

Universität Bremen

Fachbereich 3

Robotergestütztes Kalibrierungssystem für Sensoren

Bachelorarbeit

Benny Prange
Matrikel-Nummer 2597237
10.08.2017

Betreuer Alexis Maldonado
Erstprüfer Prof. Dr. Michael Beetz
Zweitprüfer Dr. Karsten Sohr

Inhaltsverzeichnis

Abbildungsverzeichnis	IV
Tabellenverzeichnis	V
1 Einleitung	1
2 Grundlagen	3
2.1 Kameramodell	3
2.2 Verzeichnung	4
2.3 Kalibrierung	6
2.3.1 Kalibrierungsmuster	7
2.3.2 Bildaufnahme	9
2.3.3 Berechnung der Parameter	9
2.3.4 Weitere benutzte Frameworks	10
3 Architektur	13
3.1 Gesamtübersicht	13
3.2 Robot Hardware Interface Node	13
3.3 Moveit Path Planning Node	14
3.4 Camera Interface Node	15
3.5 Client Side Path Planning Node	15
3.6 Computer Vision Node	15
3.7 Highlevel Executive Controller	16
4 Implementierung	17
4.1 Robot Hardware Interface Node	17
4.2 Moveit Path Planning Node	17

Inhaltsverzeichnis

4.3 Client Side Path Planning Node	19
4.4 Computer Vision Node	20
4.5 MovementController	25
4.6 High Level Executive	27
5 Evaluierung	34
6 Fazit und Ausblick	38
Literaturverzeichnis	40

Abbildungsverzeichnis

2.1	Beispiele für radialsymmetrische Verzeichnung, von Wikipedia: [Fan09]	5
2.2	Die durchgezogenen Linien sind ohne Verzeichnung, die gestrichelten Linien zeigen die tangentiale Verzeichnung. Aus: [WCH92]	5
2.3	Schachbrettmuster als Kalibrierungsmuster	7
2.4	Punkte als Kalibrierungsmuster	8
3.1	Übersicht über die Architektur. Die Pfeile verdeutlichen die Kommunikation zwischen den Knoten.	14
4.1	Befestigung des Kalibrierungsmusters.	18
4.2	Ablaufdiagramm für die Client Side Path Planning Node.	21
4.3	Ablaufdiagramm für den find_calibration_object Actionserver	24
4.4	Das Originalbild.	25
4.5	Ein Ausschnitt des Bildes mit dem eingezeichneten Kalibrierungsmuster. Man sieht die feinen Linien, die um das schwarze Rechteck und in die Kreise eingefügt wurden.	26
4.6	Ablaufdiagramm für den calculate_parameters Actionserver.	27
4.7	Ablaufdiagramm für den MovementController.	28
4.8	Die weißen Kugeln sind die verschiedenen Positionen für das Kalibrierungsmuster, die grüne Box rechts ist die Kamera.	31
4.9	Ablaufdiagramm für den High Level Executive.	33
5.1	Der Aufbau zur Kalibrierung aus Sicht der Kamera.	35
5.2	Der Aufbau zur Kalibrierung von der Seite betrachtet.	35

Tabellenverzeichnis

4.1	Beschreibung der Parameter für die Computer Vision Node	22
4.2	Beschreibung der Parameter für den High Level Executive	29
4.3	Beschreibung der Parameter für den High Level Executive	33
5.1	Ergebnisse der Kalibrierung	37

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 Entfernung und Orientierungen zu sehen ist. Schließlich lässt sich aus diesen Daten die Kameramatrix berechnen.

1 Einleitung

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 Muster jeweils auf einem Stativ zu befestigen, um scharfe Bilder zu erzeugen. Damit ist gewährleistet, dass die Erkennung des Musters nicht von unscharfen Bildern gestört wird, weil der Mensch die Kamera oder das Muster nicht still genug gehalten hat. Allerdings wird der Vorgang durch das Verstellen der Stativen 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 Kalibrierungsmuster befestigt ist. Anschließend startet er das Programm, welches den Arm zu verschiedene Positionen bewegt, sodass die Kamera das Muster aus mehreren Entfernung und Orientierungen aufnehmen kann. Zum Schluss berechnet das Programm die benötigten Parameter und der Benutzer kann die Kamera mit diesen Parametern einsetzen.

In Kapitel 2 werden die Grundlagen zur Kalibrierung und der eingesetzten Frameworks beschrieben. In Kapitel 3 wird der Aufbau des Gesamtprogramms dargestellt. Kapitel 4 geht detailliert auf die Implementierung der Programme ein. In Kapitel 5 wird eine Kalibrierung exemplarisch durchgeführt. Im letzten Kapitel wird die Arbeit zusammengefasst; mögliche Ansätze zur Verbesserung und Erweiterung werden diskutiert.

2 Grundlagen

Kameras sind viel genutzte Sensoren, da sie teilweise recht günstig hergestellt werden und sie aufgrund ihrer Ähnlichkeit zum menschlichen Auge geeignete Sensoren sind, um die Umgebung zu erfassen. Da sich aber gezeigt hat, dass die Toleranzen bei der Produktion, der Aufbau der Kameras und insbesondere ihrer Linsen, 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. Dieses Modell wird auch von OpenCV, siehe [Ope], und HALCON, siehe [MVT17] benutzt.

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.

2 Grundlagen

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

mit

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

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.

Es gibt zwei verschiedene Formen von Verzeichnung. Ist die Verzeichnung radialsymmetrisch zum Bildmittelpunkt, spricht man von kissen- oder tonnenförmiger Verzeichnung, siehe Abbildung 2.1.

2 Grundlagen

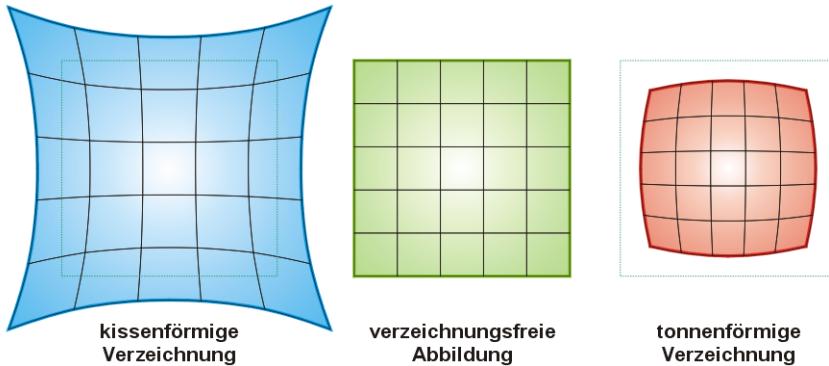


Abbildung 2.1: Beispiele für radialsymmetrische Verzeichnung, von Wikipedia: [Fan09]

Die zweite Form von Verzeichnung beschreibt die tangentiale Verzeichnung. Siehe dazu Abbildung 2.2.

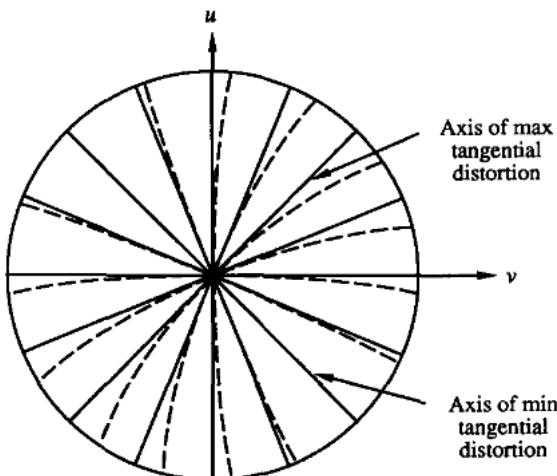


Abbildung 2.2: Die durchgezogenen Linien sind ohne Verzeichnung, die gestrichelten Linien zeigen die tangentiale Verzeichnung. Aus: [WCH92]

Es existieren verschiedene Modelle, um diese Verzeichnungen zu beschreiben. Das Divisionsmodell kann die radialsymmetrische Verzeichnung beschreiben und herausrechnen. Dazu wird nur ein einziger Parameter k benötigt. Dies hat den Vorteil, dass nur relativ wenig Bilder benötigt werden, um einen guten Wert für k zu erhalten. Der Nachteil an diesem Modell ist, dass nur die radialsymmetrische Verzeichnung erfasst wird und nicht

2 Grundlagen

die tangentiale Verzeichnung. Zudem ist die radialsymmetrische Verzeichnung nicht immer über das ganze Bild gleich verteilt, sodass das Modell nicht jede Situation optimal beschreibt.

