Universität Bremen
Fachbereich 3

# Robotergestütztes Kalibrierungssystem für Sensoren

Bachelorarbeit

Benny Prange
Matrikel-Nummer 2597237

**Betreuer** Alexis Maldonado
**Erstprüfer** Prof. Dr. Michael Beetz
**Zweitprüfer** Prof. Dr. Doe

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

In der Robotik werden bildgebende Sensoren benutzt, um Informationen über das Umfeld des Roboters zu erhalten. Diese Informationen können genutzt werden, um Hindernisse zu erkennen, die der Roboter meiden soll oder es werden Gegenstände gesucht, die der Roboter greifen und mit ihnen interagieren soll. Es gibt verschiedene Arten von bildgebenden Sensoren, wie z.B. Kameras, die ein zweidimensionales Bild erzeugen und Kameras, die zusätzlich zu dem zweidimensionalen Bild auch Tiefeninformationen erkennen. Andere Sensoren, die ein räumliches Abbild ohne Farbinformationen erstellen, sind z.B. Lidar-, Ultraschall- und Radarsensoren.

Bei all diesen Sensoren ist es wichtig, dass diese korrekt kalibriert sind, damit die gemessenen Daten der Realität möglichst nahe kommen. Stimmen die Messdaten nicht mit der Realität überein, stößt der Greifarm ein Objekt möglicherweise um, anstatt es zu greifen.

Kamerasensoren, die nach dem Lochkameraprinzip arbeiten, werden durch die Kameramatrix definiert. Sie enthält Angaben über den internen Aufbau der Kamera, wie Bildsensorgröße und Brennweite, sowie die Position der Kamera in Relation zu der realen Umgebung. Dadurch lassen sich aus einem aufgenommenen Bild Informationen über die dreidimensionale Welt berechnen.

Das Ziel der Kalibrierung einer Kamera ist es, eine Kameramatrix zu berechnen, die eine möglichst exakte Relation von Kamerabildern zu der realen Umgebung ermöglicht. Diese Kameramatrix kann mithilfe von Kalibrierungsmustern erstellt werden. Dazu werden mehrere Fotos aufgenommen, auf denen das Muster in verschiedenen Entfernungen und Orientierungen zu sehen ist. Schließlich lässt sich aus diesen Daten die Kameramatrix berechnen.

Dieser Vorgang ist sehr zeitaufwändig. Zum einen müssen viele Bilder gemacht werden, um eine möglichst präzise Kalibrierung zu erhalten. Daher werden bis zu 50 Bilder aufgenommen. Zum anderen empfiehlt es sich, Kamera und Kalibiriermuster jeweils auf einem Stativ zu befestigen, um scharfe Bilder zu erzeugen. Damit ist gewährleistet, dass die Erkennung des Kalibriermusters nicht von unscharfen Bildern gestört wird, allerdings wird der Vorgang durch das Verstellen der Stative auch weiter verlängert.

Um die Kalibrierung zu vereinfachen, soll daher ein Roboterarm das Bewegen des Kalibriermusters übernehmen und der ganze Ablauf mit dem Erstellen der Bilder und dem Berechnen der Parameter durch ein Programm gesteuert werden. Der Benutzer platziert die Kamera auf einem Stativ vor dem Roboterarm, an dem das Kalibiriermuster befestigt ist. Anschließend startet er das Programm, welches den Arm zu verschiedene Positionen bewegt, sodass die Kamera das Muster aus mehreren Entfernungen und Orientierungen aufnehmen kann. Zum Schluss berechnet das Programm die benötigten Parameter und der Benutzer kann die Kamera mit diesen Parametern einsetzen.

TODO: Kurze Beschreibung der folgenden Kapitel

# 2 Grundlagen

Kameras sind wichtige Sensoren in der Robotik, indem sie dem Roboter helfen, die Umgebung wahrzunehmen. Abhängig von dem Einsatzgebiet, ist ein genaues Abbild der Umgebung außerordentlich wichtig für die Erfüllung einer Aufgabe. Dies ist der Fall, wenn der Roboter über das Kamerabild einen Gegenstand lokalisieren muss, um ihn anschließend zu greifen, oder er muss in dem Kamerabild Hindernisse erkennen, um einen Weg zum Ziel zu planen.

Der relativ günstige Preis von Lochkameras gegenüber anderen bildgebenden Sensoren hat dazu beigetragen, dass Kameras ein viel genutzter Sensor sind. Da sich aber gezeigt hat, dass die Toleranzen bei der Produktion und der Aufbau der Kameras an sich zu Verzerrungen im Bild führen, wurden Modelle entwickelt, um diese Verzerrungen herauszurechnen und das Bild zu korrigieren. Ein häufig genutztes Modell wird im folgenden beschrieben.

## 2.1 Kameramodell

In [Zha00] wird ein gängiges Kameramodell beschrieben, welches hier erklärt wird.

Die Beziehung zwischen einem Punkt $(u, v)$ in einem zweidimensionalen Bild und einem Punkt $(x, y, z)$ in der dreidimensionalen Welt wird durch Gleichung 2.1 beschrieben:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.1}$$

mit

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 0 \end{bmatrix} \tag{2.2}$$

$K$ enthält die intrinsischen Parameter der Kamera, die auf Grund der Fertigungstoleranzen für jede Kamera unterschiedlich sind. $f$ definiert die Brennweite der Kamera. Da die Pixel nicht immer quadratisch sind, wird dieser Wert für die x und y-Achse angegeben. $c_x$ und $c_y$ beschreiben das optische Zentrum des Bildes, welches ebenfalls nicht immer genau im Mittelpunktes des Bildes liegt.

$R$ und $T$ definieren die extrinsischen Parameter. $R$ ist eine Rotationsmatrix, die die Rotation zwischen dem Kamera- und Weltkoordinatensystem beschreibt, $T$ ist ein Vektor, der die Transformation zwischen Kamera- und Weltkoordinatensystem beschreibt.

## 2.2 Verzeichnung

Mit dem Kameramodell kann nun eine Beziehung zwischen einem Punkt im Bild und dem Punkt in der Welt hergestellt werden. Kameras bilden das Bild jedoch nicht immer korrekt ab. Häufig erkennt man in Bildern den Effekt, dass gerade Linien in der Realität, wie z.B. Häuserkanten, nicht gerade im Bild abgebildet werden, sondern das diese gebogen sind. Diesen Effekt nennt man Verzeichnung.

