Hochschule Neubrandenburg
University of Applied Sciences

Hochschule Neubrandenburg

Fachbereich Landschaftswissenschaften und Geomatik

Entwicklung einer Sensorfusion basierten, UAV gestützten Navigationslösung in geschlossenen baulichen Strukturen unter Verwendung von 3D-Raummodellen

Masterthesis

zur Erlangung des akademischen Grades
„Master of Engineering“(M.Eng.)

vorgelegt von Dipl. Ing. Olaf Keitsch

Geprüft von:
Prof. Dr. Tobias Hillmann
Prof. Dr. Sven Brämer

Vorgelegt am:
13. Januar 2020

urn:
urn:nbn:de:gbv:519-thesis2019-0075-5

Kurzfassung

Im Zuge der fortschreitenden Digitalisierung und Automatisierung in der maritimen Industrie steigt der Umfang an Mess- und Dokumentationsaufgaben fortlaufend. Im Bereich des Schiffbau erfolgen diese Arbeiten überwiegend in geschlossenen Baustrukturen die durch schlechten Zugang und Erreichbarkeit gekennzeichnet sind. Das steigende Volumen an Daten erfordert einen immer höheren personellen Aufwand mit wenig Potential zur Automatisierung der Datenerfassung. Große Probleme stellt der Zugang von Schiffssektionen in der Gasfreimessung des Arbeitsschutzes sowie der Dokumentation von Teilfertigungen im Bereich von Ballastwasertanks zu Dokumentationszwecken in der Qualitätssicherung dar.
Mit Hilfe eines autonom navigierenden *Unmanned Aerial Vehicle* (UAV) besteht die Möglichkeit, Teil- oder Gesamtaufgaben der Mess- und Dokumentationsprozesse zu automatisieren. Die Bestimmung von Position und Orientierung ist innerhalb geschlossener Stahlbaukonstruktionen mittels *Global Navigation Satellite System* (GNSS) und Kompass nicht möglich. Die vorliegende Arbeit beschäftigt sich mit der Möglichkeit der Bestimmung von Position und Orientierung innerhalb geschlossener baulicher Strukturen auf der Basis bekannter 3D-Modelle. Hierzu wird eine Software konzipiert, die auf einem Onboard-Rechnersystem des UAV auf der Basis von Sensordaten und dem vorhanden 3D-Modell der Umgebung die aktuelle Position und Orientierung des UAV bestimmt. Die Software ist für den autonomen Betrieb des UAV ausgelegt und bedarf keiner Sensoren oder Rechnersysteme, die nicht auf dem UAV implementiert sind. Für die Datenerfassung werden ein Sensor *Guidance* des Herstellers *DJI* sowie ein Laserscanner *sweep V1.0* des Herstellers *Scanse* verwendet. Die Sensordaten werden mit Hilfe der kontinuierlich, rekursiv verbesserten Positions- und Orientierungsdaten vororientiert und mittels *Iterative Closest Point* (ICP) Algorithmus in das bestehende 3D-Modell der Umgebung eingepasst. Auf Basis der Sensordaten ist es möglich, ein vereinfachtes Modell der Helmert Transformation für den Algorithmus zu verwenden. Zusätzlich wird der 3D Modelldatensatz durch ein Occlusion Culling für die Ausgleichung minimiert. Dies führt zu einem erheblichen Performance Gewinn sowie einer sicheren Lösung des Algorithmus. Die Einbindung der Sensorik sowie die Umsetzung aller Softwaremodule erfolgt auf der Basis von Nodes in einer *Robot Operation System* (ROS) Umgebung. Für die Übergabe der Position und Orientierung, alternativ zum verwendeten GNSS-Kompass Modul des UAV, wird ein Weg durch Implementierung eines Schnittellen-Node für den *Controller Area Network* (CAN)-Bus des Flug-Controllers aufgezeigt. Eine Umsetzung erfolgt in dieser Arbeit jedoch nicht.

Abstract

As automation in shipbuilding is in a continuos progress, the range of tasks in measurement and documentation grows. In shipbuilding, measurement and documentation processes need to be done in difficult accessible or closed units of the ship. One of the biggest problem in safety at work is the release of non-toxic rooms after the measurement of gases. A further concern is the documentation of the assembled ballast water tanks for quality assurance.
Work processes are automated further by using an *Unmanned Arial Vehicle* (UAV). The determination of position and orientation by GNSS and compass is not possible in closed units. This master thesis deals with the determination of position and orientation by fitting a measured model into a known 3D model of the planning. For this purpose, a software is designed that determines the position and orientation of the UAS on an onboard computer system using the two 3D point clouds. This software supports an autonomous operation of the UAS (Unmanned Arial Service/System). All sensors and computer systems are located on board of the UAS. Laser scanner *sweep V1.0* from *Scanse* and sensor *Guidance* from *DJI* measures and collects the 3D data. Sensor data are recursively improved and pre-oriented before they get fitted into the 3D planning ship model. Fitting process takes place via the *Iterative Closest Point* (ICP) algorithm. The ICP algorithm uses a simplified model of the Helmert transformation. In addition, the data cloud is minimized through occlusion culling. The result of the algorithm is faster and more secure through these two optimizations. The integration of the sensors and the implementation of all software modules takes place on the basis of nodes in the *Robot Operation System* (ROS) environment.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

ANSI *American National Standards Institute*
AMR *Anisotropic Magneto Resistance*
ASCII *American Standard Code for Information Interchange*
BREP *Boundary Representation*
CAN *Controller Area Network*
CMR *Colossal Magneto Resistance*
DMP *Digital Motion Prozessor*
DTG *Dynamically Tuned Gyroscope*
DXF *Drawing Interchange File Format*
EMR *Extraordinary Megneto Resistance*
FOG *Fibre Optic Gyroscope*
GMR *Giant Magneto Resistance*
GNSS *Global Navigation Satellite System*
GPL *General Public License*
ICAO *International Civil Aviation Organisation*
ICP *Iterative Closest Point*
IGES *Initial Graphics Exchange Specification*
ILS *Instrument Landing System*
IMU *Inertial Measurement Unit*
INS *Inertial Navigation System*
ISA *International Standard Atmosphere*
LIDAR *Light Detection and Ranging*
MEMS *Micro-Electro-Mechanical Systems*
NHN *Normalhöhennull*
RFID *Radio Frequency Identification*
RLG *Ring Laser Gyroscope*
ROS *Robot Operation System*
SCL *STEP Class Library*
SDAI *STEP Data Access Interface*
SDK *Software Development Kit*
SLAM *Simultaneous Localization and Mapping*
STEP *Standard for the Exchange of Product model data*
STL *Stereo Lithography, Standard Tessellation Language*
TMR *Tunneling Magneto Resistance*
USB *Universal Serial Bus*
UAS *Unmanned Aerial System*
UART *Universal Asynchronous Receiver Transmitter*
UAV *Unmanned Aerial Vehicle*
UAW *Ultra Wideband*
VTOL *Vertical Take-off and Landing*
XMR *X-Magneto-Resistive*

