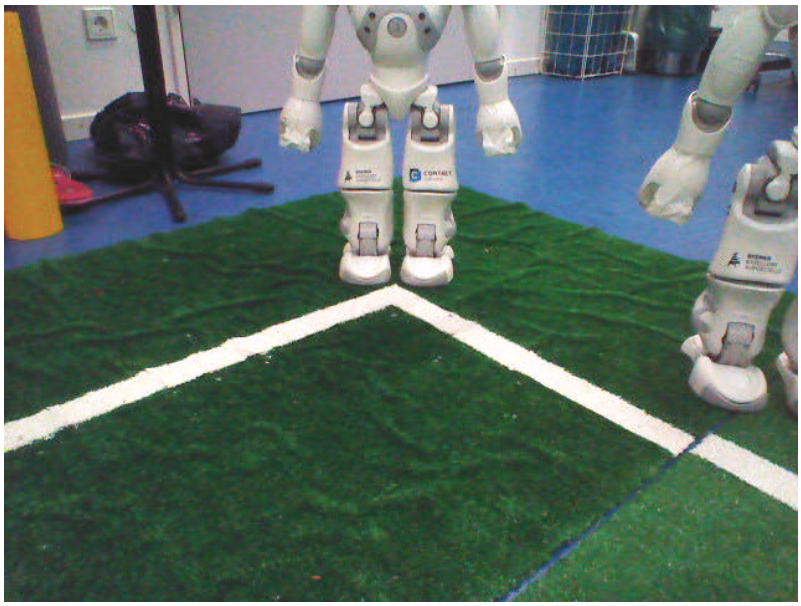
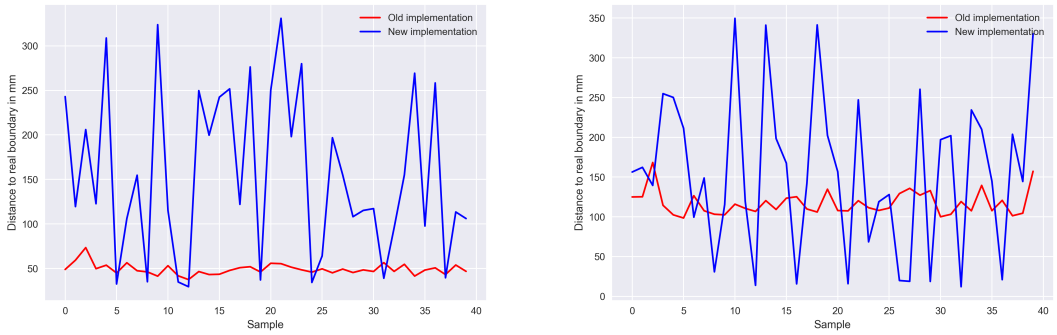


Abbildung 6.10: Beispielbild für Experiment 4 - Ohne Kopfbewegung

Die aufgenommenen Daten sind den Abbildungen 6.11 und 6.13 zu entnehmen. Bei Betrachtung von Abbildung 6.11a fällt auf, dass das neu entwickelte Modul durchweg einen größeren Fehler und auch eine viel größere Varianz aufweist. Das ist darauf

zurückzuführen, dass bei Feldecken manchmal keine zweite Gerade gefunden werden kann (siehe z.B. 6.12). Das kann passieren, wenn die erste Gerade einen ungünstigen Winkel aufweist und auf der anderen Seite der Feldecke nicht genügend Punkte vorhanden sind, um diesen Fehler aufzufangen. Werden zwei Geraden, und somit eine Feldecke, gefunden, weisen beide Module den beinahe selben Fehler auf. Die Fehlerkennungen sind in diesem Falle aber zu häufig aufgetreten. Abbildung 6.11b weist ein ähnliches Bild auf. Die Fehlerwerte des neuen Moduls variieren sehr stark. Wenn eine Feldecke detektiert wird (Tiefpunkte), ist der Fehler allerdings deutlich geringer, als der des alten Moduls. In Abbildung 6.12 wird deutlich, warum Falscherkennungen für so große Fehler sorgen. Weil es sich um eine Gerade handelt, welche erst am Bildrand endet, wird der Fehler unverhältnismäßig groß, da der Abstand zur realen Feldgrenze immer größer wird. Das alte Modul weist hier auch einen vergleichsweise großen Fehler auf, aber da diese auch eine kleine Varianz aufweist, kann man das alte Modul in diesem Falle als besser ansehen. Diese Beobachtungen spiegeln sich auch in den Durchschnittswerten wieder (Tabelle 6.7). Bei den Modullaufzeiten lieferte das neue Modul auch wieder deutlich schlechtere Werte (Tabelle 6.8).