Man kann zwischen zwei verschiedenen Arten der Verzeichnung unterscheiden. Die Verzeichnung, die gerade Linien gekrümmt darstellt, nennt sich radiale Verzeichnung. Um diese Verzeichnung herauszurechnen, werden die drei Faktoren $k_1, k_2, k_3$ benötigt. Sind diese gegeben, lässt sich die Verzeichnung korrigieren. $r$ ist der Abstand vom Bildmittelpunkt zum Punkt, der korrigiert werden soll. $x', y'$ sind die falsch abgebildeten Punkte, $x, y$ sind die korrigierten Punkte.

$$x = x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.3}$$

$$y = y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.4}$$

Zusätzlich zu der radialen Verzeichnung gibt es die tangentiale Verzeichnung. Diese tritt auf, wenn die Kameralinse nicht parallel zum Sensor eingebaut ist. Diese Verzeichnung führt dazu, dass Linien, die im idealen Bild parallel verlaufen, nun einen gemeinsamen Fluchtpunkt besitzen. Um diese Verzeichnung zu korrigieren, werden die Faktoren $p_1, p_2$ benötigt.

$$x = x' + (2p_1xy + p_2(r^2 + 2x^2)) \tag{2.5}$$

$$y = y' + (p_1(r^2 + 2y^2) + 2p_2xy) \tag{2.6}$$

## 2.3 Kalibrierung

Der Kalibrierungsvorgang dient dazu, die intrinsischen und gegebenenfalls extrinsischen Parameter sowie die Verzeichnungsparameter zu berechnen. Dies erfolgt in mehreren Schritten.

### 2.3.1 Kalibrierungsmuster

Man benötigt ein Kalibrierungsmuster, von dem einige Aufnahmen gemacht werden. Es gibt verschiedene Kalibrierungsmuster, die benutzt werden können. TODO: Quellen für Muster Eine grundlegende gemeinsame Eigenschaft aller Kalibrierungsmuster ist, dass das Muster aus geometrischen Formen besteht, die gut von Bilderkennungsalgorithmen gefunden werden können. Häufig wird dazu ein Schachbrettmuster benutzt, siehe Abbildung 2.1. In diesem Muster haben die Rechtecke eine Kantenlänge von exakt 3cm. Inge samt sind 6 x 8 Rechtecke in dem Muster vorhanden. Der Kalibrierungsalgorithmus sucht nach den Kanten der Quadrate. Der Schnittpunkt von zwei Kanten, also die Ecke eines Quadrats, bildet den Kalibrierungspunkt. Dadurch enthält ein Bild mit diesem Kalibrierungsmuster 5 x 7, also 35, Kalibrierungspunkte.

Ein anderes Kalibrierungsmuster besteht aus Kreisen, die regelmäßig versetzt zueinander positioniert sind, siehe Abbildung 2.2. In diesem Muster sind die Mittelpunkte der

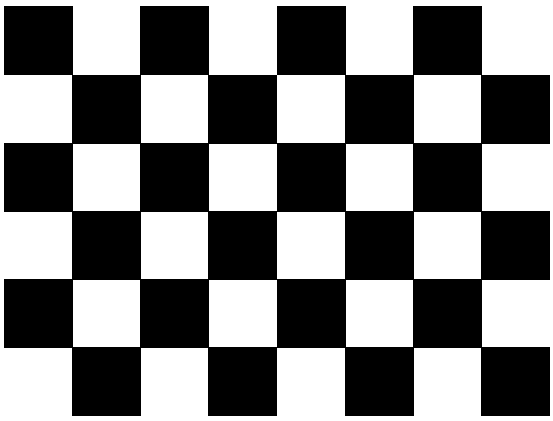

**Abbildung 2.1:** Schachbrettmuster als Kalibrierungsmuster

Kreise die Kalibrierungspunkte. In diesem Muster sind 10 x 11 Kreise eingezeichnet, sodass deutlich mehr Informationen für die Kalibrierung in einer Aufnahme vorhanden sind.

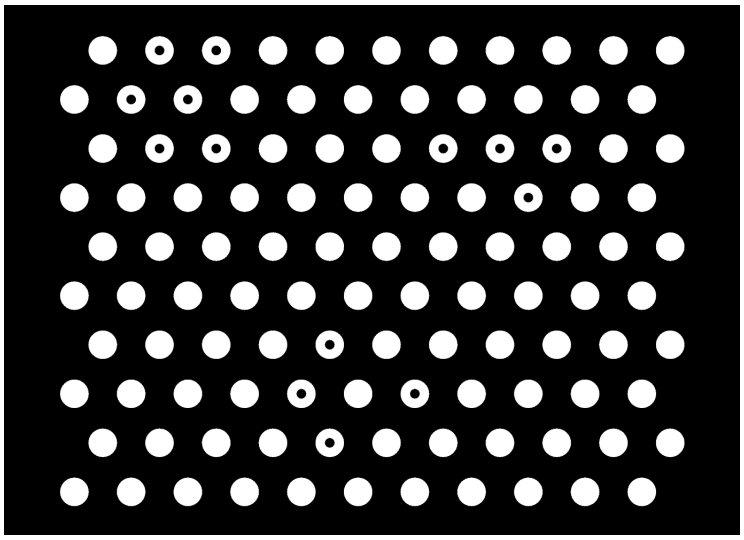

**Abbildung 2.2:** Punkte als Kalibrierungsmuster

Das Muster ist außerdem speziell für HALCON entwickelt. Während die Kreismuster von OpenCV im ganzen Bild den gleichen Punkt benutzen, sind in diesem Muster einzelne Punkte zusätzlich markiert. Dadurch kann HALCON auch dann noch Kalibrierungsinformationen aus einer Aufnahme errechnen, wenn das Kalibrierungsmuster nur teilweise zu sehen ist. Bei dem Schachbrettmuster oder dem Punktmuster von OpenCV muss hingegen immer das ganze Muster erkennbar sein.

Unabhängig davon, welches Muster man einsetzt, muss immer die Größe und Form der ausgedruckten Muster exakt den erwarteten Daten entsprechen. Da manche Drucker versuchen, die Dokumente auf DinA4-Größe anzupassen, muss man für jedes ausgedruckte Muster die Größe und Abstände der geometrischen Objekte überprüfen. Hat das ausgedruckte Muster die korrekte Größe, muss dieses auf eine möglichst ebene Oberfläche geklebt werden. Ein gekrümmtes Muster in einer Aufnahme, die zur Kalibrierung benutzt wird, kann die Ergebnisse verfälschen.

### 2.3.2 Bildaufnahme

Damit der Kalibrierungsalgorithmus die Parameter berechnen kann, muss er aus mehreren Bildern die Kalibrierungspunkte extrahieren. Dabei werden an die Bilder, die der Kalibrierung dienen, mehrere Anforderungen gestellt.

Wie bei anderen Fotos auch, ist es wichtig, dass das Motiv, also das Kalibrierungsmuster, scharf dargestellt wird und gut belichtet ist. Dabei ist es hilfreich, wenn Kamera und Kalibrierungsmuster auf Stativen befestigt sind.

Das Kalibrierungsmuster sollte außerdem in verschiedenen Distanzen aufgenommen werden. Andernfalls kann es passieren, dass die berechneten Parameter nur zu dem Abstand passen, den das Kalibrierungsmuster zur Kamera während der Aufnahmen hatte. Möchte man nun mit diesen Parametern die Kamera benutzen und Objekte mit einem anderen Abstand beobachten, können die Parameter das Bild nun verschlechtern und die Bildinformationen passen nicht mehr zur Realität.

Nicht nur die Entfernung zur Kamera ist wichtig sondern auch die Orientierung des Kalibrierungsmusters. Indem man das Kalibrierungsmuster an jeder Position nach oben und unten sowie nach rechts und links neigt, verbessert man die Qualität der Kalibrierung.