1. Einleitung

1.1. Zielstellung

Ziel der vorliegenden Arbeit ist es, Wege für eine Navigationslösung innerhalb geschlossener baulicher Strukturen auf zu zeigen. Im Sinne einer Navigation ist sowohl die Lage, wie auch die Orientierung eines Systems in einem übergeordneten globalen Koordinatensystem zu betrachten. In Folge werden alle Betrachtungen im dreidimensionalen Raum $\mathbb{R}3$ vorgenommen.
Ein UAV beschreibt ein körpereigenes Koordinatensystem. Dieses kann auch als UAV- oder Ausgangskoordinatensystem bezeichnet werden. In der Literatur wird dieses häufig auch als *body frame* bezeichnet. Der Ursprung des UAV-Koordinatensystems kann auf unterschiedliche Weise festgelegt werden. In der Regel wird der geometrische Mittelpunkt oder Masseschwerpunkt des Systems gewählt. Gemäß DIN 9300 [DN90] ist das körpereigene Koordinatensystem als rechtshändiges kartesisches Koordinatensystem wie in Abbildung 1 definiert. Bilden einzelne Sensoren auf dem System raumbezo-

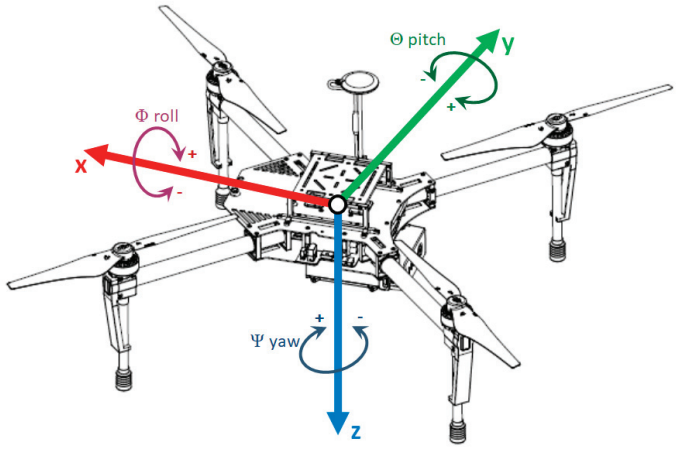

Abbildung 1: UAV-Koordinatensystem

gene Daten in einem eigene Koordinatensysteme ab, so müssen diese zusätzlich in das UAV-Koordinatensystem bezüglich Lage und Orientierung überführt werden.
Das globale oder auch Zielkoordinatensystem wird durch das 3D Modell der gegebenen baulichen Struktur bestimmt. Das Zielkoordinatensystem stellt wiederum ein eigenes System bezüglich Lage und Ausrichtung dar. Dieses muss jedoch nicht zwangsweise mit dem Erdkoordinatensystem übereinstimmen. Eine mögliche Flugroute zur Erfüllung einer automatisierten Flugaufgabe ist durch eine Anzahl aufeinanderfolgender Wegpunkte im Zielkoordinatensystem des 3D Modell definiert. Die Flugroute wird dem UAV für die Navigation zur Verfügung gestellt.
Zur Erfüllung der geplanten Navigationsaufgabe muss das UAV fortlaufend seine Position und Ausrichtung im Zielkoordinatensystem bestimmen um die Wegpunkte der

Route anfliegen zu können. Das UAV ermittelt seine Position und Ausrichtung an Hand der gemessenen Bewegungs- und räumlichen Datenwerte seiner mitgeführten Sensoren. Für die beschriebene Aufgabe sollen geeignete Sensoren und Verfahren zur Bestimmung von Lage und Ausrichtung im Zielkoordinatensystem gefunden werden. Die Prozessierung und Verarbeitung der Sensordaten erfolgt auf dem UAV im Rahmen einer Online Datenverarbeitung.
Die Erarbeitung einer Onboard Navigationslösung stellt die Basis eines automatisierten Fluges des UAV zur Erfüllung einer übergeordneten Aufgabe dar. Dabei stellt die für die übergeordnete Aufgaben benötigte Sensorik den Status einer Nutzlast dar und ist in dieser Arbeit nicht beschrieben und auch nicht Bestandteil der Sensorik der Navigationslösung. Aktuell angestrebte übergeordnete Aufgaben in geschlossenen Räumen sind:

- Messung von Gaskonzentrationen, zum Beispiel in Balastwassertanks oder Sektionen von Schiffen,
- Bilddokumentation von Ausführungsfertigungen oder Zustandsanalysen einzelner Bauteile oder Segmente in Schiffssektionen, Großkesseln oder Gebäuden und
- Infrarot-Messkampanien usw.

Für die derzeit avisierten Einsatzszenarien ist keine Genauigkeit der Position und Orientierung über die der zur Navigation benötigten Genauigkeit zu erwarten. Aus diesem Grund soll sich die angestrebte Genauigkeit der Position und Orientierung im Zielkoordinatensystem an der Navigationsaufgabe ausrichten. Wird im Rahmen einer Messkampagne eine höhere Genauigkeit der Kampagne-Messdaten gefordert, so muss diese auf anderem Wege sicher gestellt werden. Als Zielstellung wird eine Genauigkeit der Lage im Zielkoordinatensystem im Dezimeterbereich sowie der Orientierung mit einem maximalen Fehler von 1° angestrebt.
Für das vorliegende 3D Modell der baulichen Struktur wir nicht von einer vollständigen Umsetzung gegenüber der Realität ausgegangen. Grund hierfür ist die Tatsache, dass CAD-Vorlagen von Bauteilen, wie zum Beispiel im Schiffbau, in der Praxis nicht zu 100% durch konstruiert werden. Diese weisen in der Regel eine Konstruktionsgrad von maximal 90% gegenüber dem realen Bauteil auf. Gleichzeitig kann von baubedingten Abweichungen oder Änderungen gegenüber der Konstruktion während der Fertigung ausgegangen werden. Zusätzlich können sich Objekte im realen Umfeld befinden, die generell nicht Bestandteile der Konstruktion darstellen. Hierfür kommen sowohl Hilfszeuge für die Fertigung oder individuelle Einrichtungsgegenstände in Frage. Somit wird angestrebt, dass die zu entwickelnde Navigationslösung hinreichend robust gegenüber Abweichungen zwischen der realen Umgebung und dem 3D Modell ist.