Das Polynommodell kann hingegen beide Formen der Verzeichnung beschreiben. Dazu werden insgesamt fünf Parameter benötigt, welche im Folgenden beschrieben werden. Die radialsymmetrische Verzeichnung kann in diesem Modell feiner bestimmt werden. Dies ist hilfreich, da diese Verzeichnung nicht immer gleichmäßig über das ganze Bild auftritt, sondern abhängig von der Entfernung zum Mittelpunkt unterschiedlich stark sein kann. Die tangentiale Verzeichnung wird ebenfalls beachtet. Der Nachteil ist, dass mehr Bilder benötigt werden, um gute Ergebnisse zu erhalten. Außerdem ist es bei diesem Modell sehr wichtig, dass unterschiedliche Distanzen betrachtet werden und das Muster in jedem Ausschnitt des Bildes zu sehen ist, da ansonsten die Werte für die nicht betrachteten Teile des Bildes extrem schlechte Ergebnisse liefern können.

Um die radialsymmetrische Verzeichnung herauszurechnen, werden die drei Faktoren k_1, k_2, k_3 benötigt. Sind diese gegeben, lässt sich die Verzeichnung mit den beiden folgenden Gleichungen korrigieren. r ist der Abstand vom Bildmittelpunkt zu dem Punkt, der korrigiert werden soll. x', y' sind die falsch abgebildeten Punkte, x, y sind die korrigierten Punkte.

$$x = x'(1 + k_1r^2 + k_2r^4 + k_3r^6) \quad (2.3)$$

$$y = y'(1 + k_1r^2 + k_2r^4 + k_3r^6) \quad (2.4)$$

Um die tangentiale Verzeichnung zu korrigieren, werden die beiden folgenden Gleichungen benutzt. Dazu werden die Faktoren p_1, p_2 benötigt.

$$x = x' + (2p_1xy + p_2(r^2 + 2x^2)) \quad (2.5)$$

$$y = y' + (p_1(r^2 + 2y^2) + 2p_2xy) \quad (2.6)$$

2.3 Kalibrierung

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

2 Grundlagen

Schritten.

2.3.1 Kalibrierungsmuster

Man benötigt ein Kalibrierungsmuster, von dem einige Aufnahmen gemacht werden. Es gibt verschiedene Kalibrierungsmuster, die benutzt werden können. 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 2.3. In diesem Muster haben die Rechtecke eine Kantenlänge von exakt 3cm. Insgesamt sind 6×8 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 5×7 , also 35 Kalibrierungspunkte.

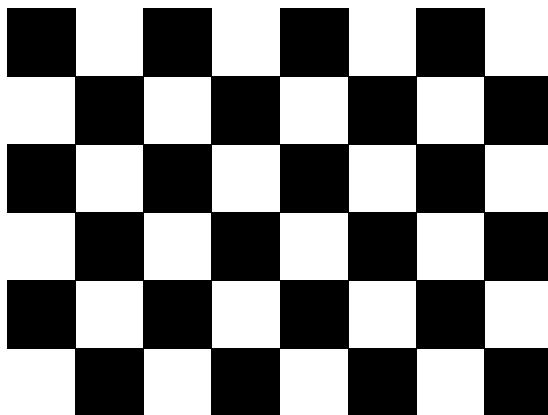


Abbildung 2.3: Schachbrettmuster als Kalibrierungsmuster

Ein anderes Kalibrierungsmuster besteht aus Kreisen, die regelmäßig versetzt zueinander positioniert sind, siehe Abbildung 2.4. Die Mittelpunkte und Ränder der Kreise sind in diesem Fall die Kalibrierungspunkte. In diesem Muster sind 10×11 Kreise eingezeichnet, sodass deutlich mehr Informationen für die Kalibrierung in einer Aufnahme vorhanden sind.

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

2 Grundlagen

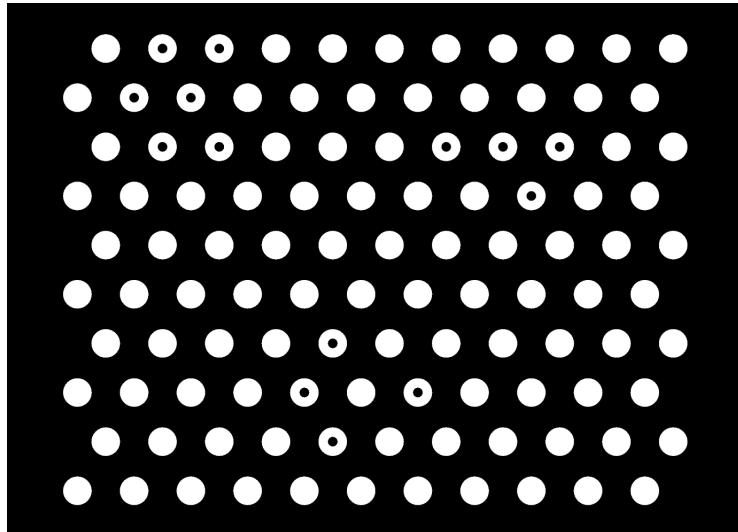


Abbildung 2.4: Punkte als Kalibrierungsmuster

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. Der Hersteller von HALCON, MVtec, bietet aus diesem Grund Kalibrierungsmuster zum Kauf an. Dies werden teilweise aus Aluminium hergestellt. MVtec garantiert dabei zum einen, dass die Abweichungen sehr gering sind, zum anderen können die Abweichungen exakt gemessen werden und der Kunde erhält zu dem Muster eine individuelle Datei, in der die exakten Abmessungen enthalten sind. Diese Datei kann dann zur Kalibrierung verwendet werden.

2 Grundlagen

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.

Wenn das Polynommodell zur Beschreibung der Verzeichnung benutzt wird, besteht eine erhöhte Gefahr der Überanpassung. Dieser Begriff beschreibt das Problem, wenn die gefundenen Parameter zwar sehr gut auf die Situation passen, die zur Kalibrierung untersucht wurde, aber in anderen Situationen sehr schlechte Ergebnisse liefern. In diesem Kontext ist daher darauf achten, dass das Muster in verschiedenen Distanzen zur Kamera aufgenommen wurde. Ansonsten könnten die Parameter schlechte Ergebnisse erzeugen, wenn Objekte betrachtet werden, die deutlich näher oder weiter entfernt von der Kamera sind, als das Muster.

Ist das Muster nur an einer Stelle im Bild zu sehen und an anderen Stellen gar nicht, kann ebenfalls eine Überanpassung auftreten. Daher muss das Muster über alle Aufnahmen verteilt im ganzen Bild sichtbar sein.

Die Kalibrierung wird zusätzlich verbessert, wenn das Muster nicht immer orthogonal zur Kamera steht, sondern auch horizontal oder vertikal geneigt wird.

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.

2 Grundlagen

Anschließend werden in jedem aufgenommenen Bild die Kalibrierungspunkte 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 anzuwenden ist. Außerdem ist ein Kalibrierungsassistent enthalten, der die aufgenommen Bilder vor der Berechnung der Parameter anhand ihrer Bildschärfe, Belichtung und weiterer Eigenarten bewertet. Des Weiteren ist es von Vorteil, dass HALCON nicht das ganze Muster sehen muss, sondern bereits Teile davon zur Kalibrierung benutzt werden können. Teilweise haben die Algorithmen eine geringere Laufzeit. In der Arbeitsgruppe "Künstliche Intelligenz" von Prof. Beetz wird HALCON eingesetzt.

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.

2 Grundlagen

Möglich ist die Einbindung fertiger Pakete, die von anderen Benutzern bereitgestellt werden, zum Beispiel MoveIt!, siehe Abschnitt 2.3.4. Außerdem kann man eigene Programme schreiben und diese in ROS integrieren. ROS wird nicht nur für so einfache Roboter wie der UR5-Arm benutzt, sondern auch für komplexere Roboter wie der PR2 oder auch in autonom fahrenden Autos.

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`, zu deutsch `Knoten`, 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 eine andere Kamera einsetzt.

2 Grundlagen

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.

UR5

Der UR5 ist ein Roboterarm von Universal Robots. Dieser ist in drei verschiedenen Größen erhältlich, der UR5 ist die mittlere Größe. Alle UR-Roboter können kollaborierend mit Menschen arbeiten, sie müssen also nicht räumlich getrennt von Menschen arbeiten. Darüber hinaus lassen sie sich mit dem angeschlossenen Touchpannel relativ einfach steuern und programmieren. Mit einem Knopf am Touchpad kann man den Roboter durch einfaches Bewegen des Arms an die gewünschte Position bringen und diese als Wegpunkt oder Position abspeichern.

Zusätzlich wurde als ROS-Paket ein Interface geschrieben, um den Roboter auch über ROS und MoveIt! steuern zu können.

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. Jeder Knoten stellt dabei eine ROS-Node dar, also ein einzelnes Programm. Knoten, 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.

Um die Übersicht über die Architektur möglichst generisch zu halten, haben die Nodes einen sehr deskriptiven Namen erhalten. Die Programme, die in der Anwendung gestartet werden, haben daher teilweise andere Namen.

3.2 Robot Hardware Interface Node

Für diese Node wird das Paket `ur_modern_driver` verwendet, 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, waren einige wenige Anpassungen am Sourcecode nötig, um es in diesem Projekt benutzen zu können.