Letztendlich sollte das Kalibrierungsmuster über alle einzelnen Bilder verteilt den ganzen Bildausschnitt füllen. Würden beispielsweise alle Bilder das Kalibrierungsmuster nur in der Bildmitte zeigen, könnten die Parameter so gewählt werden, dass die Bildmitte zwar sehr gut kalibriert ist, aber das Bild an den Bildrändern stark verzeichnet ist.

### 2.3.3 Berechnung der Parameter

Wurden genügend Bilder aufgenommen, die den Anforderungen entsprechen, können die Parameter berechnet werden. Dazu ist eine initiale Konfiguration der intrinsischen Parameter nötig. Die dafür notwendigen Informationen kann man den Spezifikationen der Kamera oder des Sensors entnehmen. Die Brennweite $f$ wird in der Regel vom Hersteller angegeben. Für das optische Zentrum $c_x$ und $c_y$ kann man die Bildmitte, also die halbe Auflösung des Bildes, annehmen.

Anschließend werden in jedem aufgenommenen Bild die Kalibrierungspunkte aus dem Bild extrahiert und abgespeichert. Im nächsten Schritt werden dann mit diesen Informationen die Gleichungssysteme gelöst, die die gesuchten Parameter enthalten. Zusätzlich zu den Parametern berechnen die Algorithmen den durchschnittlichen Fehler in Pixeln. Dieser sollte möglichst klein, am besten unter eins sein. Wird das Kalibrierungsmuster nicht in verschiedenen Orientierungen aufgenommen oder es ist nicht im ganzen Bildausschnitt zu sehen, kann dies den Fehler vergrößern.

Es gibt mehrere Frameworks, die man zum Kalibrieren benutzen kann. OpenCV ist eine freie Bibliothek zur Bildverarbeitung und zum maschinellen Sehen. Es wird kein fertiges Programm angeboten, mit dem der Vorgang direkt gestartet werden kann. Stattdessen existieren Bibliotheken in C++ und Python, in denen der Benutzer Funktionen findet, mit denen er sein eigenes Programm schreiben kann, um eine Kamera zu kalibrieren. Für Benutzer, die mit dieser Materie nicht so vertraut sind, ist dies eine hohe Hürde. Profis können damit aber das Programm auf die eigenen Bedürfnisse anpassen.

Ein anderes Framework ist HALCON. Es wird kommerziell vermarktet und ist nicht frei verfügbar. Wie bei OpenCV werden Bibliotheken für verschiedene Programmiersprachen angeboten, mit denen Benutzer ihre eigenen Programme schreiben können. Zusätzlich existiert mit HDevelop eine integrierte Entwicklungsumgebung. In HDevelop wird eine eigene Programmiersprache benutzt, die weniger tief und detailliert als andere Sprachen ist, aber dafür für unerfahrene Benutzer deutlich sprechender und intuitiver zu benutzen ist. Außerdem ist ein Kalibrierungsassistent enthalten, der die aufgenommen Bilder vor der Berechnung der Parameter anhand ihrer Bildschärfe, Belichtung und weiterer Eigenschaften bewertet.

### 2.3.4 Weitere benutzte Frameworks

## ROS

ROS [QCG+09] steht für Robot Operating System. Es wurde entwickelt, um ein Framework für die Programmierung von unterschiedlichen Robotern zur Verfügung zu stellen. Möglich ist die Einbindung fertige Pakete, die von anderen Benutzern bereitgestellt werden, zum Beispiel MoveIt!, sieheAbschnitt 2.3.4. Außerdem kann man kann außerdem eigene Programme schreiben und in ROS integrieren.

In der Regel startet man mehrere Programme, die jeweils eine bestimmte Aufgabe erfüllen sollen. Es gibt unterschiedliche Möglichkeiten, mit denen die Programme untereinander kommunizieren können. ROS stellt Bibliotheken für viele verschiedene Programmiersprachen bereit, damit man eigene Pakete schreiben kann, die sich in das Gesamtsystem integrieren lassen.

Im weiteren Verlauf werden einige ROS-spezifische Begrifflichkeiten benutzt, die nun kurz erklärt werden.

Da ein in ROS eingesetztes Programm einem bestimmten Zweck dient, werden meistens mehrere Programme parallel gestartet, die miteinander kommunizieren. Ein einzelnes Programm wird als `Node` bezeichnet. Die einzelnen Nodes bilden dazu ein eigenes kleines Netzwerk. Manche Nodes sprechen einen Sensor an, um seine Daten zu veröffentlichen und anderen Programmen zugänglich zu machen. Dazu veröffentlichen sie die Sensordaten auf einem `Topic`. Die Node, die die Daten veröffentlicht, nennt man dann `Publisher`. Die Nodes, die diese Daten empfangen möchten, um sie auszuwerten, nennt man `Subscriber`.

Zusätzlich dazu gibt es die Möglichkeit, dass sich eine Node als `Actionserver` im Netzwerk registriert und andere Nodes als `Actionclients` Aufgaben an diese Node schicken können. Die Clients bekommen Informationen dazu, was für eine Aufgabe ein Actionserver anbietet und welche Parameter der Client angeben kann. Erhält der Server eine Aufgabe, versucht er sie durchzuführen und gibt dem Client zurück, ob die Aufgabe erfolgreich ausgeführt wurde oder nicht. Anschließend wartet der Actionserver auf weitere Aufgaben.

Benutzer können interne Parameter einer Node zum Startzeitpunkt ändern, wenn der Programmierer dies vorgesehen hat. Dazu definiert der Benutzer beim Aufruf der Node `Argumente`, die an das Programm übergeben werden. Dadurch lässt sich zum Beispiel der Name eines Topics ändern, wenn man zum Beispiel eine andere Kamera benutzt.

### MoveIt!

MoveIt! [CSC12] ist ein Framework, dass die Steuerung eines Roboters abstrahiert. MoveIt! stellt Bibliotheken für verschiedene Programmiersprachen bereit, mit denen man die Bewegung des Roboters kontrollieren kann. Möchte man wie in diesem Projekt einen Roboterarm steuern, kann man eine Zielposition und Orientierung für die Spitze des Arms angeben. MoveIt! berechnet für diese Kombination aus Position und Orientierung dann die Konfiguration der Robotergelenke. Im nächsten Schritt wird dann eine Abfolge von Zwischenkonfigurationen gesucht, damit sich der Roboter von seiner Startkonfiguration zur Zielkonfiguration bewegt, ohne dass er sich selbst oder andere Objekte in seiner Umgebung berührt. Existiert eine solche Abfolge von Konfigurationen, wird diese ausgeführt, sodass sich der Roboterarm letztendlich an der gewünschten Position mit der gewünschten Orientierung befindet.

TODO: HALCON TODO: OpenCV

# 3 Architektur

Für dieses Projekt wurden mehrere Softwareprogramme integriert, teilweise wurden eigene Programme geschrieben. In den folgenden Abschnitten wird ein Gesamtüberblick gegeben und die einzelnen Programme werden hinsichtlich ihrer Aufgaben detailliert beschrieben.