1.2. Ausgangsbedingungen

Für die praktischen Arbeiten und Tests steht ein UAV des Herstellers *DJI* vom Typ *Matrice 100* zur Verfügung (Abbildung 2). Das UAV ist als Entwicklungsplattform aus-

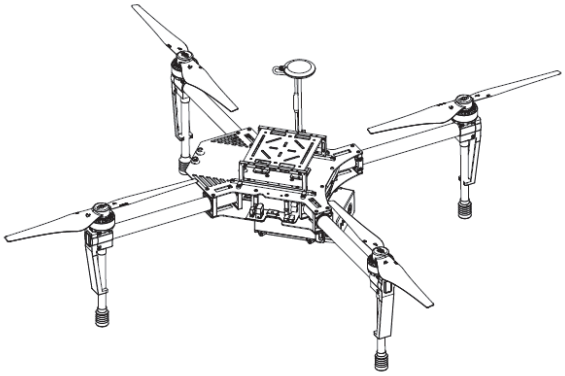

Abbildung 2: Matrice 100 [DJI16]

gelegt und bietet somit die Möglichkeit unproblematisch zusätzliche Sensorik und Komponenten zu montieren. Elektrische und mechanische Anpassungen sind somit leicht möglich.
Zusätzlich wurde ein Multisensorring vom Typ *Guidance* beschafft (Abbildung 3). Das

Abbildung 3: Multisensorsystem Guidance[1]

Sensorsystem dient in erster Linie der Realisierung einer Antikollisionslösung und kann direkt ohne weitere Anpassungen in die Flugsteuerung integriert werden. Für beide Systeme ist ein *Software Development Kit* (SDK) aus dem Hause *DJI* verfügbar. Speziell für den Sensorring *Guidance* besteht somit die Möglichkeit einen kontinuierlichen Datenstrom einzelner Sensoren sowie fusionierte Sensordaten über diverse Standardschnittstellen zu erhalten. Eine Nutzung dieser Sensordaten für andere Aufgaben ist somit möglich.
Als *Light Detection and Ranging* (LIDAR) Sensoren stehen zwei einkanalige Laserscanner die für einen Einsatz auf UAV geeignet sind zur Verfügung. Der Laserscanner

[1] Quelle: `https://www.dji.com/de/guidance?site=brandsite&from=nav`
[2] Quelle: `https://www.hokuyo-aut.jp/search/single.php?serial=167`

Abbildung 4: Scanner *UST-20LX*[2] und *sweep v1.0*[LLC17]

sweep v1.0 des Herstellers *Scanse LLC* stellt hierbei eine low cost Variante dar. Der Laserscanner *UST-20LX* des Herstellers *HOKUYO Automatic Co., LTD* ist im mittelpreisigen Segment für Kleinscanner angesiedelt. Beide Scanner liefern Daten in einem horizontalen Erfassungsbereich von $270 - 360°$ für einen Kanal.
Als embedded System für den on board Einsatz auf dem UAV stehen ein *Raspberry Pi 3* sowie ein Rechnersystem *AscTec Mastermind Board V1.1* der Firma *Ascending Technologies* inklusive WLAN Modul zur Verfügung. Durch die *Neptun Werft* wurde

Abbildung 5: *AscTec Mastermind Board V1.1*[Mak15] und *Raspberry Pi 3 Model B*[3]

die Konstruktionszeichnung einer Sektion des Mock-up Blocks eines Flussfahrtkreuzers im *Stereo Lithography, Standard Tessellation Language* (STL) Format als Datei zur Verfügung gestellt. Zusätzlich wurde im Vorfeld auf der Basis von Konstruktionszeichnungen der *Hochschule Neubrandenburg* das Haus II in Teilen als 3D Modell erstellt. Das Teilmodell liegt als *Autodesk Inventor* CAD Datei vor. Somit stehen ausreichende Modelle für Tests im eigenen Hause sowie im Industrieumfeld zur Verfügung. Die Möglichkeit für praktische Test in der *Neptun Werft* ist gegeben.

1.3. Abgrenzung

Kernaufgabe der Arbeit ist es nicht eine mechanische Trägerplattform als UAV zu entwickeln. Für die Arbeiten steht ein wie in Kapitel 1.2 beschriebenes UAV System zur Verfügung. Die angestrebten Lösungen werden auf dieser Plattform getestet. Es ist jedoch nicht Ziel der Arbeit eine Navigationslösung ausschließlich für dieses *Unmanned*