3 Architektur

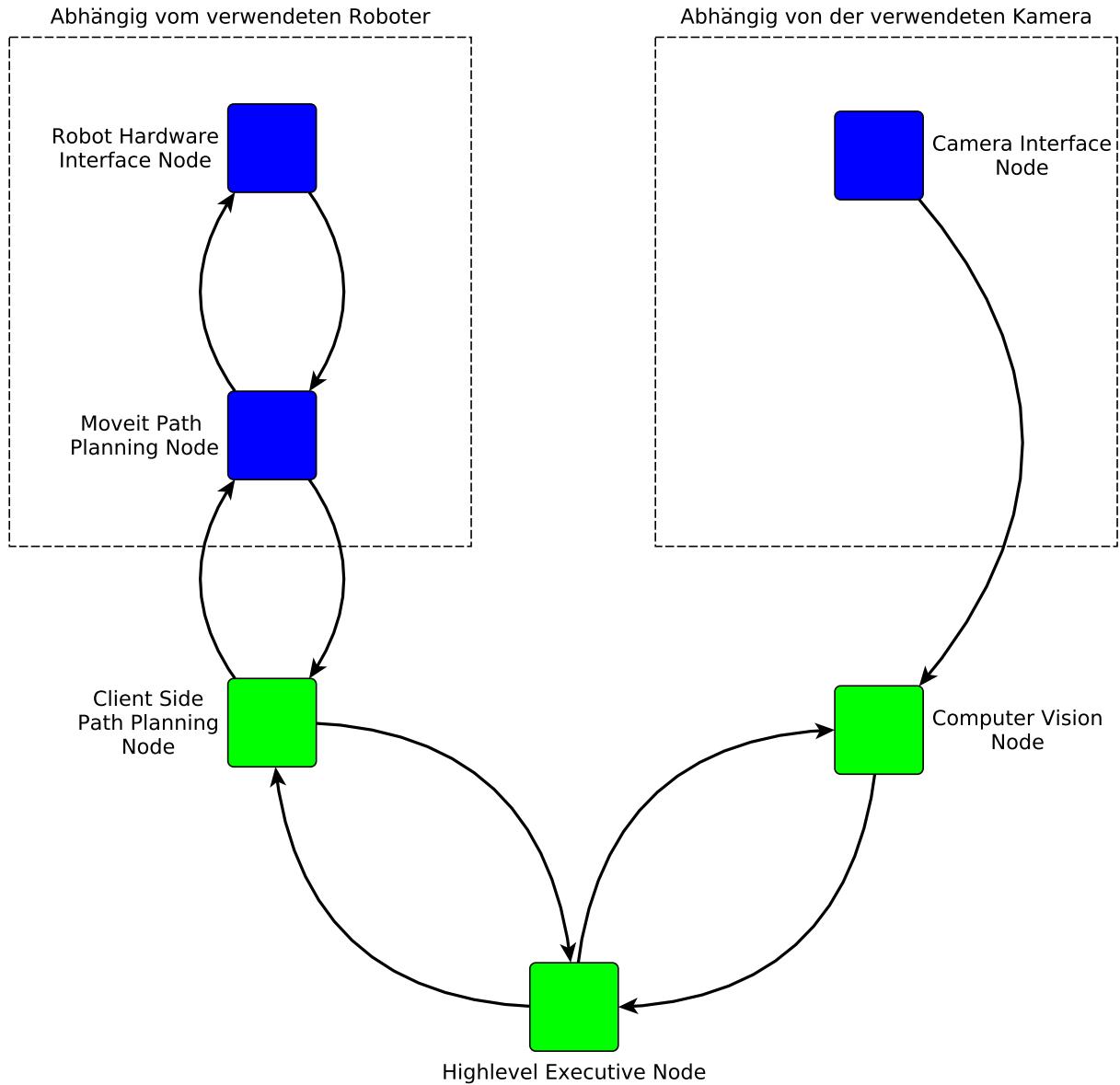


Abbildung 3.1: Übersicht über die Architektur

3.3 Moveit Path Planning Node

Auch für diese Node wird ein öffentlich verfügbares Paket genommen: `universal_robot`, siehe [EGH⁺17]. In dieser Node ist MoveIt! integriert, welches zum Bewegen des Roboterarms benutzt wird. Siehe Abschnitt 2.3.4 für weitere Details über MoveIt!.

3.4 Camera Interface Node

Auch diese Node ist als Paket `libuvc_camera` in ROS verfügbar. Sie dient dazu, das Bild der angeschlossenen Kamera 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 auf einen festen Wert eingestellt, damit der Bereich, in dem sich das Kalibrierungsmuster befinden wird, scharf zu sehen ist und sich die Brennweite im Verlauf des Vorgangs nicht mehr ändert.

3.5 Client Side Path Planning Node

Dieses Programm wurde im Rahmen dieser Bachelorarbeit selbst entwickelt. Es dient dazu, der Highlevel Executive Node 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 diese Node den Highlevel Executive Controller, ob der Auftrag erfolgreich ausgeführt wurde.

3.6 Computer Vision 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.

3 Architektur

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 Highlevel Executive Controller 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 Highlevel Executive Controller zurückgegeben.

3.7 Highlevel Executive Controller

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 nach oben, unten, rechts und links geneigt. Hat der Actionserver den Auftrag erfolgreich ausgeführt, wird der Computer Vision 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 dargestellt.

4.1 Robot Hardware Interface Node

An diesem Programm wurden nur kleine Änderungen vorgenommen, um zum Einen das Paket mit ROS Kinetic benutzen zu können und zum Anderen um die Bewegung des Arms etwas gleichmäßiger durchführen zu können.

4.2 Moveit Path Planning Node

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 verwendet, für eine gewünschte Position und Orientierung des Roboterarms die dazu erforderlichen Winkel der Gelenke zu berechnen. Während der Versuche war es teilweise zu beobachten, dass MoveIt! keinen oder nur einen viel zu komplizierten Weg von der Start- zur Zielkonfiguration gefunden hat. Dieses Problem konnte teilweise durch diesen Austausch mit TRAC_IK 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.

4 Implementierung



Abbildung 4.1: Befestigung des Kalibrierungsmusters.

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. Siehe Abbildung 4.1 für ein Bild von der Befestigung des Kalibrierungsmusters.

4 Implementierung

4.3 Client Side Path Planning Node

Diese Node wurde in der Programmiersprache Python entwickelt und hat den Namen MovementHandler.py. Sie hat den Auftrag, die Bewegung des Arms zu koordinieren und dazu mit MoveIt! zu kommunizieren. Die Node befindet sich zusammen mit der High Level Executive Node in dem ROS-Paket calibration_executive.

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

Die Aufgabe move_arm 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:

$$\vec{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad (4.1)$$

Um nun die Orientierung zu errechnen, subtrahiert man zunächst den Vektor für die 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.

$$\vec{p}_{Diff} = \vec{p}_{Roboter} - \vec{p}_{Kamera} \quad (4.2)$$

Mit diesem Vektor lässt sich nun der Gierwinkel α und Neigungswinkel β bestimmen, damit das Muster orthogonal zur Kamera steht:

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

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

4 Implementierung

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 γ bestimmt werden. In diesem Anwendungsfall soll das Kalibrierungsmuster immer nach außen zeigen und parallel zum Boden stehen, damit die Reichweite des Roboters erhöht wird. Dazu wird die folgende Gleichung gelöst:

$$a = y_{\text{Roboter}} \times x_{\text{Kamera}} - y_{\text{Kamera}} \times x_{\text{Roboter}} \quad (4.5)$$

Ist $a > 0$, so befindet sich das Muster von der Kamera aus gesehen in der rechten Hälfte des Bildes. Damit das Muster nach außen zeigt, muss der Rollwinkel 0° betragen. Ist $a < 0$, befindet sich das Muster in der linken Hälfte und der Rollwinkel wird auf 180° gesetzt.

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. Siehe Abbildung 4.2 für ein Ablaufdiagramm.