## 3.1 Gesamtübersicht

In Abbildung 3.1 ist die Architektur graphisch dargestellt. Jedes Element stellt dabei eine ROS-Node dar, also ein einzelnes Programm. Elemente, die blau eingefärbt sind, stehen für Programme, die von anderen Benutzern programmiert wurden und gegebenenfalls zu diesem Einsatzzweck angepasst wurden. Programme, die im Rahmen dieser Bachelorarbeit programmiert wurden, sind grün eingefärbt.

TODO: Andere Namen für die Nodes

TODO: Alternative für Element Die beiden Elemente ur_moder_driver und libuvc_camera steuern direkt die Hardware an. Im Fall von ur_modern_driver wird der UR5-Arm gesteuert, libuvc_camera kommuniziert mit der Webcam.

Das Element universal_robot stellt eine abstrahierte Schnittstelle zur Steuerung des Roboterarms zur Verfügung.

Die beiden Elemente MoveArmServer und caltab_detector_node wurden im Rahmen dieser Bachelorarbeit entwickelt. MoveArmServer bietet Funktionen zur Steuerung des Arms, die speziell in diesem Projekt benötigt werden. caltab_detector_node verarbeitet die Bildinformationen.

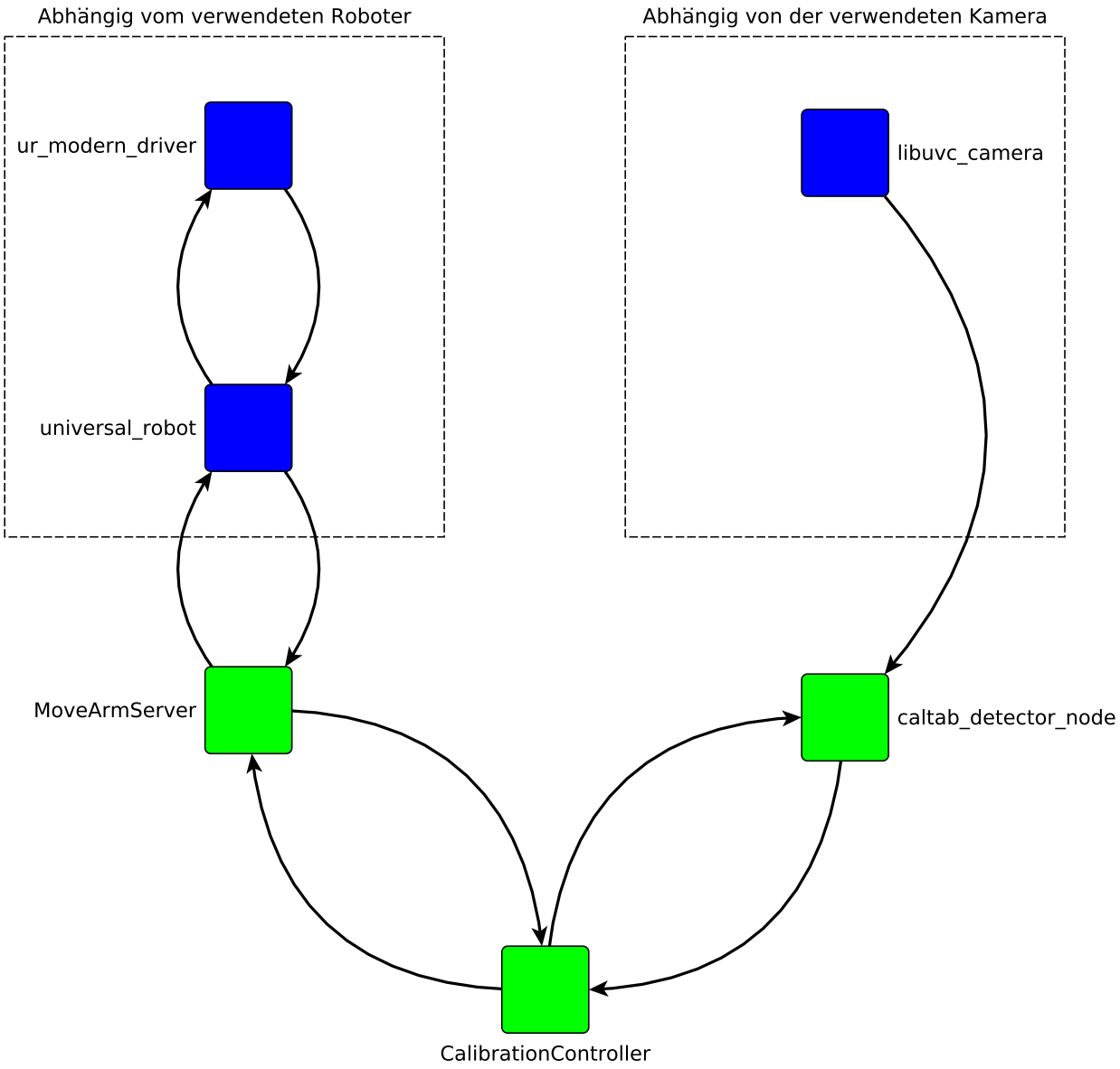


**Abbildung 3.1:** Übersicht über die Architektur

Der CalibrationController wurde ebenfalls im Rahmen dieser Bachelorarbeit entwickelt. Dieses Programm steuert den gesamten Ablauf, kommuniziert dazu mit den beiden selbstgeschrieben Nodes und sorgt dafür, dass der Roboter sich an die gewünschten Positionen bewegt und die Bilder im richtigen Moment aufgenommen werden. Zum Schluss werden dem Benutzer dann die ermittelten Parameter zurückgegeben.

Durch diese Architektur und den Einsatz von ROS hat man den Vorteil, dass man weder an den UR5 Roboterarm noch an die Webcam mit libuvc gebunden ist, um eine Kamera mit Hilfe eines Roboters zu kalibrieren. Die caltab_detector_node benötigt lediglich ein Topic, auf dem die Bilder der eingesetzten Kamera veröffentlicht werden. Dies kann man wie in diesem Fall mit libuvc_camera machen, aber abhängig von der eingesetzten Kamera kann man auch andere Programme benutzen, die diese Funktionalität zur Verfügung stellen. Man kann ebenso den Roboterarm austauschen. Wichtig ist nur, dass es für den Roboter eine MoveIt!-Integration gibt und sich an dessen Ende ein Kalibrierungsmuster befestigen lässt.

## 3.2 ur_modern_driver

Diese Node wurde bereits als Paket in ROS veröffentlicht, siehe [And15]. Sie ist die Schnittstelle zwischen dem Roboterarm UR5 und dem ROS Framework. Da dieses Paket nicht für die in diesem Projekt benutzte Version von ROS entwickelt wurde, wurden einige wenige Anpassungen am Sourcecode vorgenommen, um das es in diesem Projekt benutzen zu können.

## 3.3 universal_robot

Diese Node wurde ebenfalls als Paket in ROS veröffentlicht, siehe [EGH+17]. In dieser Node ist MoveIt! integriert, welches zum Bewegen des Roboterarms benutzt wird. Man kann diesem Programm als Ziel die gewünschte Position und Orientierung des Roboterarms übergeben; das Programm berechnet dann die Abfolge der Roboterkonfigurationen, die zum Erreichen dieses Ziels benötigt werden und schickt die Abfolge dann an den ur_modern_driver.