[3]Quelle: `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`

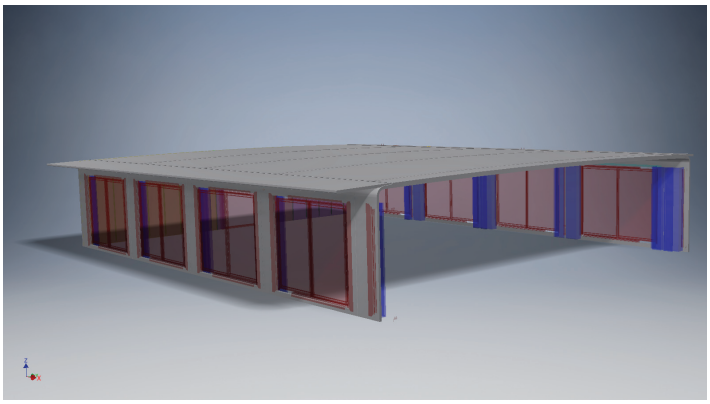

Abbildung 6: Elbe Mock up Sektion

Aerial System (UAS) zu entwickeln. Eine problemlose Portierung auf andere UAS soll gegeben sein.
Fast alle derzeit auf dem Markt verfügbare UAS für den professionellen Einsatz sind für Flüge mit GNSS Nutzung zur Navigation konzipiert. Auf Grund der instabilen Flugmechanik von UAV, speziell Multikopter, ist die Verwendung von Regelsystemen zur Fluglage Stabilisierung derzeit Stand der Technik und ermöglicht dem Anwender ohne übermäßigen Übungsaufwand die Beherrschung notwendiger Flugmanöver und schafft ihm zusätzlich die Möglichkeit, sich auf seine eigentliche Aufgabe wie der Aufnahme von Luftbildern oder ähnliches zu konzentrieren. Einen großen Teil der für die Fluglage Stabilisierung benötigten Messdaten liefert eine integrierte *Inertial Measurement Unit* (IMU). Diese ist jedoch nicht geeignet, wie in Kapitel 3 Abschnitt 3.1 beschrieben, Positions- und Ausrichtungsdaten über einen längeren Zeitraum fehlerfrei zu liefern. Aus diesem Grund werden in allen UAS Informationen zur Positionsbestimmung aus geeigneten GNSS Sensoren und Lageinformationen aus Kompass- und Höhen-Sensoren verarbeitet. Innerhalb geschlossener baulicher Strukturen ist zu fast 100% davon auszugehen, dass durch Abschirmung keine Satelliten zur Erfassung eines verwertbaren GNSS Signales empfangen werden können. Damit liegen in der Regel keine verwertbaren Positionsdaten vor. Innerhalb von Gebäuden, die zu großen Teilen aus Stahlbeton errichtet wurden und vor allem innerhalb von Schiffsstrukturen aus Stahl ist zusätzlich kein verwertbares Kompass Signal verfügbar. Eine Bestimmung der Ausrichtung des UAV um die Hochachse ist somit ebenfalls nicht gegeben. Aus diesen Gründen wird ein Bestimmung der Position und Ausrichtung des UAV in dieser Arbeit grundsätzlich ohne Verwendung von GNSS und Kompass Sensoren durchgeführt.
Die Erfassung und Prozessierung der Sensor Daten erfolgt ausschließlich online und on Board. Unter online ist zu verstehen, dass ein fortlaufender Datenstrom kontinuierlich für eine Zeitnahe Positions- und Richtungsbestimmung verarbeitet wird. On Board bedeutet, dass die erfassten Datenströme auf einem Prozessorsystem verarbeitet werden, der vom UAV System getragen wird. Eine Übertragung der Daten auf eine externe Rechnereinheit, deren Verarbeitung und Rückübertragung ist nicht Bestand-

teil des Konzeptes, jedoch in der Entwicklungs- und Testphase probat. Ebenso werden ausschließlich Sensoren verwendet die sich auf dem UAV System befinden. Eine Bestimmung der Position durch Tracken des UAV mittels Lasertracker, Tachymeter oder bildgebende Verfahren an festen Raumstandorten und daraus resultierende Positions- und Ausrichtungsdaten des UAV sollen nicht Bestandteil einer Navigationslösung darstellen. Grund hierfür ist die Annahme, dass im praktischen Einsatz der Zugang zu befliegender Räume nicht oder nur teilweise gegeben ist. Zusätzlich sind Flüge außerhalb des direkten Sichtfeldes des Pilotenstandortes zulässig.
Die Aufgabenstellung stellt keine Lösung einer klassischen *Simultaneous Localization and Mapping* (SLAM) Problematik dar. Das Mapping ist durch das gegebene Raummodell vorgegeben. Eine Erweiterung, Korrektur oder Verbesserung der Modelldaten auf der Basis der Sensordaten ist nicht geplant. Eine Transformation der UAV bezogenen Ursprungskoordinaten sowie des Ausrichtungsvektors des UAV erfolgt ausschließlich in das Zielsystem des gegebenen 3D Raummodelles.
In der Aufgabenstellung ist die Zulässigkeit eines unscharfen Raummodelles gegenüber der realen Umgebung definiert. Eine Genauigkeit der Positionsbestimmung und Ausrichtung ist zusätzlich nur in dem Rahmen gefordert, der zur Erfüllung der Navigationsaufgabe notwendig ist. Daraus ergibt sich der Umstand, dass die Gefahr der Kollision des UAV mit nicht im 3D Modell vorhandenen Strukturen oder durch ungewollte Annäherung an vorhandenen Strukturen durch zu große Fehler in der Positionsbestimmung jederzeit besteht. Daraus ergeben sich unterschiedliche Problemstellung. Im Rahmen eines autonomen Fluges des UAV muss die Kollision mit Hindernissen, unabhängig davon, ob sie nicht im Modell vorhanden sind oder durch Messfehler in der Positionsbestimmung zu einer Kollisionsgefährdung führen, vermieden werden. Zur Lösung dieses Problemes sind mehrere Lösungsansätze denkbar wie

- Navigation unter Verwendung einer ausschließlich SLAM basierten Navigationslösung,
- Erweiterung und Korrektur des bestehenden 3D Modelles unter Verwendung des fortlaufenden Datenstromes der spatialen Sensorik,
- Implementierung einer Antikollisionslösung mit automatischem Routing von Umgehungswegen für Hindernisse.

Nach Abwägung der Vor- und Nachteile aller genannten Verfahren wurde entschieden, dass eine Antikollisionslösung mit automatisiertem Ausweichrouting als eigenständiges Systemmodul implementiert wird. Diese Antikollisionslösung ist nicht Bestandteil diese Arbeit. Die Einordnung der Antikollisionslösung neben der hier vorgestellten Navigationslösung im Gesamtsystem der UAV Steuerung wird in Kapitel 6 gezeigt.

Teil I.
Grundlagen

2. Unmanned Aerial System - UAS

Als UAV werden Fluggeräte bezeichnet, die unbemannt zum Einsatz kommen. Sie fliegen autonom oder werden durch einen am Boden stationierten Piloten ferngesteuert und überwacht. Wird ausschließlich das Fluggerät beschrieben, so wird die Terminologie UAV verwendet. Wird das Gesamtsystem inklusive Boden gestütztem Equipment zur Steuerung und Überwachung des Fluges sowie der Nutzlast des UAV beschrieben, so wird in diesem Sinn die Terminologie UAS verwendet. Im Rahmen der Klassifizierung von UAS besteht ein breites Spektrum von Kategorien wie die Klassifizierung nach Gewicht, Einsatzzweck, Einsatzhöhe, ziviler oder militärischer Nutzung, aerodynamisches Flugprinzip und viele mehr. Gemäß Klassifizierung durch die *International Civil Aviation Organisation* (ICAO) [ICA11] fallen ferngesteuerte Flugmodelle aus dem Segment des Modellsports nicht in die Kategorie der UAS.
Mit Sicht auf die Bauform oder das aerodynamische Flugprinzip kann man drei grundsätzliche Kategorien von UAV unterscheiden:

1. Multikopter
2. Flächenflieger
3. VTOL

Multikopter stellen UAV dar, die das aerodynamische Flugprinzip durch mehr als einen Hauptrotor umsetzen. In der Praxis haben sich Multikopter mit einer Anzahl von vier bis acht Rotoren durchgesetzt. Da, wie beim Hubschrauber, die Hauptrotoren mit