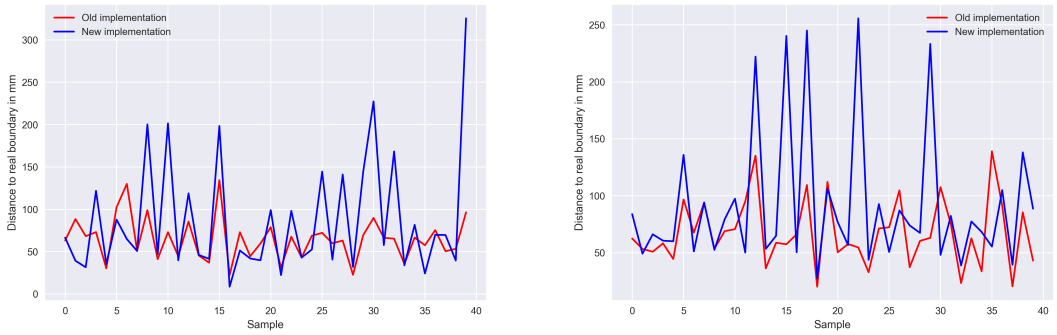


(a) Auf kleiner Entfernung (b) Auf großer Entfernung

Abbildung 6.11: Daten für Experiment 4 - Ohne Kopfbewegung

Abbildung 6.12: Experiment 4: Fehlerkennung bei Feldecke

Wenn nun Kopfbewegungen hinzukommen (Abbildung 6.13), stabilisiert sich das neue Modul ein wenig. Vor allem bei einer kurzen Entfernung (siehe Abbildung 6.13a) sind die berechneten Fehler nun wesentlich konstanter. Auch hier gibt es aber auch einige kleine Ausreißer. Bei einer größeren Entfernung sorgen die Fehlerkennungen aber



(a) Auf kleiner Entfernung (b) Auf großer Entfernung

Abbildung 6.13: Daten für Experiment 4 - Mit Kopfbewegung

wieder für zu große Fehler (siehe Abbildung 6.13b). In den meisten Fällen liefern beide Module hier aber die annähernd selben Fehler. Es ist allerdings nicht von Vorteil, wenn manchmal nur eine Feldgrenze detektiert wird, sodass die Erkennungsleistung vom alten Modul hier als besser eingestuft wird.

	Altes Modul	**Neues Modul**
Nach vorne schauend		
Kleine Entfernung	49,1984	155,865
Große Entfernung	117,17	155,291
Mit bewegendem Kopf		
Kleine Entfernung	65,0392	86,2397
Große Entfernung	67,4332	91,7414

Tabelle 6.7: Durchschnittswerte für Experiment 4 in mm

	Altes Modul	**Neues Modul**
Nach vorne schauend		
Kleine Entfernung	0,22	0,88
Große Entfernung	0,28	0,82
Mit bewegendem Kopf		
Kleine Entfernung	0,23	0,74
Große Entfernung	0,23	0,82

Tabelle 6.8: Modullaufzeiten für Experiment 4 in ms

Abschließend für diese Testreihe kann man sagen, dass die Feldecken-Erkennung des neuen Moduls darunter leidet, dass zu oft keine zweite Gerade gefunden werden kann. Das kann zum einen daran liegen, dass die erste Gerade einen ungünstigen Winkel aufweist und die verbliebenen Punkte auf der zweiten Feldgrenze diesen nicht auffangen können. Zum zweiten könnte es auch daran liegen, dass auf der zweiten Feldgrenze nicht genügend Punkte vorhanden sind, um eine gut passende Gerade zu finden. Die Feldecken-Erkennung des neuen Moduls mit Hindernissen im Bild ist also nach diesen Tests schlechter, als die des alten Moduls.

6.3 Tracking

In den vorherigen Tests hat sich herausgestellt, dass das neu entwickelte Modul Probleme bei kleinen Feldecken hat. Zu diesem Zwecke wurde das *Tracking* implementiert, welches relevante Punkte aus gefundenen Geraden über eine gewisse Zeit speichert. Bei Anwendung dieses *Trackings* müssten auch kleine Feldecken demnach segmentiert werden können, da mehr Punkte als nur die aus dem aktuellen Bild vorhanden sind.