Diese Node wird im Projekt von dem Programm MoveArmServer angesprochen. Dazu stellt sie über MoveIt! einige Schnittstellen bereit, die in diesem Fall benutzt werden, um die Position und Orientierung des Kalibrierungsmusters zu verändern.

## 3.4 libuvc_camera

Auch diese Node ist als Paket in ROS verfügbar. Sie dient dazu, das Bild der angeschlossenen Webcam über ROS verfügbar zu machen. In dem Script, mit dem diese Node gestartet wird, lassen sich verschiedene Parameter einstellen. In diesem Fall wurde mittels der Vendor-ID und Product-ID die zu benutzende Kamera definiert, die Auflösung wurde auf Full-HD gesetzt und der Fokus wurde so eingestellt, dass der Bereich, in dem sich das Kalibrierungsmuster befinden wird, scharf zu sehen ist.

Die Node veröffentlicht die Bilder der Kamera über Topics. Das Programm caltab_detector_node empfängt die Bilder und wertet sie aus.

## 3.5 MoveArmServer

Dieses Programm wurde im Rahmen dieser Bachelorarbeit selbst entwickelt. Es dient dazu, dem CalibrationController eine weitere Abstraktionsschicht zum Bewegen des Arms zu bieten.

Dazu wurde ein Actionserver erstellt, der die Position des Kalibrierungsmusters als Auftrag erhält. Zusätzlich kann als weiterer Parameter für den Auftrag die Neigung des Musters angegeben werden. Das Programm berechnet dann für die gewünschte Position zunächst die Orientierung für den Roboterendpunkt, damit das Muster senkrecht zur Kamera steht. Anschließend wird zu der berechneten Orientierung die gewünschte Neigung hinzugefügt. Die so errechnete Position und Orientierung wird dann an MoveIt! übergeben. Abhängig davon, ob MoveIt! den Arm erfolgreich bewegen konnte, informiert der MoveArmServer den CalibrationController, ob der Auftrag erfolgreich ausgeführt wurde.

## 3.6 caltab_detector_node

Auch dieses Programm wurde selbst entwickelt. Es benutzt die Bibliothek von HALCON, um in den Bildern, die es von der Kamera empfängt, nach dem Kalibrierungsmuster zu suchen und die Kalibrierung durchzuführen.

Diese Node bietet zwei Actionserver an. Der erste Actionserver erwartet einen Auftrag, um in den aktuellen Bildern nach dem Kalibrierungsmuster zu suchen. Dazu wird ein Parameter übergeben, der angibt, wie viele Bilder ausgewertet werden sollen. Die Algorithmen von HALCON finden nicht immer im ersten Versuch das Muster, daher lässt sich die Anzahl der Bilder mit Hilfe dieses Parameters definieren. Die Node meldet dem CalibrationController zurück, ob nach der angegeben Anzahl der Bilder das Muster gefunden wurde oder nicht.

Der zweite Actionserver nimmt den Auftrag zur Durchführung der Kalibrierung an. Dazu werden die Bilder, in denen das Muster zu sehen war, ausgewertet. Danach werden die errechneten intrinsischen Parameter, die Verzeichnungsparameter sowie der ermittelte durchschnittliche Fehler an den CalibrationController zurückgegeben.

## 3.7 CalibrationController

Diese letzte Node wurde ebenfalls selbst entwickelt. Sie dient dazu, den gesamten Ablauf der Kalibrierung zu steuern, indem sie die Actionserver mit Aufgaben versorgt. Zunächst übergibt der Benutzer dem Programm die Position der Kamera in Relation zur Basis des Roboterarms und die initialen intrinsischen Parameter. Dann wird aus diesen Werten das Sichtfeld der Kamera bestimmt. Mit diesem Sichtfeld werden danach die verschiedenen Distanzen zur Kamera berechnet und anschließend die Positionen für das Kalibrierungsmuster, damit dieses das ganze Sichtfeld der Kamera abdeckt.

Diese Positionen werden nacheinander an den MoveArmServer übergeben. Für jede Position wird das Muster für um $10^\circ$, $20^\circ$, $30^\circ$ und $40^\circ$ nach oben, unten, rechts und links geneigt. Hat der Actionserver den Auftrag erfolgreich ausgeführt, wird der caltab_detector_node der Auftrag übergeben, in dem Kamerabild nach dem Muster zu suchen. Ist dieser Auftrag abgeschlossen, wird die nächste Position angesteuert.

Wurden alle Positionen abgearbeitet, wird zum Schluss der Auftrag zur Kalibrierung durchgeführt. Die dadurch erhaltenen Parameter werden dem Benutzer angezeigt und das Programm hat seine Durchführung beendet.

# 4 Implementierung

Nachdem der grundsätzliche Aufbau und Ablauf der Kommunikation des Gesamtprogramms beschrieben wurde, werden nun die Details der einzelnen Module aufgeführt.

## 4.1 universal_robot

An diesem Programm wurden einige Anpassungen vorgenommen. Zum einen wurde die in MoveIt! benutzte Bibliothek zur Lösung der inversen Kinematik KDL durch TRAC_IK ausgetauscht. Diese Bibliothek wird dazu benutzt, für eine gewünschte Position und Orientierung des Roboterarms die dazu erforderlichen Winkel der Gelenke zu berechnen. Während der Versuche kam es vor, dass MoveIt! keinen oder nur einen viel zu komplizierten Weg von der Start- zur Zielkonfiguration gefunden hat. Dieses Problem konnte teilweise durch diesen Austausch behoben werden.

Zusätzlich wurde das URDF in diesem Paket angepasst. Das URDF enthält die geometrischen Informationen über den Roboterarm, wie z.B. die Länge und Form der einzelnen Glieder und die Position der Werkzeugaufnahme des Arms. Wird dem Arm eine Zielposition und Orientierung übergeben, so gilt diese normalerweise für diese Werkzeugaufnahme. Für die Bachelorarbeit wurde jedoch das Kalibrierungsmuster an diese Stelle montiert. Das Muster wurde nicht mittig an der Werkzeugaufnahme angebracht sondern steht seitlich ab. Daher wurde im URDF der Endpunkt des Arms von der Werkzeugaufnahme auf den Mittelpunkt des Kalibrierungsmusters versetzt. Das führt dazu, dass sich nun nicht mehr die Werkzeugaufnahme des Arms an der Zielposition befindet, sondern der Mittelpunkt des Kalibrierungsmusters.

## 4.2 MoveArmServer

Diese Node wurde in der Programmiersprache Python entwickelt. Sie hat den Auftrag, die Bewegung des Arms zu koordinieren und dazu mit MoveIt! zu kommunizieren.

Das Programm erhält beim Start die Position der Kamera in Relation zur Basis des Arms. Außerdem ein Actionserver gestartet, der auf Aufgaben vom Typ `MoveArm` wartet. Zusätzlich wird der Kommunikationskanal zu MoveIt! gestartet.