Abbildung 7: *INSPIRE 2* und *AGRAS MG-1* Multikopter der Firma *DJI*[4]

ihren Rotorblättern durch Rotation alle notwendigen Auftriebs- und Richtungsvektoren für den Flug erzeugen ist ein horizontale Bewegungskomponente für den Erhalt

[4]Quelle: `https://www.dji.com/de`

des Auftriebes nicht notwendig. Somit ist die Möglichkeit senkrechter Starts und Landungen als Grundflugeigenschaft gegeben. Entgegen der Verwendung eines Heckrotor beim Hubschrauber zum Ausgleich des Gegendrehmomentes des Hauptrotor besitzen Multikopter üblicherweise eine gerade Anzahl von Hauptrotoren die zur Gesamtkompensation der Gegendrehmomente der Rotoren paarig gegenläufig arbeiten. Durch die Verwendung mehrerer Rotoren in einer symmetrischen Anordnung wird ein weiterer Vorteil für die Steuerung des Multikopter erreicht. Herkömmliche Hubschrauber steuern ihren Bewegungsvektor durch zyklische Verstellung des Anstellwinkels der Rotorblätter oder Neigung des Hauptrotor in die gewünschte Flugrichtung. Dies bedingt eine aufwendige Mechanik. Multikopter können im Gegensatz dazu direkt durch die Änderung der Drehzahl einzelner Rotoren gesteuert werden. Somit entfällt die mechanische Steuerungskomponente.
UAV als Flächenflieger entsprechen dem klassischen Bild von Flugzeugen. Diese ge-

Abbildung 8: *Intel Sirius*[5] und *DELAIR UX 11* UAV Flächenenflieger[6]

winnen ihren Auftrieb mit horizontaler Bewegung durch die Luft an ihren Tragflächen. Wie bei herkömmlichen Flugzeugen benötigen auch UAV als Flächenflieger eine Mindestfluggeschwindigkeit zur Aufrechterhaltung des Auftriebes welche zu keinem Zeitpunkt des Fluges unterschritten werden darf. Mit Unterschreitung der Mindestfluggeschwindigkeit muss zum Erhalt des notwendigen Auftriebsvektors an den Tragflächen deren Anstellwinkel weiter erhöht werden. Steigt dieser über den kritischen Anstellwinkel an, so kommt es zum Strömungsabriss und in Folge zum Absturz. Aus Sicht der Manövrierfähigkeit sind Flächenflieger damit den Multikoptern unterlegen, besitzen aber hinsichtlich Einsatzdauer, Geschwindigkeit und Nutzlast ein besseres Masse-Leistungs-Verhältnis. Für einen Einsatz in geschlossenen baulichen Strukturen ist die Verwendung von Flächenfliegern als UAV auf Grund ihrer schlechten Manövrierfähigkeit jedoch weniger sinnvoll.
Die Gruppe der VTOL stellen eine Spezialform des Flächenfliegers dar. Sie nutzen den Vorteil der aerodynamischen Auftriebsgewinnung von Tragflächen im Horizontalflug und die Möglichkeit senkrechter Starts und Landung. Triebwerke nach dem Strahl- oder Propeller-Prinzip sind entweder als schwenkbare Komponenten installiert oder es bestehen Vorrichtungen, die den Schubvektor lenken. Hierbei geht es darum, für senk-

[5]Quelle: `https://www.intel.de/content/www/de/de/products/drones/sirius-pro.html`
[6]Quelle: `https://delair.aero/professional-drones/#ux11-anchor`

rechte Starts und Landungen den Schubvektor der Antriebsaggregate in senkrechter Ausrichtung zu verwenden. Der Übergang in einen Horizontalflug erfolgt fließend, wobei der Schubvektor kontinuierlich von der Verwendung als senkrecht wirkender Vektor in einen Vektor zur Vorwärtsbewegung überführt wird. In gleichem Maße, wie hierbei der Schubvektor seinen Anteil als Auftriebsvektor verliert, nimmt der aerodynamische Auftriebsvektor der Tragflächen durch die steigende Vorwärtsgeschwindigkeit zu. Zur

Abbildung 9: VTOL *Wingcopter* des Startup Unternehmens *Wingcopter* aus Darmstadt[7]

Umsetzung dieses Prinzips ist wiederum ein hoher mechanischer Aufwand notwendig. Zusätzlich stellt der Übergang vom Vertikal- zum Horizontalflug und umgekehrt hohe Anforderungen an die Fluglageregelung und ist nicht trivial. Dieser Aufwand lohnt nur, wenn der Vorteil senkrechter Starts und Landungen mit den Masse-Leistungs-Vorteilen der Flächenflieger kombiniert werden müssen. Auch die Gruppe der VTOL ist für den Einsatz in geschlossenen baulichen Umgebungen wenig geeignet. Ihre Vorteile können kaum bis gar nicht ausgenutzt werden. Die weiterhin schlechtere Manövrierfähigkeit gegenüber den Multikoptern wirkt sich eher nachteilig aus.
Der erhebliche Teil auf dem Markt befindlicher UAV nutzt als Energieversorgung Akkumulatoren. Im Segment bis 25 Kilogramm Abfluggewicht setzen fast alle Hersteller auf diese Technologie. Verbrennungsmotoren oder Turbinentriebwerke, wie in der konventionellen Luftfahrt fast ausschließlich in Verwendung, kommen selten bis gar nicht zum Einsatz. Vereinzelt sind erste Brennstoffzellen und Prototypen von UAV mit einer Energieversorgung auf der Basis von Brennstoffzellen auf dem Markt. Eine weitere Zielrichtung stellt den Einsatz von Brennstoffzellen in hybrider Kombination mit herkömmlichen Akkus dar [Ene18b] [Ene18a].
UAV haben ihr alleiniges Image als „Spielzeug“ nachhaltig verlassen. Die Anzahl professioneller Unternehmen, die sich mit der Herstellung von UAV für den kommerziellen Markt in Industrie und Gewerbe befassen, steigt ständig auf der ganzen Welt. Etablierte Anwendungsgebiete stellen die Foto- und Videodokumentation, Überwachung und Inspektion, Messung und Erkundung in der Industrie, Land-und Forstwirtschaft, Vermessungswesen und Kartographie oder der Polizei, dem Katastrophenschutz und

[7] Quelle: `http://www.wingcopter.com/`

Rettungsdienst dar. Der Einsatz im Logistik Sektor scheitert aktuell vorwiegend an flugrechtlichen Hürden.