6.3.1 Experiment 5

Um das *Tracking* zu testen, wurde der Roboter sehr ähnlich zu *Experiment 3* vor einer Feldecke platziert. Der Kopf dreht sich abwechselnd von links nach rechts. Der Test wird ein mal ohne und ein mal mit aktiviertem *Tracking* durchgeführt. Wenn das System funktioniert, müssten die Ergebnisse bei dem zweiten Durchlauf besser sein, als bei dem ersten. Anders als bei den vorangegangenen Tests wurde hier allerdings nicht der Abstand zu der realen Feldgrenze berechnet. Dieser Fehler ist hier nicht praktikabel, da auch schlecht passende Feldecken einen größeren Fehler erreichen können, als eine Gerade. In diesem Falle soll aber nur getestet werden, wie oft eine Feldecke gefunden wird. Zu diesem Zwecke wird in jedem Durchlauf des Moduls die Anzahl der gefundenen Geraden abgespeichert. Dies passiert über einen Zeitraum von 20 Sekunden. Da der Prozess, in dem das Modul läuft, mit 30 Herz läuft, kommt man also auf eine Anzahl von 600 Samples. Bei aktiviertem *Tracking* wurden die Punkte hier für 2 Sekunden gespeichert.

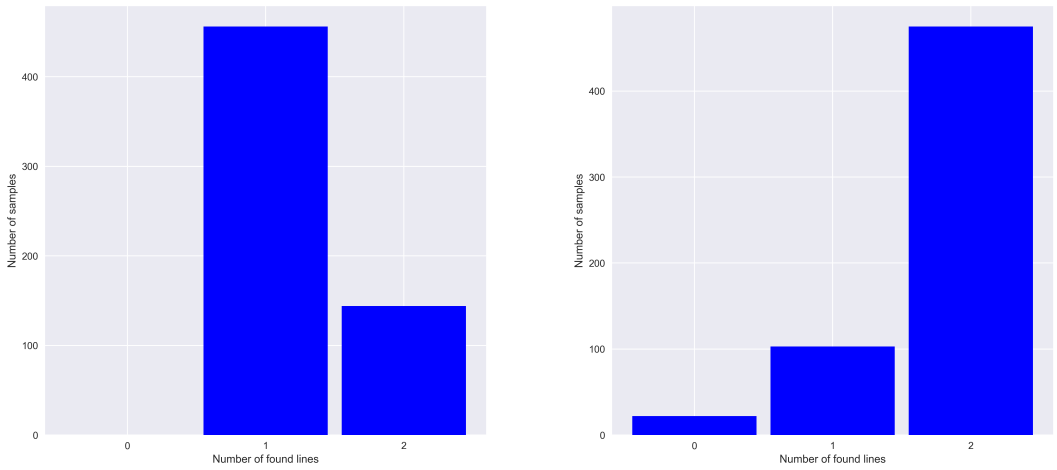
Die aufgenommenen Daten sind der Abbildung 6.14 zu entnehmen. Ohne *Tracking* wurde in den meisten Fällen nur eine Gerade gefunden, anstatt zwei. Genauer sind es 456 gegen 144 (siehe Tabelle 6.9).

Testdurchlauf	**0 Geraden**	**1 Gerade**	**2 Geraden**
Ohne Tracking	0	456	144
Mit Tracking	22	103	475

Tabelle 6.9: Daten für Experiment 5

Mit dem *Tracking* wurde 475 mal zwei Geraden gefunden. Also nur in 125 Fällen von insgesamt 600 wurden weniger als zwei Geraden gefunden, was man als Erfolg verbuchen kann. Zu beachten ist aber auch, dass in 22 Fällen keine Geraden gefunden werden konnte. Das lässt entweder auf einen Fehler in der Implementierung schließen, oder die Menge an Punkten war so groß, dass in der festgelegten Anzahl von Iterationen keine gute Gerade gefunden werden konnte. Dieses Problem könnte man vielleicht umgehen, indem man die Anzahl der Iterationen im *RANSAC*-Algorithmus an die Anzahl der zu überprüfenden Punkte anpasst. Wenn man die Anzahl der Samples miteinander vergleicht, in denen zwei Geraden, also eine Feldecke, gefunden wurden, stehen 457 vom Testdurchlauf mit dem *Tracking* gegen 144 vom Testdurchlauf ohne das *Tracking*. Das ist ein Gewinn von circa 55 %.