Die Aufgabe `MoveArm` ermöglicht es einem Actionclient, den Roboterarm zu einer von ihm gewählten Position zu bewegen. Zusätzlich kann er die horizontale und vertikale Neigung des Musters bestimmen.

Erhält das Programm eine Aufgabe, so wird zunächst für die gewünschte Position die Orientierung des Arms berechnet. Dazu wird zunächst angenommen, dass das Kalibrierungsmuster orthogonal zur Kamera stehen soll. Jede Position lässt sich mit den drei Variablen $x, y, z$ definieren. Dadurch lässt sich jede Position auch als Ortsvektor darstellen:

$$\vec{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{4.1}$$

Um nun die Orientierung zu errechnen, subtrahiert man zunächst den Vektor für Position der Kamera vom Vektor für die Position des Roboterarms. Der daraus resultierende Vektor $\vec{p}_{diff}$ beschreibt die geometrische Transformation vom Arm zur Kamera.

$$\vec{p}_{Diff} = \vec{p}_{Roboter} - \vec{p}_{Kamera} \tag{4.2}$$

Mit diesem Vektor lässt sich nun der Gierwinkel $\alpha$ und Neigungswinkel $\beta$ bestimmen, damit das Muster orthogonal zur Kamera steht:

$$\alpha = \arctan2(y_{diff}, x_{diff}) \tag{4.3}$$

$$\beta = -\arcsin(x_{diff}) \tag{4.4}$$

Mit diesen Winkeln würde nun das Muster orthogonal zur Kamera zeigen. Wurden in der Aufgabe zusätzliche Neigungen angegeben, werden diese nun zu diesen Winkeln addiert.

Schließlich muss noch der Rollwinkel $\gamma$ bestimmt werden. In diesem Anwendungsfall soll das Kalibrierungsmuster immer nach außen zeigen und parallel zum Boden stehen, damit die Reichweite des Roboters erhöht wird.

TODO: Algorithmus zur Berechnung des Rollwinkels beschreiben

Wurde die Orientierung berechnet, muss für die Position und Orientierung ein Plan generiert werden, mit dem sich der Arm aus der Startposition zur Zielposition bewegt, ohne dass er mit sich selbst oder der Umgebung kollidiert. Diese Aufgabe übernimmt MoveIt!. Aufgrund des eingesetzten Algorithmus in MoveIt! können für gleiche Start- und Zielpositionen unterschiedliche Pläne erstellt werden, teilweise kann MoveIt! auch keinen Plan finden. Daher werden für jede Aufgabe, die der Actionserver erhält, fünf Pläne von MoveIt! erzeugt. Wenn die gewünschte Position nicht erreichbar ist oder Teile des Roboters oder das Kalibrierungsmuster mit sich selber oder der Umgebung kollidieren würden, sind alle fünf Pläne leer. In diesem Fall wird die Aufgabe abgebrochen und dem Actionclient eine negative Rückmeldung zurückgegeben. Ansonsten wird der Plan ausgewählt, der die geringste Anzahl an Zwischenpositionen enthält, und an MoveIt! zur Ausführung übermittelt. Hat MoveIt! die erfolgreiche Ausführung bestätigt, wird dem Actionclient die Aufgabe positiv bestätigt.

## 4.3 caltab_detector_node

Diese Node wurde in der Programmiersprache C++ programmiert. Sie dient dazu, die von der Kamera übermittelten Bilder auszuwerten und zur Kalibrierung vorzuhalten.

Über das Topic `/camera/image_raw` erhält das Programm die Bilder der Webcam. Zusätzlich stellt das Programm zwei Actionserver zur Verfügung. Der Actionserver mit dem Namen `FindCaltab` hat die Aufgabe, in dem aktuellen Bild der Webcam nach dem Kalibrierungsmuster zu suchen. Der Actionclient kann für jeden Auftrag angeben, in wie vielen aufeinanderfolgenden Bildern der Server nach dem Muster suchen soll, da dieses nicht immer im ersten Versuch gefunden wird. Hat der Server das Muster innerhalb der

erlaubten Anzahl von Bildern gefunden, erhält der Client eine positive Rückmeldung, ansonsten eine negative.

Der zweite Actionserver `Calibrate` führt die Kalibrierung durch und errechnet aus allen Bildern, in denen das Muster zu sehen war, die intrinsischen Parameter, die Verzeichnungsparameter und den mittleren Fehler. Zusätzlich werden diese Bilder auf der Festplatte abgespeichert. Für jedes Bild wird außerdem eine Kopie angefertigt, in welches eine Simulation des Musters über das im Bild zu sehende Muster gelegt wird. Dadurch kann zusätzlich visuell die Qualität der Kalibrierung überprüft werden.

Beim Start der Node kann der Benutzer ein Verzeichnis angeben, in dem die Bilder gespeichert werden sollen. Andernfalls wird in dem aktuellen Verzeichnis ein Unterordner erstellt. Außerdem wird die Datei gelesen, in der die initialen intrinsischen Parameter für die eingesetzte Kamera hinterlegt sind. Ebenfalls wird die Datei gelesen, in der die Dimensionen und Eigenschaften des benutzten Kalibrierungsmusters definiert sind.
TODO: Das muss ebenfalls vom Benutzer konfiguriert werden können. Außerdem muss die Erstellu
Anschließend verbindet sich das Programm mit dem Topic der Kamera, um die Bilder zu empfangen.

Jedes Mal, wenn das Programm ein aktuelles Bild von der Kamera erhält, wird die Methode `subscriberCallback` aufgerufen. Die Methode überprüft zunächst, ob auf Grund eines laufenden Auftrags für den FindCaltab Actionserver noch Bilder überprüft werden sollen. Ist dies nicht der Fall, wird die Methode direkt beendet und das Programm wartet weiter auf eingehende Aufträge oder neue Bilder.

Wenn jedoch ein Auftrag aktiv ist und das aktuelle Bild ausgewertet werden soll, wird die Methode weiter abgearbeitet. Das Bild, das über das Topic empfangen wurde, ist vom Typ `sensor_msgs::Image`. Um es aber mit HALCON benutzen zu können, muss es zu dem Typ `HObject` umgewandelt werden. Dazu wird das Bild in einem Zwischenschritt mit Hilfe der Methode `toCvCopy` aus dem Paket `cv_bridge` in ein Objekt vom Typ `cv::Mat` konvertiert. Dadurch kann man nun relativ einfach auf den Wert für jedes einzelne Pixel zugreifen. Anschließend wird Pixel für Pixel über das gesamte Bild iteriert und die Werte der Pixel werden in einem `unsigned char*` gespeichert. Die HALCON-Methode `GenImage1` liest dann die Werte aus dem `unsigned char*` und erstellt das `HObject`.

Für jedes auszuwertende Bild wird ein fortlaufender Index erstellt, der im weiteren Verlauf benötigt wird. Es wird nun die HALCON-Methode `FindCalibObject` aufgerufen.