3. Sensorik

Auf UAV Systemen werden Sensoren zur Bestimmung von Lage, Position und Orientierung eingesetzt, die auch in der klassischen Luftfahrt für diese Aufgaben zu finden sind. Zusätzlich haben hier jedoch Technologien Einzug gehalten die auf bemannten Luftfahrtsystemen keine Verwendung finden.
In den folgenden Kapiteln wird ein kurzer Überblick über geeignete Sensorik zur Bestimmung von Lage, Position, Ausrichtung und Bewegungsdaten von UAV im Indoor Einsatz gegeben. Hierbei werden ausschließlich Sensoren betrachtet, die keinen expliziten Aufbau zusätzliche Infrastruktur im Einsatzbereich erfordern sowie eine Onboard Datenerfassung und Prozessierung ermöglichen. Die folgenden Einführungen in die Sensorik geben einen Überblick über Sensoren der jeweiligen Kategorie, wobei nur auf Messverfahren und Sensoren die auf UAV zum Einsatz kommen genauer eingegangen wird. Im Logistiksektor kommen Positionsbestimmungssysteme zum Einsatz, die auf der Basis lokal installierter Infrastruktur basieren. Beispiele hierfür stellen Umsetzungen auf Basis von *Bluetooth Low Energie* (Bacons), *Ultra Wideband* (UAW) sowie *Radio Frequency Identification* (RFID) dar. Diese System bedürfen jedoch der Installation einer teils engmaschigen lokalen Infrastruktur. Deren Installation lohnt in der Regel jedoch nur für Bereiche mit Aufgabenstellungen für dauerhafte wiederkehrende Ortsbestimmungs- oder Identifikationsaufgaben wie zum Beispiel in der automatisierten Lagerhaltung. Auf Grund des hohen Installationsaufwandes der Infrastruktur werden diese Systeme hier ebenfalls nicht betrachtet, auch wenn diese für den Einsatz auf UAV geeignet erscheinen.
Für den Einsatz auf UAV finden unterschiedlichste Sensoren zur Messung physikalischer Zustandsgrößen Verwendung, die für die Bestimmung von Lage, Position und Bewegungsgrößen im Raum geeignet sind. Tabelle 1 zeigt die gebräuchlichsten Messprinzipien und stellt deren Haupteinsatzgebiete für UAS gegenüber.

3.1. Inertialsensoren

Als Inertialsensoren werden Sensoren bezeichnet, die Trägheitskräfte auf der Basis auftretender Beschleunigungskräfte messen. Unter dem Begriff IMU sind Messsysteme zusammengefasst, die auf der Basis von unterschiedlichen Beschleunigungssensoren translative Bewegungen entlang der Raumachsen sowie Drehraten um die Raumachsen messen. Systeme, die auf der Basis einer IMU Positions-, Lage-, Geschwindigkeits- und Beschleunigungsdaten erfassen werden auch unter dem Begriff *Inertial Navigation System* (INS) geführt.

Messprinzip/Sensor	**Verwendung**
Optische Sensoren (Kamerasysteme, LIDAR)	Hinderniserkennung, Distanzmessung, Bewegungsdetektion mittels optischer Fluss, Mapping
Beschleunigungsmessung	Winkelbeschleunigung für Rotation um Raumachsen, Translative Beschleunigung für Bewegung, Erdbeschleunigung zur Messung der horizontalen Lage im Raum
Druck	barometrische Höhenmessung, Geschwindigkeitsmessung mit Staurohr
Funk/Radar	Hinderniserkennung, Distanzmessung, Messung Höhe über Grund
Ultraschall	Hinderniserkennung, Distanzmessung, Messung Höhe über Grund
Magnetfeld	magnetischer Kurs

Tabelle 1: Sensoreinteilung nach dem Messprinzip

Sensoren zur Messung translativer Beschleunigungswerte werden allgemein als Beschleunigungsmesser oder auch *Accelerometer* bezeichnet. Nach ihrem Funktionsprinzip werden die Beschleunigungsmesser in folgende Typen unterteilt

- Kreisel-Beschleunigungsmesser,
- Pendel-Beschleunigungsmesser,
- Vibrating-Beam-Beschleunigungsmesser,
- Piezo-Beschleunigungsmesser,
- Kapazitive-Beschleunigungsmessung.

Detaillierte Beschreibungen zu den genannten Typen sind in [Law93, 42–70], [Mer96, S. 162–181], [Wen11, S. 66–68] oder [HWLW03, S. 218–221] zu finden. Alle Beschleunigungsmesser beruhen auf dem Prinzip der Masseträgheit einer Probemasse und der Messung derer kinematischen Beschleunigung in Bezug auf einen äußeren Rahmen. In einem gravitativen Umfeld wirkt zusätzlich zur kinematischen Beschleunigung die gravitative Beschleunigung auf die Probemasse. Gemäß Formel 1 [Kni18, S. 52] muss zur Bestimmung der kinematischen Kraft die gravitative Beschleunigung in Abzug gebracht werden.

$$\vec{a}^{\,i} = \vec{f}^{\,i} = \vec{\ddot{r}}^{\,i} - \vec{g}^{\,i} \tag{1}$$

Aus der ermittelten Beschleunigung des inertialen Systems kann durch einfache Integration die Geschwindigkeitsänderung, Formel 3, und durch weitere Integration der

zurück gelegt Weg, Formel 5, bestimmt werden.

$$\int \vec{a}^{\,i}(t)\,dt = \vec{v}^{\,i}(t) + \vec{v}_0^{\,i} \tag{2}$$

$$\Delta\vec{v}^{\,i} = \int_{t_{k-1}}^{t_k} \vec{a}^{\,i}(t)\,dt \tag{3}$$

$$\int \vec{v}^{\,i}(t)\,dt = \vec{r}^{\,i}(t) + \vec{r}_0^{\,i} \tag{4}$$

$$\Delta\vec{r}^{\,i} = \int_{t_{k-1}}^{t_k} \vec{v}^{\,i}(t)\,dt \tag{5}$$

Durch Integration diskret gemessener Beschleunigungswerte können somit nur die Werte der Geschwindigkeit- und Wegänderung bestimmt werden. Durch Aufsummieren der aktuell bestimmten Änderungswerte mit den gegebenen Anfangswerten $\vec{v}_0^{\,i}$ und $\vec{r}_0^{\,i}$ kann der Geschwindigkeitsvektor sowie der zurück gelegte Weg als Streckenvektor seit Beginn der Messung bestimmt werden.
Drehratensensoren messen die Winkelgeschwindigkeit an jeweils einer Achse des inertialen Systems. Zur Messung aller Winkelgeschwindigkeiten der drei orthogonalen Achsen werden somit drei Drehratensensoren mit rechtwinkliger Ausrichtung zueinander benötigt. Drehratensensoren werden nach ihrem Messprinzip unterschieden. Hierbei kommen zwei Messprinzipien zum Einsatz,