4.4 Computer Vision Node

Diese Node wurde in der Programmiersprache C++ programmiert und liegt im Paket `calibration_perception` mit dem Dateinamen `caltab_detector_node.cpp`. Sie dient da-

4 Implementierung

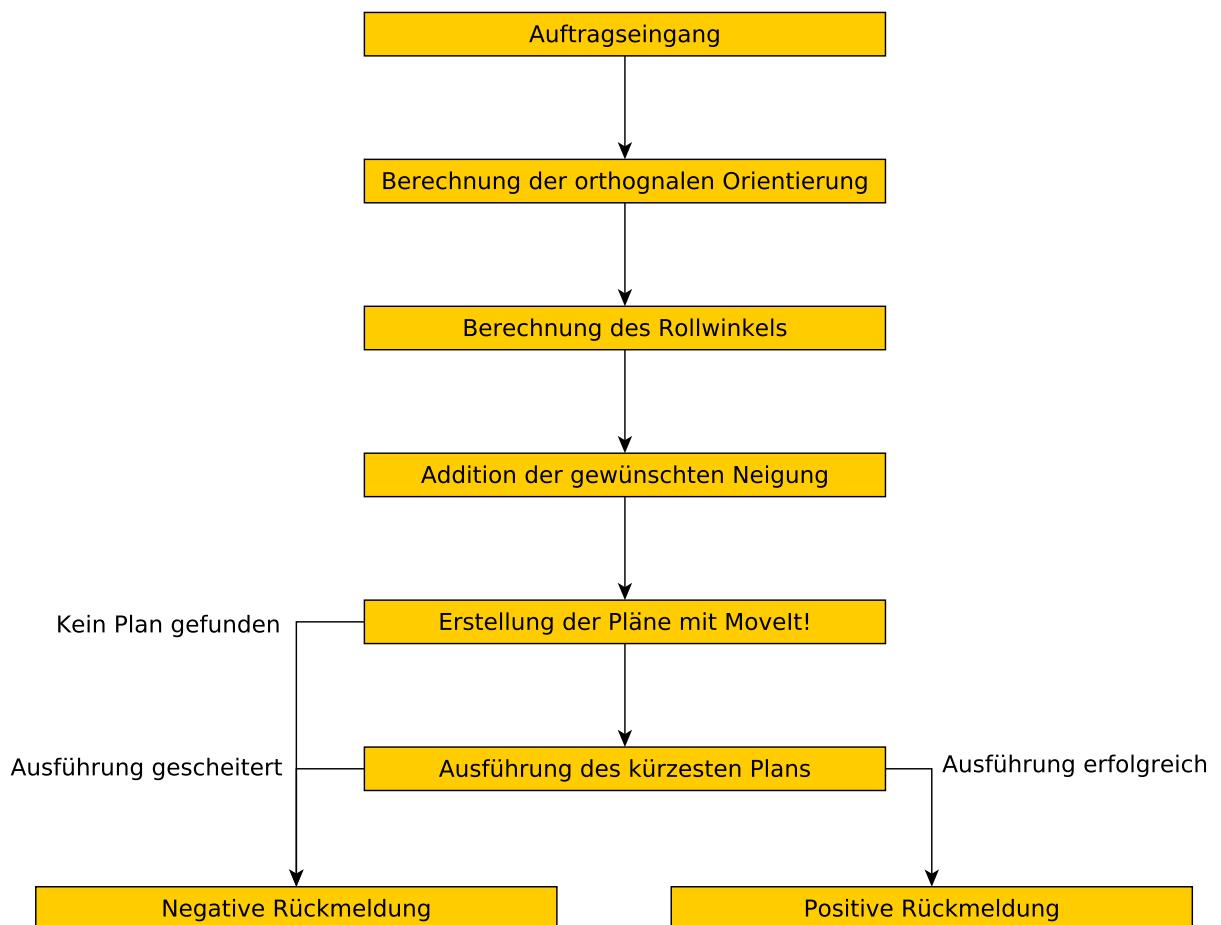


Abbildung 4.2: Ablaufdiagramm für die Client Side Path Planning Node.

4 Implementierung

Parametername	Beschreibung
focal_length	Brennweite in Metern
sensor_size x	Größe des Sensors entlang der x-Achse in Metern
sensor_size y	Größe des Sensors entlang der y-Achse in Metern
resolution x	Anzahl der Pixel des Bildes entlang der x-Achse
resolution y	Anzahl der Pixel des Bildes entlang der y-Achse
image_topic	Der Name des Topics, auf dem die Bilder von der Kamera veröffentlicht werden
description_file	Der Pfad zu der Datei, die die Beschreibung für das Kalibrierungsmuster enthält

Tabelle 4.1: Beschreibung der Parameter für die Computer Vision Node

zu, die von der Kamera übermittelten Bilder auszuwerten und zur Kalibrierung vorzuhalten.

Vor dem Start muss der Benutzer über die Datei `caltab_detector/config/parameters.yaml` einige Parameter konfigurieren. Diese sind in Tabelle 4.1 aufgeführt.

Über das angegebene Topic erhält das Programm die Bilder der Webcam. Zusätzlich stellt das Programm zwei Actionserver zur Verfügung. Der Actionserver mit dem Namen `find_calibration_object` 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. Außerdem wird ein Timeout mit angegeben. Hat der Server das Muster innerhalb der erlaubten Zeit und 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 der 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

4 Implementierung

für die eingesetzte Kamera hinterlegt sind. Ebenfalls wird die Datei gelesen, in der die Dimensionen und Eigenschaften des benutzten Kalibrierungsmusters definiert sind. 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 `find_calibration_object` 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 empfangene Bild 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 erstellten 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 `find_calibration_object` Actionserver einen neuen Auftrag, wird die Methode `find_calibration_objectAction` aufgerufen. Zunächst wird der boolesche 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

4 Implementierung

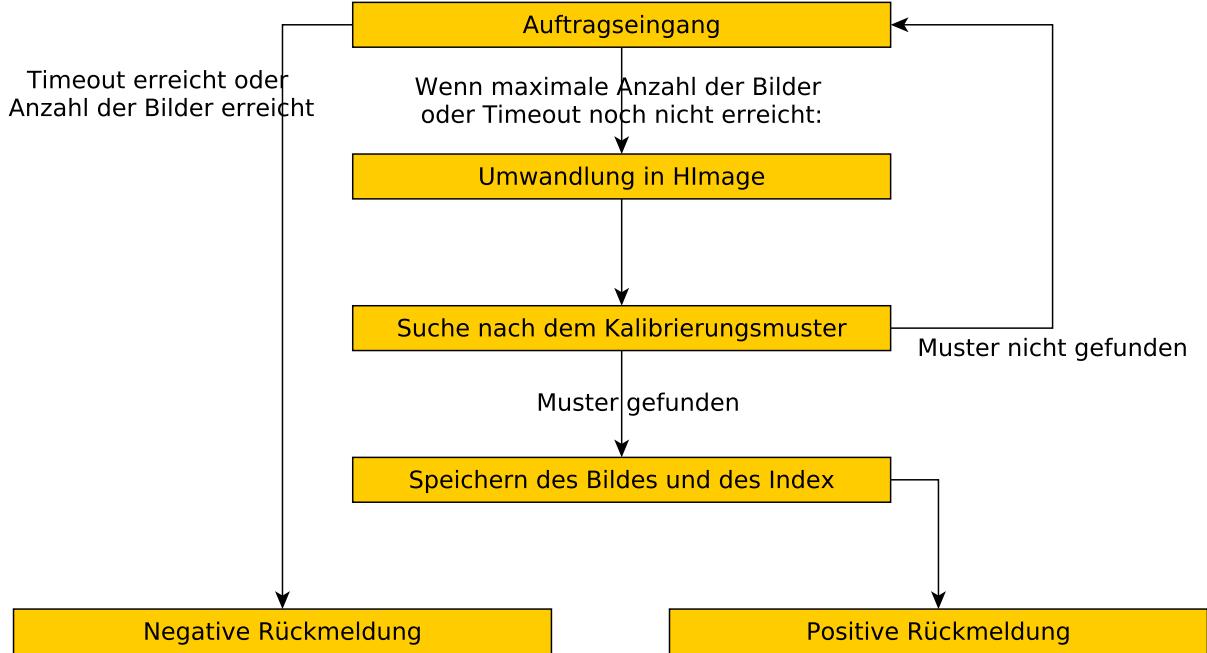


Abbildung 4.3: Ablaufdiagramm für den `find_calibration_object` Actionserver

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. Siehe dazu Abbildung 4.3.

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 erkannt wurde. Dann werden diese Informationen mit den neuen Kameraparametern korrigiert. Schließlich wird das so korrigierte Kalibrierungsmuster über

4 Implementierung

das eingelesene Bild gelegt und das Bild so auf dem Dateisystem abgespeichert. Siehe dazu Abbildung 4.4 und Abbildung 4.5 sowie für das Ablaufdiagramm Abbildung 4.6.

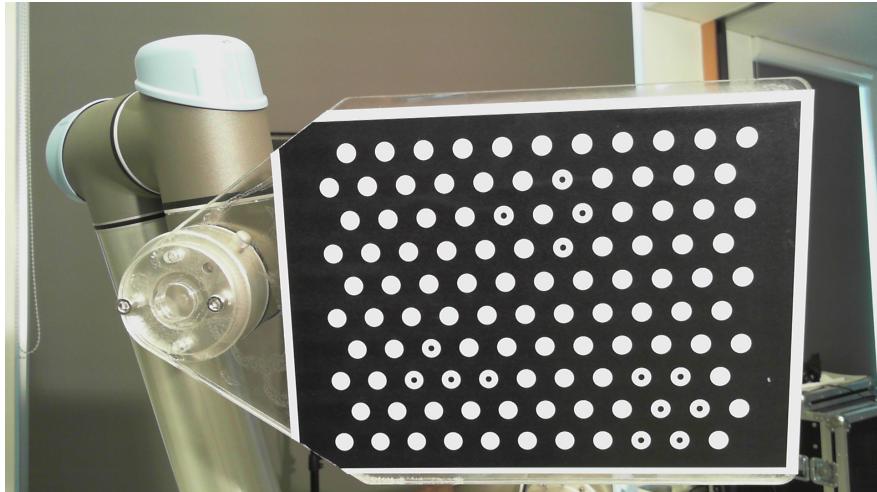


Abbildung 4.4: Das Originalbild.