Diese sucht in dem eben erstellen Bild nach dem Kalibrierungsmuster. Wird das Muster nicht erkannt, wirft diese Methode eine Exception. Diese wird im weiteren Verlauf der `subscriberCallback`-Methode gefangen und in der Konsole wird die Fehlermeldung von HALCON ausgegeben. Kann HALCON das Muster erkennen, wird es intern unter dem Index abgespeichert und zusätzlich in dem vom Benutzer angegeben Ordner gespeichert. In einer von diesem Programm verwalteten Liste wird zusätzlich der Index dieses Bildes abgespeichert. Der boolesche Wert caltabFound wird auf wahr gesetzt und die Anzahl der noch auszuwertenden Bilder für den aktuellen Auftrag auf null.

Erhält der FindCaltab Actionserver einen neuen Auftrag, wird die Methode `findCaltabAction` aufgerufen. Zunächst wird der booleschen Wert auf falsch und die Anzahl der auszuwertenden Bilder auf den Wert gesetzt, der im Auftrag vom Client angegeben wurde. Dann wartet die Methode so lange, bis entweder die angegebene Anzahl an Bildern ausgewertet wurde oder vorher ein Kalibrierungsmuster gefunden wurde. Ist der boolesche Wert caltabFound wahr, wird dem Client eine positive Rückmeldung gegeben, sonst eine negative.

Wenn der Calibrate Actionserver einen Auftrag erhält, wird die Kalibrierung durchgeführt. Dazu wird zunächst die HALCON-Methode `CalibrateCameras` aufgerufen. Diese berechnet intern die neuen intrinsischen Parameter und die Parameter für die Verzeichnung. Außerdem wird der mittlere Fehler zurückgegeben. Um nun die neuen Parameter zu erhalten, wird die HALCON-Methode `GetCalibData` aufgerufen. Die so erhaltenen Werte werden zunächst in Werte vom Typ `double` umgewandelt. Dann werden dem Client diese Werte, der Fehler und die Anzahl der zur Kalibrierung benutzten Bilder zurückgegeben.

Anschließend wird nacheinander jedes abgespeicherte Bild mit Hilfe der Liste, die alle Indizes enthält, noch einmal mit HALCON eingelesen. Zunächst wird für das Bild die Position und Orientierung des Kalibrierungsmuster von HALCON ausgelesen, wie es von HALCON von erkannt wurde. Dann werden diese Informationen mit denen neuen Kameraparametern korrigiert. Schließlich wird das so korrigierte Kalibrierungsmuster über das eingelesene Bild gelegt und das Bild so auf dem Dateisystem abgespeichert.
TODO: Beispiel Screenshot einfügen und erklären.

## 4.4 MovementController

Die Klasse MovementController wurde wieder in Python entwickelt. Sie agiert als Actionclient für den MoveArm Actionserver und den FindCaltab Actionserver und koordiniert die Bewegung des Arms zu den verschiedenen Positionen und die anschließende Suche nach dem Kalibrierungsmuster im Bild.

Beim Start der Klasse über den `CalibrationController` kann ihr als Argument durch den Benutzer eine Verzögerung in Sekunden übergeben werden. Dies hat den Hintergrund, dass einige Kamerasensoren eine nicht unerhebliche Verzögerung zwischen der Bildaufnahme und dem Veröffentlichen über einen Topic aufweisen. Bei einer zu geringen Verzögerung zwischen dem Erreichen der Position und der Auswertung der Bilder kann der Arm auf den Bildern noch in Bewegung sein. Dadurch wird das Muster entweder an der falschen Stelle ausgewertet oder es ist so unscharf abgebildet, dass es nicht erkannt werden kann.

Die Klasse führt nach ihrer Initialisierung nichts weiter aus sondern stellt dem `calibrationcontroller` die Methode `execute_different_orientations` zur Verfügung. Dieser Methode muss eine Position übergeben werden, an die das Kalibrierungsmuster bewegt werden soll. Für diese Position wird dann zunächst ein Bild ohne zusätzliche Neigung gemacht. Anschließend wird das Bild in 10°-Schritten nach oben, unten, links und rechts geneigt, wobei für jede Orientierung wieder ein Bild aufgenommen wird. Dazu ruft der `MovementController` die Methode `take_picture_with_orientation` auf. Dieser Methode wird ebenfalls die Position übergeben und zusätzliche die gewünschte horizontale und vertikale Neigung. Zunächst wird also diese Methode ohne zusätzliche Neigung aufgerufen, anschließend mit den unterschiedlichen Varianten.

Die Methode `take_picture_with_orientation`-Methode schickt zunächst an den MoveArm Actionserver einen neuen Auftrag mit der übergebenen Position und Orientierung. Anschließend wartet sie auf die Rückmeldung vom Server. Ist diese negativ, gibt sie auch der Methode `execute_different_orientations`, die sie aufgerufen hat, eine negative Rückmeldung. Wenn der Arm erfolgreich an die gewünschte Position bewegt wurde, wartet sie die konfigurierte Zeit ab, bis sie dem FindCaltab Actionserver einen Auftrag schickt, um in den neusten Bildern nach dem Kalibrierungsmuster zu suchen. Die Rückmeldung von diesem Server ist wieder dafür ausschlaggebend, ob die Methode eine positive oder negative Rückmeldung zurückgibt.

Wenn der Aufruf der `take_picture_with_orientation`-Methode ohne zusätzliche Neigung fehlschlägt, werden die unterschiedlichen Neigungen übersprungen. Dies passiert entweder, wenn die Position nicht vom Roboter erreicht werden kann, oder ein Fehler an einem anderen Teil der Software auftritt.

Nachdem die `execute_different_orientations`-Methode alle unterschiedlichen Orientierungen durchgegangen oder bereits an der ersten Position gescheitert ist, gibt sie die Anzahl der Bilder zurück, in denen das Muster gefunden wurde.

## 4.5 CalibrationController

Diese Klasse wurde auch in Python entwickelt. Sie errechnet die verschiedenen Positionen, an die das Kalibrierungsmuster bewegt werden soll, übergibt diese nacheinander an den `MovementController` und ruft zum Schluss den Calibrate Actionserver auf.

Beim Aufruf dieser Klasse müssen ihr mehrere Argumente übergeben werden. Dazu gehören die Größe des Kamerasensors, die Brennweite der Kamera, die Größe des Kalibrierungsmusters und die Position der Kamera in Relation zur Basis des Arms. Zusätzlich kann der Benutzer die Faktoren bestimmen, mit denen die drei unterschiedlichen Entfernungen des Kalibrierungsmusters zur Kamera bestimmt werden.

Die Entfernung des Musters zur Kamera wird nicht absolut angegeben. Stattdessen wird die Höhe des Musters im Bild zur Entfernungsbestimmung benutzt. Die Entfernung, bei der die Höhe des Musters im Bild 80% der gesamten Bildhöhe ausmacht, ist die kurze Entfernung. Bei der mittleren Entfernung beträgt die Höhe des Musters 50%. Bei der weiten Entfernung sind es nur noch 30%. Je nach Sensorgröße und Brennweite sind diese Werte allerdings nicht geeignet, die Entfernungen könnten zu nah beieinander liegen oder aber sie sind so groß, dass die Positionen nicht mehr für den Arm erreichbar sind. In diesen Fällen müssen die Faktoren an die Situation angepasst werden.