Ausgehend von den Graphen aus Abbildung 6.14 kann man also sagen, dass das *Tracking* sehr gut funktioniert und bei Spezialfällen die Erkennungsraten verbessern kann. Wenn man sich dann die Modullaufzeit ansieht, ist aber immer noch Luft nach oben (Tabelle 6.10). Die vielen Punkte, welche bei dem *Tracking* gespeichert



(a) Ohne Tracking (b) Mit Tracking

Abbildung 6.14: Daten für Experiment 5

werden, werden in jedem Moduldurchlauf von Feld-Koordinaten in Bild-Koordinaten transformiert, damit sie im *RANSAC*-Algorithmus verarbeitet werden können. Der *RANSAC*-Algorithmus durchläuft diese Punkte auch wieder mehrmals, was leider zu sehr langen Laufzeiten bei sehr vielen Punkten führen kann.

Ohne Kopfbewegung	**Mit Kopfbewegung**
0,74	3,0

Tabelle 6.10: Modullaufzeiten für Experiment 5 in ms

7. Fazit und Ausblick

Ziel dieser Arbeit war es, ein *Modul* im *B-Human*-Framework zu entwickeln, welches die Feldgrenze in einem Bild erkennen kann. Außerdem sollten in diesem *Modul* die letzten erkannten Feldgrenzen über einen gewissen Zeitintervall gespeichert werden, um Feldgrenzen und Ecken außerhalb des Bildes erkennen zu können. Die Erkennung sollte über den *RANSAC*-Algorithmus geschehen. Die Ausführung des gesamten *Moduls* sollte dabei nicht länger als zwei Millisekunden dauern.

Dieses *Modul* wurde im Laufe der Arbeit implementiert und der momentan in *B-Human* genutzten Feldgrenzen-Erkennung gegenübergestellt. Dabei hat sich herausgestellt, dass die Erkennung von Geraden bei Hindernissen im Bild robuster ist, als die des alten *Moduls*. Die Erkennung von Geraden ohne Hindernisse ist leicht schlechter. Dies lässt sich auf den *RANSAC*-Algorithmus zurückführen, bei dem es mehrere Lösungen für eine Menge von Punkten geben kann. Die Erkennung von Feldecken funktioniert eingeschränkt. In den Experimenten wurde nachgewiesen, dass das neue *Modul* Probleme bei der Erkennung von kurzen Geraden und somit auch kleinen Feldecken aufweist. Dieses Problem kann allerdings durch Benutzung des *Trackings* umgangen werden, wie das letzte Experiment gezeigt hat. Mit deaktiviertem *Tracking* wird die Modullaufzeit mit 0,7 ms der Zielsetzung gerecht. Mit aktiviertem *Tracking* konnte die Zielvorgabe von zwei Millisekunden allerdings nicht eingehalten werden. Grundsätzlich sollte das *Modul* wie in dieser Arbeit beschrieben aber einsetzbar sein. Der größte Vorteil gegenüber des momentan in *B-Human* genutzten *Moduls* ist die Robustheit bei der Geraden-Erkennung.

Um das *Modul* weiter zu verbessern, kann man verschiedene Ansätze probieren. Um Feldecken auch aus größerer Entfernung erkennen zu können, könnte man die *RANSAC*-Parameter relativ zu der Entfernung der Feldrand-Punkte zum Roboter skalieren. Der Abstand zu der Gerade für die Berechnung des *Consensus Set* sollte auf größerer Entfernung wesentlich kleiner sein. Auch sollte man den Grenzwert erhöhen, an dem man ein *Consensus Set* akzeptiert, da bei Feldgrenzen auf größerer Entfernung folglich auch mehr Feldrand-Punkte auf diesen liegen. Einige Ansätze könnte man außerdem wesentlich effizienter implementieren, wenn man *Single Instruction Multiple Data* (SIMD) nutzt. Auf diese Weise könnte man z.B. die Berechnung der Geraden-Punkte in Kapitel 5.4.5 parallelisieren, was eine sehr große Verringerung

der Rechenzeit mit sich bringen könnte. Außerdem könnte man die Odometrie besser einbeziehen, indem man auf Grundlage der letzten erkannten Feldgrenzen und der Odometrie nur nach einer bestimmten Anzahl von Geraden sucht. Das setzt allerdings voraus, dass die letzten Messungen verlässlich waren, da ansonsten nach der falschen Anzahl gesucht wird, was zu Fehlerkennungen führen kann.