4.5 MovementController

Die Klasse MovementController wurde wieder in Python entwickelt. Sie agiert als Action-client für den move_arm Actionserver und den find_calibration_object Actionserver und koordiniert die Bewegung des Arms zu den verschiedenen Positionen und die anschließende Suche nach dem Kalibrierungsmuster im Bild. Sie wird vom High Level Executive gestartet.

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.

4 Implementierung

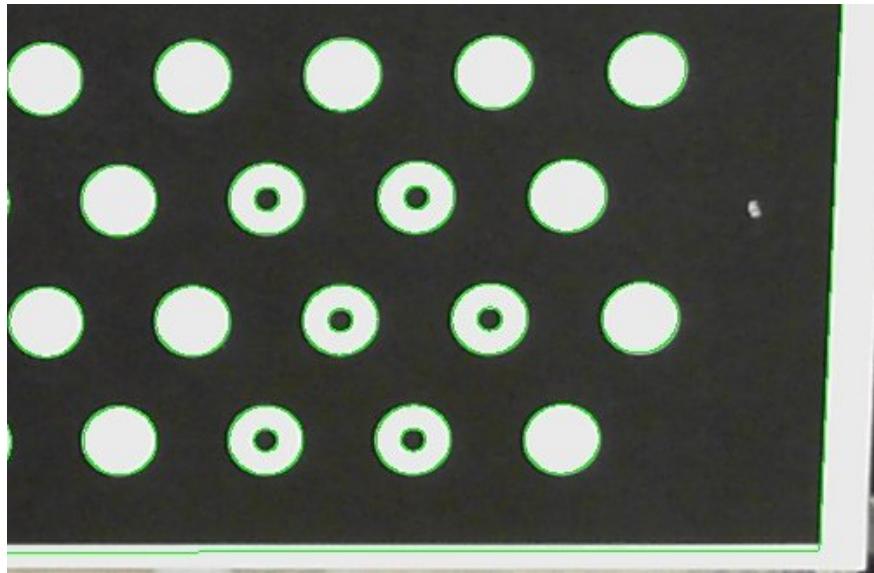


Abbildung 4.5: Ein Ausschnitt des Bildes mit dem eingezeichneten Kalibrierungsmuster. Man sieht die feinen Linien, die um das schwarze Rechteck und in die Kreise eingefügt wurden.

Die Klasse führt nach ihrer Initialisierung nichts weiter aus, sondern stellt dem High Level Executive 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 zusätzlich die gewünschte horizontale und vertikale Neigung. Zunächst wird also diese Methode ohne zusätzliche Neigung aufgerufen, anschließend mit den unterschiedlichen Varianten.

Die `take_picture_with_orientation`-Methode schickt zunächst an den `move_arm` 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 `find_calibration_object` 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

4 Implementierung

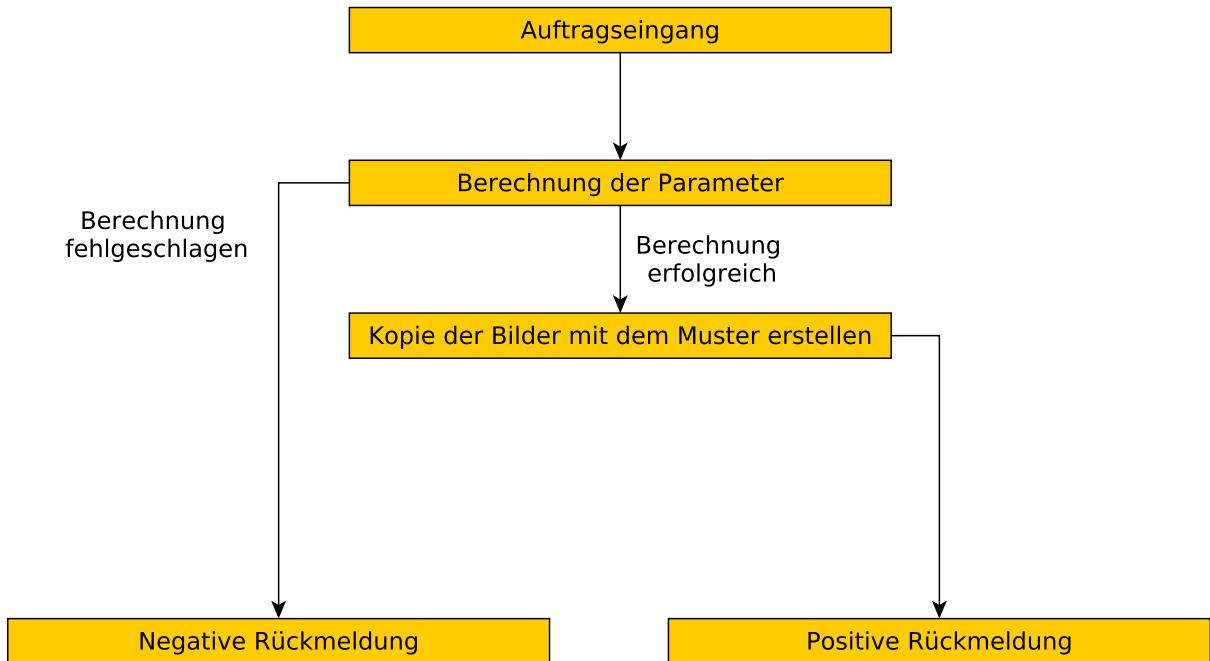


Abbildung 4.6: Ablaufdiagramm für den calculate_parameters Actionserver.

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. Siehe dazu das Ablaufdiagramm in Abbildung 4.2.

4.6 High Level Executive

Diese Klasse wurde auch in Python entwickelt. Sie errechnet die verschiedenen Positionen, an die das Kalibrierungsmuster bewegt werden soll, übergibt diese nachein-

4 Implementierung

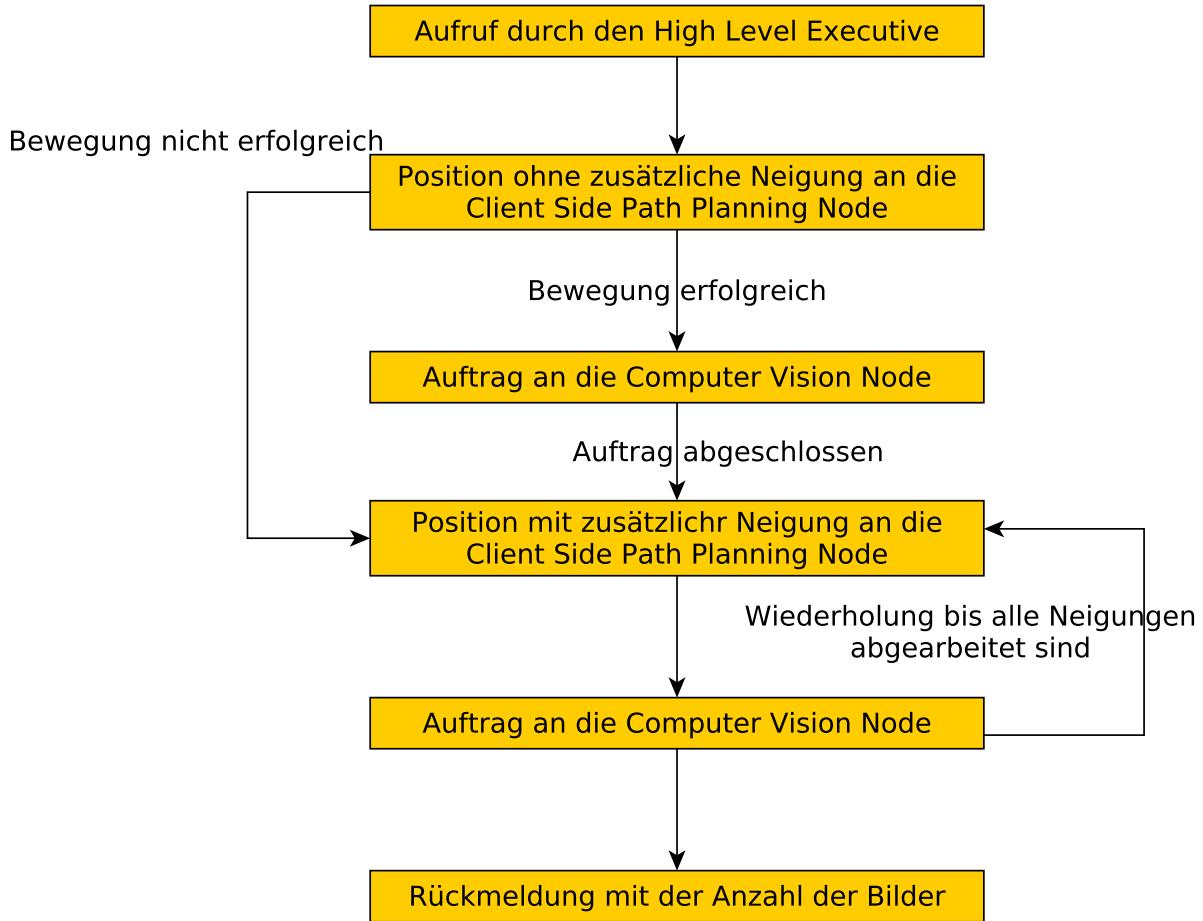


Abbildung 4.7: Ablaufdiagramm für den MovementController.

ander an den MovementController und ruft zum Schluss den Calibrate Actionserver auf.