1. Messung der Corioliskraft auf mechanische Systeme,
2. Messung auf Basis des *Sagnac Effektes* in optischen Systemen.

Zur Messung der Winkelgeschwindigkeit durch Ausnutzung der Corioloskraft werden

- halbkardanische Kreisel (Kreisel mit einem Freiheitsgrad),
- *Dynamically Tuned Gyroscope* (DTG) mit Ausnutzung des Tuned Effektes,
- kapazitiver Feder-Masse-Sensoren,
- Vibrantionskreisel

eingesetzt. Den *Sagnac Effekt* nutzen Drehratensensoren wie der Ringlaser, *Ring Laser Gyroscope* (RLG), oder der Faserkreisel, *Fibre Optic Gyroscope* (FOG). Detaillierte Beschreibungen zu den genannten Typen sind in [Law93, S. 84–228], [Mer96, S. 186–262], [Wen11, S. 61–66] oder [HWLW03, S. 221–227] zu finden. Aus der gemessenen Drehrate wird mittels Integration der Lagewinkel bestimmt.

$$\int \vec{\omega}^{\,i}(t)\,dt = \varphi^i(t) + \varphi_0^i \tag{6}$$

$$\Delta\varphi^i = \int_{t_{k-1}}^{t_k} \vec{\omega}^{\,i}(t)\,dt \tag{7}$$

Für den Einsatz auf UAV werden ausschließlich *Micro-Electro-Mechanical Systems* (MEMS) zur Beschleunigungsmessung verwendet. MEMS stellen mikro-elektromechanische Bauteile dar, die mikromechanische- und Logikelemente in einem Chip vereinen. In modernen MEMS wird in der Regel auf die Methode der Kapazitiven Messung für Beschleunigungswerte in CMOS-Technologie zurück gegriffen. Für jede Achse integrieren MEMS jeweils eine Sensoreinheit für die translative Beschleunigungsmessung sowie je eine Sensoreinheit für die Drehratenmessung. Für die richtige Achsen-Zuordnung der Messwerte ist eine korrekte Einbaulage im inertialen System erforderlich. Wie in Abbildung 10 dargestellt wird diese für jedes Device durch den Hersteller angegeben. Mit

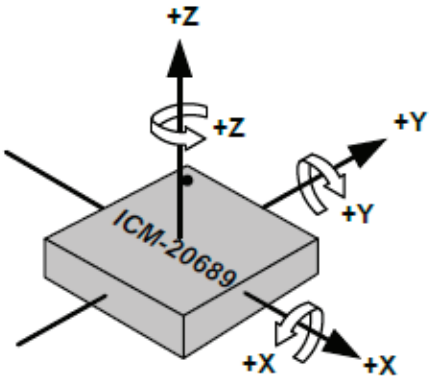

Abbildung 10: Achsenorientierung und Polarität für das Device *ICM-20689* [TDK18, S. 18]

Hilfe der Accelerometer kann wie in Abbildung 11 dargestellt eine absolute Lagebestimmung im Erdschwerefeld vorgenommen werden. Hierbei gilt

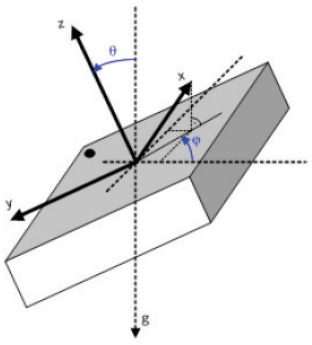

Abbildung 11: Definition der Vektorkomponenten des inertialen Systems [Bos14, S. 37]

$$x_{acc} = 1g * \sin\theta * \cos\psi \tag{8}$$

$$y_{acc} = -\,1g * \sin\theta * \sin\psi \tag{9}$$

$$z_{acc} = 1g * \cos\theta \tag{10}$$

$$\frac{y_{acc}}{x_{acc}} = -\tan\psi \tag{11}$$

Als Beispiel hierfür steht die IMU Einheit des Autopiloten *Pixhawk*. Diese IMU integriert zwei *Motion Tracking Device* aus Gründen der Redundanz. Das Device *ICM-20689* aus dem Hause *TDK* sowie ein Device *BMI055* des Herstellers *Bosch*. Beide Typen integrieren jeweils drei Beschleunigungsmesser sowie drei Gyroskope. Zusätzlich dazu stellt *TDK* im *ICM-20689* noch einen *Digital Motion Prozessor* (DMP) bereit. Abbildung 12 zeigt das Blockdiagramm der Signalverarbeitung und -Bereitstellung der Beschleunigungswerte der sechs Freiheitsgrade durch das Device *ICM-20689*.

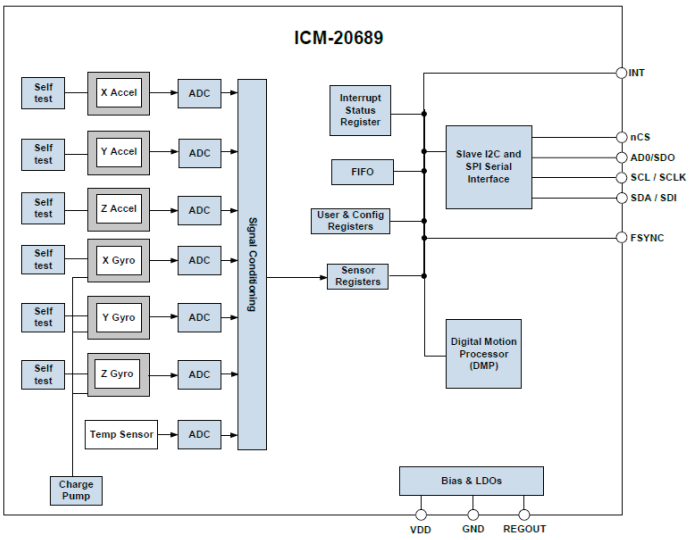

Abbildung 12: Blockdiagramm Device *ICM-20689* [TDK18, S. 20]

3.2. Kompass

Zur Bestimmung der Orientierung eines Systems zur Nordausrichtung im Erdkoordinatensystem werden Kompasssensoren verwendet. Mit ihrer Hilfe ist es möglich eine driftfreie Bestimmung des absoluten Lagewinkels Ψ um die z- und somit Hochachse vorzunehmen. Als Kompass kommen folgende Systeme zum Einsatz

- Kreiselkompass,
- Magnetkompass,
- elektronischer Kompass auf der Basis von Hall-, *X-Magneto-Resistive* (XMR)-Sensoren oder Fluxgate-Magnetometern.