Literaturverzeichnis

[1] Röfer, Thomas ; Laue, Tim ; Müller, Judith ; Bartsch, Michel ; Batram, Malte J. ; Böckmann, Arne ; Böschen, Martin ; Kroker, Martin ; Maass, Florian ; Münder, Thomas ; Steinbeck, Marcel ; Stolpmann, Andreas ; Taddiken, Simon ; Tsogias, Alexis ; Wenk, Felix: *B-Human Team Report and Code Release 2013*. 2013. – Only available online: http://www.b-human.de/downloads/publications/2013/CodeRelease2013.pdf

[2] Leitte, JProf. Dr. H.: *Algorithmische Geometrie – 2. Konvexe Hülle*. http://www.iwr.uni-heidelberg.de/groups/CoVis/Teaching/AG_SS12/AG_2_ConvexHull.pdf, 2012. – Aufgerufen: 04.09.2017

[3] Qian, Yongbo ; Lee, Daniel D.: Adaptive Field Detection and Localization in Robot Soccer. In: *20th RoboCup International Symposium*. Leipzig, Germany, July 2016

[4] Reinhardt, Thomas: *Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball*. Leipzig, October 2011

[5] Röfer, Thomas ; Laue, Tim ; Bülter, Yannick ; Krause, Daniel ; Kuball, Jonas ; Mühlenbrock, Andre ; Poppinga, Bernd ; Prinzler, Markus ; Post, Lukas ; Roehrig, Enno ; Schröder, René ; Thielke, Felix: *B-Human Team Report and Code Release 2017*. 2017. – Only available online: http://www.b-human.de/downloads/publications/2017/coderelease2017.pdf

[6] Fefilatyev, S. ; Smarodzinava, V. ; Hall, L. O. ; Goldgof, D. B.: Horizon Detection Using Machine Learning Techniques. In: *2006 5th International Conference on Machine Learning and Applications (ICMLA'06)*, 2006, S. 17–21

[7] Dusha, Damien ; Boles, Wageeh W. ; Walker, Rodney: Fixed-Wing Attitude Estimation Using Computer Vision Based Horizon Detection. In: *12th Australian International Aerospace Congress*. Melbourne Australia, 2007, 1–19

[8] In: Ahmad, Touqeer ; Bebis, George ; Regentova, Emma E. ; Nefian, Ara: *A Machine Learning Approach to Horizon Line Detection Using Local Features*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–41914–0, 181–193

[9] Boroujeni, N. S. ; Etemad, S. A. ; Whitehead, A.: Robust Horizon Detection Using Segmentation for UAV Applications. In: *2012 Ninth Conference on Computer and Robot Vision*, 2012, S. 346–352

[10] RoboCup Federation: *RoboCup - Objective*. http://www.robocup.org/objective, 2017. – Aufgerufen: 30.08.2017

[11] RoboCup Federation: *Standard platform league history*. http://spl.robocup.org/history/, 2017. – Aufgerufen: 30.08.2017

[12] Softbank Robotics: *Nao H25*. http://doc.aldebaran.com/2-1/family/nao_h25/index_h25.html, 2017. – Aufgerufen: 30.08.2017

[13] Softbank Robotics: *Find out more about Nao*. https://www.ald.softbankrobotics.com/en/robots/nao/find-out-more-about-nao, 2017. – Aufgerufen: 30.08.2017

[14] Softbank Robotics: *Nao - Video Camera*. http://doc.aldebaran.com/2-1/family/robots/video_robot.html#robot-video, 2017. – Aufgerufen: 03.09.2017

[15] B-Human: *B-Human.de*. https://www.b-human.de/, 2017. – Aufgerufen: 30.08.2017

[16] Fischler, M.A. ; Bolles, R.C.: Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. In: *Communications of the ACM* 24 (1981), Nr. 6, S. 381–395

[17] Scheid, Harald ; Schwarz, Wolfgang: *Elemente der Linearen Algebra und der Analysis*. Springer, 2009

[18] Guennebaud, Gaël ; Jacob, Benoît u. a.: *Eigen documentation*. https://eigen.tuxfamily.org/dox/classEigen_1_1Hyperplane.html, 2010. – Aufgerufen: 13.09.2017

[19] Frese, Udo: *Echtzeitbildverarbeitung*. 2017. – Vorlesungsfolien an der Universität Bremen

[20] Laue, Tim ; Röfer, Thomas: SimRobot - Development and Applications. In: Amor, Heni B. (Hrsg.) ; Boedecker, Joschka (Hrsg.) ; Obst, Oliver (Hrsg.): *The Universe of RoboCup Simulators - Implementations, Challenges and Strategies for Collaboration. Workshop Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008)*. Venice, Italy, 2008