Beim Aufruf dieser Klasse müssen ihr mehrere Parameter übergeben werden. Diese werden in der Datei `robot_assisted_calibration/config/parameters.yaml` definiert, siehe Tabelle 4.2

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

4 Implementierung

Parametername	Beschreibung
latency	Verzögerung zwischen dem Erreichen einer Position und der Auswertung der Bilder
close_distance_factor	Der Faktor, der die kurze Entfernung bestimmt
medium_distance_factor	Der Faktor, der die mittlere Entfernung bestimmt
far_distance_factor	Der Faktor, der die weite Entfernung bestimmt
focal_length	Brennweite in Metern
sensor_size_x	Größe des Sensors entlang der x-Achse in Metern
sensor_size_y	Größe des Sensors entlang der y-Achse in Metern
calibration_object_height	Die Höhe des Kalibrierungsmusters in Metern
calibration_object_width	Die Breite des Kalibrierungsmusters in Metern
camera_position	Die Position der Kamera in Relation zur Basis des Roboters
skip_orientations	zu Testzwecken kann dieser Parameter auf true gesetzt werden, um die zusätzlichen Orientierungen zu überspringen

Tabelle 4.2: Beschreibung der Parameter für den High Level Executive

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 Entfernung 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.

$$d = \frac{h_{Muster} \times f}{h_{Sensor} \times factor} \quad (4.6)$$

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

4 Implementierung

und breit das Sichtfeld ist. Dazu werden die folgenden Formeln benutzt. Alle Werte sind in Metern.

$$h_{Sichtfeld} = \frac{h_{Sensor} \times d}{f} \quad (4.7)$$

$$b_{Sichtfeld} = \frac{b_{Sensor} \times d}{f} \quad (4.8)$$

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.

$$\begin{pmatrix} d \\ -b_{Sichtfeld}/2 - b_{Muster}/2 \\ h_{Sichtfeld}/2 - h_{Muster}/2 \end{pmatrix} \quad (4.9)$$

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 4.7 und Gleichung 4.8. Die Positionen in den Ecken werden genau so wie in Gleichung 4.9 bestimmt. Für die Positionen, die auf einer der Achsen liegen, wird der entsprechende y- oder z-Wert auf null gesetzt.

Eine Übersicht über die verschiedenen Positionen sieht man in Abbildung 4.8.

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 mit der `transformPoint`-Methode jede Position in das Koordinatensystem des Roboters überführt werden.

4 Implementierung

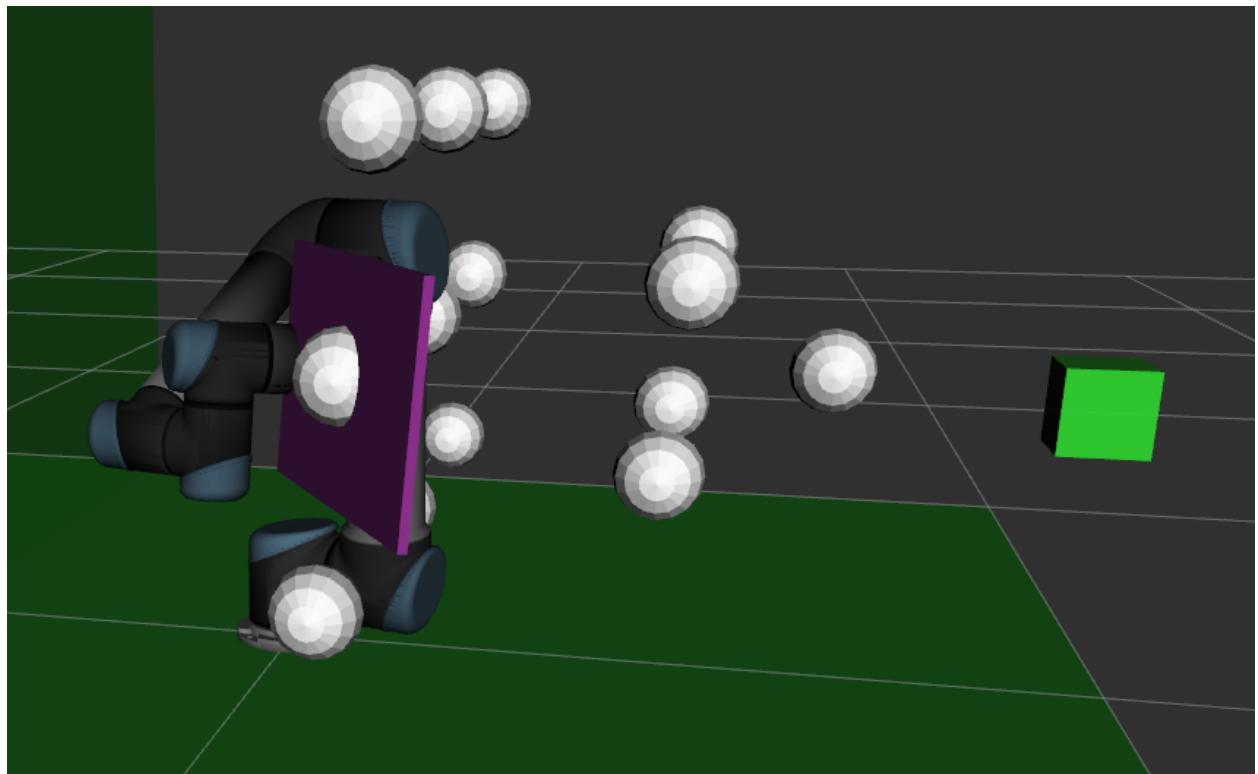


Abbildung 4.8: Die weißen Kugeln sind die verschiedenen Positionen für das Kalibrierungsmuster, die grüne Box rechts ist die Kamera.

4 Implementierung

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 Entfernung abgedeckt wurden 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. Diese Resultate werden wieder in einer `yaml`-Datei abgespeichert. Die Datei wird im Verzeichnis des `calibration_executive`-Pakets abgelegt. Der Inhalt dieser Datei ist in Tabelle 4.3 aufgelistet.

Danach wird das Programm beendet. Siehe auch hier das Ablaufdiagramm in Abbildung 4.9.

4 Implementierung

Parametername	Beschreibung
amount_of_images	Anzahl der Bilder die zur Kalibrierung benutzt wurden
distortion_coefficients / k1, k2, k3	Die Parameter, um die radialsymmetrische Verzeichnung zu berechnen
distortion_coefficients / p1, p2	Die Parameter, um die tangentiale Verzeichnung zu berechnen
error	Der mittlere Fehler in Pixeln
intrinsics / cell_height	Die Höhe eines Pixel auf dem Kamerasensor in Metern
intrinsics / cell_width	Die Breite eines Pixel auf dem Kamerasensor in Metern
intrinsics / focal_length	Die Brennweite in Metern
image_center_x	Der optische Mittelpunkt des Bildes entlang der x-Achse
image_center_y	Der optische Mittelpunkt des Bildes entlang der y-Achse