Bei der Initialisierung wird zunächst ein Objekt vom Typ `MovementController` erstellt. Danach folgt die Bestimmung der Positionen für das Kalibrierungsmuster.

Dazu werden im ersten Schritt die drei unterschiedlichen Entfernungen des Musters zur Kamera benötigt. Dies geschieht mit der folgenden Formel. $h_{Muster}$ definiert die Höhe des

Musters in Metern, $h_{Sensor}$ die Höhe des Sensors in Metern, $f$ die Brennweite in Metern und der Faktor definiert, wie hoch das Muster im Bild sein soll. Das Ergebnis $d$ ist ebenfalls in Metern.

$$d = \frac{h_{Muster} \times f}{h_{Sensor} \times factor} \tag{4.5}$$

In diese Formel werden die oben genannten Werte eingesetzt, also im Standardfall 0,8 , 0,5 und 0,3.

Nun kann die Position mit der kurzen Distanz bestimmt werden. Diese besteht aus den Koordinaten $\begin{pmatrix} d \\ 0 \\ 0 \end{pmatrix}$, da das Muster mittig vor der Kamera mit der soeben berechneten Distanz zu sehen seien soll.

Die anderen Positionen sind aufwändiger zu bestimmen, da bis auf eine alle anderen nicht zentriert vor der Kamera sind. Für die mittlere Distanz $d$ wird jetzt berechnet, wie hoch und breit das Sichtfeld ist. Dazu werden die folgenden Formeln benutzt. Alle Werte sind in Metern.

$$h_{Sichtfeld} = \frac{h_{Sensor} \times d}{f} \tag{4.6}$$

$$b_{Sichtfeld} = \frac{b_{Sensor} \times d}{f} \tag{4.7}$$

Nun ist bekannt, wie groß das Sichtfeld zu dieser Distanz ist. Das Muster soll sich in jedem Quadrant des Bildes befinden. Eine Position wird beispielhaft für den Quadranten oben links gezeigt.

$$\begin{pmatrix} d \\ -b_{Sichtfeld}/2 - b_{Muster}/2 \\ h_{Sichtfeld}/2 - h_{Muster}/2 \end{pmatrix} \tag{4.8}$$

Die weiteren Positionen für diese Distanz werden durch das Hinzufügen oder Weglassen der Vorzeichen bestimmt.

Die Berechnung der Positionen für die lange Distanz wird analog dazu durchgeführt. Das Bild wird in diesem Fall jedoch nicht in vier sondern neun Bereiche aufgeteilt. Zunächst wird wieder das Sichtfeld bestimmt, siehe Gleichung 4.6 und Gleichung 4.7. Die Positionen in den Ecken werden genau so wie in Gleichung 4.8 bestimmt. Für die Positionen, die auf einer der Achsen liegen, wird der entsprechende y- oder z-Wert auf null gesetzt.

TODO: RVIZ mit Markern für die Positionen

Nun wurden alle Positionen bestimmt. Diese sind aber in Relation zur Kamera berechnet worden. Um den Roboterarm an die Positionen zu bewegen, müssen sie aber in Relation zur Basis des Roboters vorliegen. Daher werden diese Positionen mit Hilfe von `TF` in das Koordinatensystem des Roboters übertragen.

Dazu muss zunächst die Verschiebung vom Koordinatensystem des Roboters zu dem der Kamera bestimmt werden. Die Position ist bekannt, da der Benutzer diese beim Start des Programms angegeben hat. Die Orientierung wird wie in den Gleichungen Gleichung 4.2, Gleichung 4.3 und Gleichung 4.4 berechnet. Es wird angenommen, dass der Bildmittelpunkt 0,45m über der Basis des Roboters liegt. Die so berechnete Verschiebung vom Kamerakoordinatensystem zum Roboterkoordinatensystem wird `TF` bekannt gemacht. Im Anschluss kann man mit der `transformPoint`-Methode jede Position in das Koordinatensystem des Roboters überführt werden.

Da nun die Positionen im korrekten Koordinatensystem vorhanden sind, kann geprüft werden, wie viele für den Arm erreichbar sind. Wie bereits erwähnt kann es je nach Kameratyp vorkommen, dass die Positionen zu weit entfernt sind. Diese Überprüfung findet über eine Abschätzung statt. Dazu wird die Distanz der verschiedenen Positionen zur Basis des Roboterarms bestimmt. Diese hat eine Reichweite von ungefähr 0,9m. Ist eine Position weiter entfernt, ist sie wahrscheinlich nicht erreichbar. Trifft dies auf mehr als 75% der Positionen zu, wird der Benutzer darüber informiert. In diesem Fall wird die Kalibrierung fortgesetzt, die Ergebnisse sollten aber überprüft werden, da die Anzahl der Bilder zu gering sein kann, nicht das ganze Bild oder alle Entfernungen abgedeckt wurde und dadurch die Qualität der Kalibrierung sinkt.

Nachdem die Erreichbarkeit der Positionen überprüft wurde, wird jede Position mit der Methode `execute_different_orientations` vom `MoveMentcontroller` aufgerufen. Wie bereits erwähnt koordiniert diese Methode die Bewegung des Arms und die Aufnahme der Bilder.

Wurden alle Positionen abgearbeitet, wird zum Schluss der Calibrate-Actionserver aufgerufen, um die Resultate der Kalibrierung zu erhalten. Danach wird das Programm beendet. TODO: Resultate nicht nur in der Konsole anzeigen sondern auch abspeichern

# Literaturverzeichnis


[And15] Andersen, Thomas T.: Optimizing the Universal Robots ROS driver. / Technical University of Denmark, Department of Electrical Engineering. Version: 2015. http://orbit.dtu.dk/en/publications/optimizing-the-universal-robots-ros-driver(20dde139-7e87-4552-8658-dbf2cdaab.html. 2015. – Forschungsbericht

[CSC12] Chitta, S.; Sucan, I.; Cousins, S.: MoveIt! [ROS Topics]. In: *IEEE Robotics Automation Magazine* 19 (2012), March, Nr. 1, S. 18–19. http://dx.doi.org/10.1109/MRA.2011.2181749. – DOI 10.1109/MRA.2011.2181749. – ISSN 1070–9932

[EGH+17] Edwards, Shaun; Glaser, Stuart; Hawkins, Kelsey; Meeussen, Wim; Messmer, Felix: *universal_robot*. https://github.com/ros-industrial/universal_robot, 2017

[QCG+09] Quigley, Morgan; Conley, Ken; Gerkey, Brian P.; Faust, Josh; Foote, Tully; Leibs, Jeremy; Wheeler, Rob; Ng, Andrew Y.: ROS: an open-source Robot Operating System. In: *ICRA Workshop on Open Source Software*, 2009

[Zha00] Zhang, Z.: A flexible new technique for camera calibration. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000), Nov, Nr. 11, S. 1330–1334. http://dx.doi.org/10.1109/34.888718. – DOI 10.1109/34.888718. – ISSN 0162–8828

# Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum Unterschrift