Tabelle 4.3: Beschreibung der Parameter für den High Level Executive

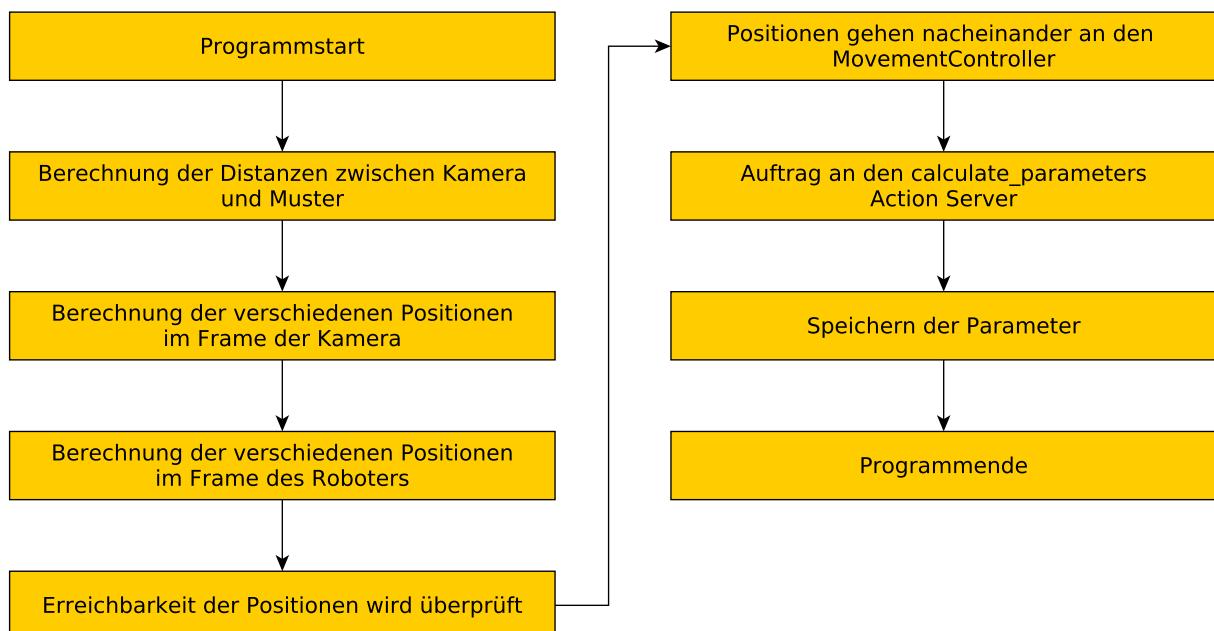


Abbildung 4.9: Ablaufdiagramm für den High Level Executive.

5 Evaluierung

In diesem Kapitel wird die Durchführung einer robotergestützten Kalibrierung beschrieben. Zusätzlich wird diese Durchführung mit einer manuellen Kalibrierung verglichen. Einen Überblick über den Aufbau geben Abbildung 5.1 und Abbildung 5.2.

Die Kalibrierung wurde mit den folgenden Komponenten durchgeführt: Als Kamera wurde die Microsoft LifeCam Studio benutzt. Sie bietet ein Farbbild mit einer Auflösung von 1920x1080 Pixeln an. Der Sensor hat laut Datenblatt eine Höhe von 6,6 mm und eine Breite von 5,85 mm. Die Brennweite beträgt 5 mm. Als Roboter wird der UR5 von Universal Robotics eingesetzt. Als Kalibrierungsmuster wird das Muster aus Abbildung 2.4 benutzt. Es wurde auf DinA4-Größe gedruckt.

Die Kamera steht von der Basis des Roboters aus gesehen an der Position $x=-0,8$, $y=0,5$ und $z=0,35$. Die Kamera ist so ausgerichtet, dass der Mittelpunkt des Bildes ungefähr mittig über der Basis in einer Höhe von 45 cm liegt. Die weiteren Parameter wie die Faktoren zur Distanzbestimmung oder die Verzögerung zwischen dem Erreichen einer Position und der Aufnahme des Bildes werden nicht verändert.

Der UR5-Roboter muss über das angeschlossene Tablet gestartet werden. Zusätzlich muss die IP-Adresse für den Roboter bekannt sein. Auch die Kamera muss über USB mit dem Rechner verbunden sein.

Zunächst werden die Nodes zur Steuerung des Roboters gestartet, beginnend mit dem Hardware Driver. Die IP-Adresse ist selbstverständlich spezifisch für dieses Szenario.

```
$ roslaunch ur_modern_driver >
ur5BringupJointLimited.launch robot_ip:=192.168.102.95
```

Die zweite Node startet MoveIt!.

5 Evaluierung

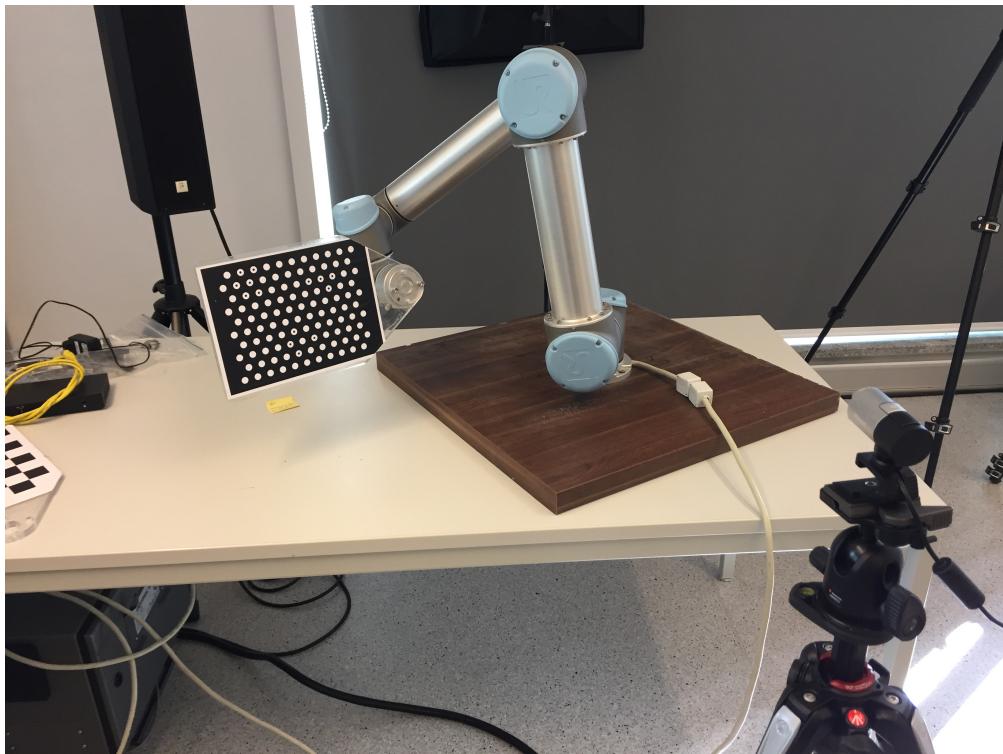


Abbildung 5.1: Der Aufbau zur Kalibrierung aus Sicht der Kamera.

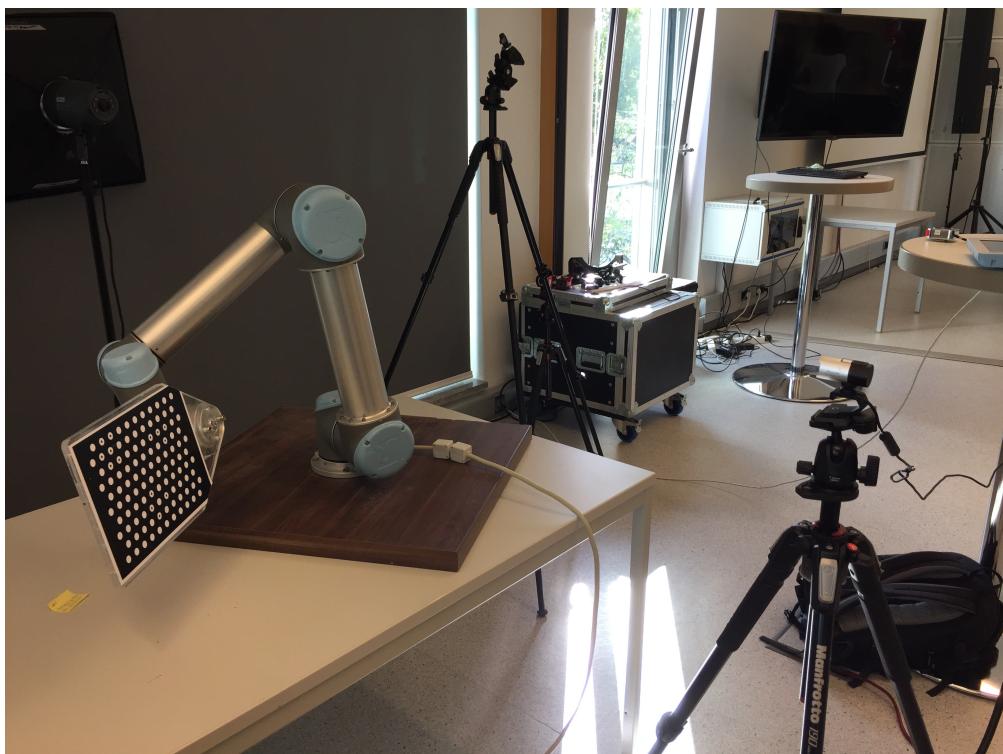


Abbildung 5.2: Der Aufbau zur Kalibrierung von der Seite betrachtet.

5 Evaluierung

```
$ rosrun rosrun ur5_moveit_config >
ur5_moveit_planning_execution.launch limited:=true
```

Die dritte Node startet RViz, um die Planning Scene visuell überprüfen zu können:

```
$ rosrun ur5_moveit_config moveit_rviz.launch config:=true
```

Damit sind die Nodes gestartet, um den Roboter bewegen zu können. Damit MoveIt! über Hindernisse in der Umgebung, wie zum Beispiel der Tisch oder Wände, informiert ist, wird die Node `publish_collision_objects` gestartet. In dieser sind die geometrischen Informationen über die Umgebung gespeichert. Ändert sich die Umgebung, muss auch diese Node angepasst werden.

```
$ rosrun calibration_executive publish_collision_objects.py
```

Um die Bilder der Kamera in ROS zu veröffentlichen, wird ein weiteres launchfile gestartet. Dieses liegt im `calibration_executive` Paket:

```
$ rosrun calibration_executive lifecam_studio.launch
```

Nun sind alle Hardware-spezifischen Nodes gestartet, sodass nun die Programme zur Kalibrierung gestartet werden können. Es wir mit der Node begonnen, die das maschinelle Sehen übernimmt. Hier ist zu beachten, dass in diesem Fall im aktuellen Verzeichnis ein neuer Ordner erstellt wird, in dem die Bilder gespeichert werden. Soll dieser Ordner in einem anderen Verzeichnis erstellt werden, muss dieses Verzeichnis über den Parameter `image_path:=/Pfad/Zum/Verzeichnis` angegeben werden.

```
$ rosrun calibration_perception caltab_detector.launch
```

Mit dem nächsten launchfile wird die client Path Planning Node und der High Level Executive gestartet, damit die Kalibrierung beginnt:

```
$ rosrun calibration_executive high_level_executive.launch
```

Das Bewegen des Roboterarms an die errechneten Positionen und das Auswerten der Bilder dauert ungefähr 10 Minuten. Dabei wurden 54 Bilder aufgenommen. Die Ergebnisse für diese Kalibrierung sind in Tabelle Tabelle 5.1 aufgeführt.

5 Evaluierung

Parametername	Wert
amount_of_images	54
distortion_coefficients / k1	-668.3497992731012
distortion_coefficients / k2	-2558688.8393200147
distortion_coefficients / k3	4026655990788.707
distortion_coefficients / p1	0.021390840982324955
distortion_coefficients / p2	-0.13018888894479966
error	0.21944105843222872
intrinsics / cell_height	3.425925925925926e-06
intrinsics / cell_width	3.4390925635614368e-06
intrinsics / focal_length	0.004996258612286566
image_center_x	965.9302349309
image_center_y	542.5815540282198

Tabelle 5.1: Ergebnisse der Kalibrierung

Da an jeder Position mindestens ein Kalibrierungsmuster gesehen wurde, kann man sich ziemlich sicher sein, dass das ganze Sichtfeld und auch verschiedene Distanzen abgedeckt wurden.

Will man eine ähnlich gute Abdeckung von Hand erreichen, wird die Aufnahme der Fotos ungefähr eine Stunde dauern. Außerdem muss man bei der manuellen Aufnahme der Fotos selber darauf achten, dass das ganze Bild exploriert wurde und ausreichend viele unterschiedliche Orientierungen und Distanzen gewählt wurden. Durch den Einsatz des Roboters hat man also nicht nur einen Zeitvorteil, sondern man kann auch die Qualität der Aufnahmen besser abschätzen.

6 Fazit und Ausblick

In dieser Arbeit konnte gezeigt werden, dass man den zeitaufwändigen Kalibrierungsvorgang für Kameras mit einem Roboter schneller durchführen kann. Die Programme, die den Vorgang durchführen, benötigen dafür einige initiale Parameter und müssen möglicherweise an die Situation angepasst werden. Anschließend kann die Kalibrierung allerdings ohne Benutzerinteraktion durchgeführt werden.

Durch den modularen Aufbau von ROS hat man die Möglichkeit, mehr als einen Kamera-typ zu kalibrieren und man ist ebenfalls nicht an den UR5 Roboter gebunden.

Der Aufwand für den Benutzer sinkt signifikant. Er muss nicht mehr zwischen dem Umstellen der Kamera oder des Kalibrierungsmusters und der Benutzung des Rechners hin und her wechseln. Dadurch sinkt die Zeit zwischen der Aufnahme zweier Bilder stark. Außerdem muss er nicht darauf achten, welche Positionen und Orientierungen er bereits abgearbeitet hat, damit das ganze Bild abgedeckt ist und verschiedene Distanzen und Orientierungen exploriert wurden. Stattdessen muss er lediglich die Programme starten, möglicherweise die Parameter an die Situation anpassen, mit dem Notausschalter in der Nähe die Bewegung des Roboters kontrollieren und zum Schluss überprüfen, ob genügend Positionen und Orientierungen aufgenommen wurden.

Möchte man mehrere Kameras des selben Typs kalibrieren, muss man diese lediglich nacheinander auf dem Stativ befestigen und für jede Kamera das Programm starten. Andere Anpassungen müssen nicht vorgenommen werden.

Um das Programm noch intuitiver und einfacher zu bedienen, sind verschiedene Verbesserungen denkbar.

Im jetzigen Zustand muss der Benutzer auf einige Zentimeter genau ausmessen, wo sich die Kamera befindet und sicherstellen, dass diese exakt auf den Roboter gerichtet ist. Die Position der Kamera in Relation zum Kalibrierungsmuster kann jedoch bereits im ersten

6 Fazit und Ausblick

Bild, in dem das Muster zu sehen ist, berechnet werden. Es wäre also denkbar, dass der Roboter zunächst in eine Grundposition fährt und der Benutzer anschließend so die Kamera vor dem Roboter platziert, dass das Muster gut zu erkennen sein sollte. Anschließend wird das Muster im Bild gesucht, um daraus die Position und Orientierung der Kamera im Raum zu bestimmen. Danach fährt das Programm wie gewohnt fort. Damit muss der Benutzer nicht mehr die Position der Kamera vermessen.

Einige Kamerasysteme bestehen intern aus mehreren Kameras, wie zum Beispiel die Kinect Kameras von Microsoft. Bei diesen Kameras ist nicht nur die Kalibrierung der einzelnen Kameras wichtig, sondern auch die extrinsischen Parameter, also die Position und Orientierung der Kameras zueinander. HALCON ist bereits in der Lage, mehrere Kameras zu kalibrieren. Die Programme müssen allerdings erweitert werden, damit die zusätzlichen Kameras kalibriert werden und auch die extrinsischen Parameter gefunden werden.

Häufig bilden Aktuatoren wie ein Roboterarm und Sensoren wie eine Kamera eine Einheit in einem Roboter. In diesen Fällen ist nicht nur die Kalibrierung der Kamera wichtig, sondern auch die Kalibrierung des Arms. Ein Arm lässt sich dabei auf eine ähnliche Art und Weise kalibrieren: Man befestigt ein Muster am Ende des Arms und anschließend wird die Position des Musters mit einer Kamera bestimmt. Nachdem man also die Kamera kalibriert hat, könnte man diese nun benutzen, um im nächsten Schritt den Arm zu kalibrieren.

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\(20dde139-7e87-4552-8658-dbf2cdaab24b\).html](http://orbit.dtu.dk/en/publications/optimizing-the-universal-robots-ros-driver(20dde139-7e87-4552-8658-dbf2cdaab24b).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
- [Fan09] Fantagu: Verzeichnung. <https://commons.wikimedia.org/wiki/File:Verzeichnung3.png>. Version: 2009. – Abgerufen: 06.08.2017
- [MVT17] MVTEC: *Solution Guide III-C*. <https://www.mvtec.com/fileadmin/Redaktion/mvtec.com/documentation/halcon/halcon-13.0-solution-guide-iii-c-3d-vision.pdf>. Version: 2017. – Abgerufen: 06.08.2017
- [Ope] OpenCV: *Camera calibration With OpenCV*. http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html. – Abgerufen: 06.08.2017
- [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

Literaturverzeichnis

- [WCH92] Weng, J.; Cohen, P.; Herniou, M.: Camera calibration with distortion models and accuracy evaluation. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14 (1992), Oct, Nr. 10, S. 965–980. <http://dx.doi.org/10.1109/34.159901>. – DOI 10.1109/34.159901. – ISSN 0162–8828
- [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

Nachname PrangeMatrikelnr. 2597237Vorname/n Benny

Diese Erklärungen sind in jedes Exemplar der Bachelor- bzw. Masterarbeit mit einzubinden.

Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Alle Stellen, die ich wörtlich oder sinngemäß aus anderen Werken entnommen habe, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Datum

Unterschrift

Erklärung zur Veröffentlichung von Abschlussarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten.

Archiviert werden:

- 1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
- 2) Bachelorarbeiten des jeweils der ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.

Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Datum

Unterschrift