Ein Kreiselkompass ist ein kardarnisch aufgehängter mechanischer Kreisel, dessen Figurachse horizontal ausgerichtet ist. Durch seine gefesselte Aufhängung präzidiert der Kreisel in Meridianrichtung und schwingt somit in die astronomische Nordrichtung ein. Wird ein Kreiselkompass nicht vororientiert, so kann dieser Einlaufvorgang auf Nordrichtung bis zu mehreren Stunden dauern. Zusätzlich stellt sich ein kurs- und geschwindigkeitsabhängiger Fehler ein. Aus diesem Grund werden Kreiselkompasse vor

allem auf langsam fahrenden Schiffen, U-Booten oder für Vermessungsaufgaben im Berg- oder Tunnelbau eingesetzt.

Abbildung 13: Kreiselkompass Standard 22 Compact des Herstellers *Raytheon Anschütz* [Ans18, S. 1]

Ein klassischer Magnetkompass nutzt das Erdmagnetfeld zur Anzeige der magnetischen Nordrichtung mit Hilfe einer Magnetnadel. Diese ist auf einer Spitze frei gelagert. In vielen Geräten wird zur Schwingungsdämpfung das Kompassgehäuse mit einer Flüssigkeit gefüllt. Magnetkompass-Systeme sind sowohl als einfacher Marschkompass wie auch in der Schiff- und Luftfahrt zu finden. Auf Grund der nicht deckungsgleichen Lage des geographischen und magnetischen Nordpols muss beim Magnetkompass eine daraus resultierende Missweisung beachtet werden. Diese wird Deklination genannt. Die Deklination ist kein konstanter Wert für Zeit und Raum. Eine Darstellung der Abweichungen magnetischen von den geographischen Meridianen sowie der zeitlichen Verschiebung der magnetischen Pole sind in Abbildung 14 dargestellt. Wie die De-

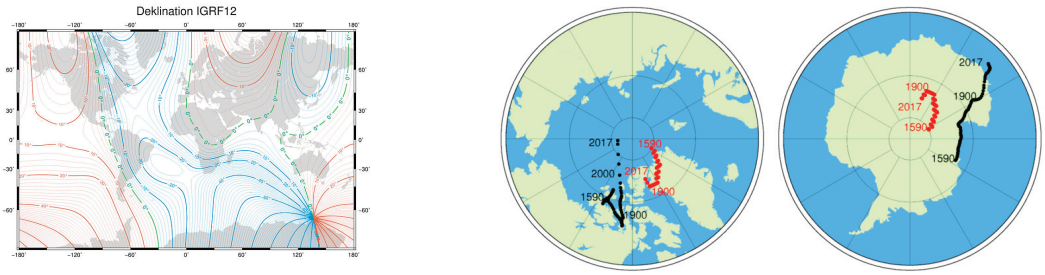

Abbildung 14: Deklinationskarte für 2015 nach IGRF12 und Polkarten der Wanderung der magnetischen Pole[8]

viation beschreibt die Inklination eine Form der Abweichung der Magnetfeldlinien des Erdmagnetfeldes. Die Inklination bezieht sich jedoch auf den vertikalen Winkel der Magnetfeldlinien zur Horizontalen der Erdoberfläche. Auch diese ist wie die Deklination kein konstanter Wert in Raum und Zeit. Eine Inklinationskarte der Erde ist in Abbildung 15 dargestellt. Zusätzlich zur Missweisung durch die Deklination und Inklination

[8]Quelle: `https://www.gfz-potsdam.de/magservice/faq/#c2276`
[9]Quelle: `https://www.ngdc.noaa.gov/geomag/WMM/data/WMM2010/WMM2010_I_MERC.pdf`

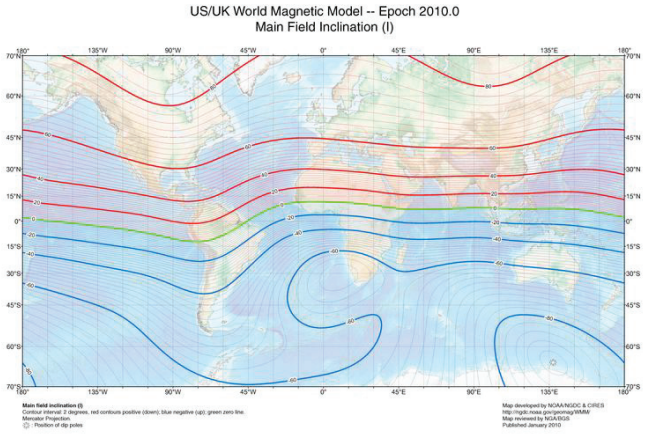

Abbildung 15: Inklinationskarte für das Jahr 2010[9]

ist eine Abweichung durch die Deviation zu beachten. Die Deviation beschreibt die Ablenkung der Magnetfeldlinien des Erdmagnetfeldes durch ferromagnetische Werkstoffe oder magnetische Störfelder im Umfeld des Kompass. Wird die Deviation durch das Trägersystem auf dem der Kompass fest montiert ist verursacht, so wird dieser Fehler in der Regel durch am Kompass befindliche Korrekturmagnete kompensiert. Fehler durch lokale Umgebungsstörfelder oder ferromagnetische Baustoffe wie zum Beispiel Stahl oder Stahlbeton können in der Regel schwer oder nicht kompensiert werden. Zu starke Störfelder oder Abschirmungseffekte durch Stahl im Umfeld können sogar zum vollständigen Ausfall des magnetischen Kompass führen. Elektronische Kompasssysteme auf der Basis von Hall-, XMR-Sensoren oder Fluxgate-Magnetometern unterliegen wie der Magnetkompass ebenfalls den Missweisungserscheinungen der Deklination, Inklination und Deviation.

Fluxgate-Magnetometer basieren auf einer parallelen Anordnung zweier weichmagnetischen Spulenkerne oder einem Ringkern, die durch eine Primärwicklung periodisch in eine magnetische Sättigung versetzt werden. In zwei Empfängerwicklungen kann eine induzierte Spannung gemessen werden. Werden diese gegensinnig geschaltet, so hebt sich diese Spannung gegen Null auf. Wirkt jedoch ein äußeres magnetisch Feld auf den Sensor ein, so wird in den Primärwicklungen in Abhängigkeit von der Einfallsrichtung des Magnetfeldes ein zeitlich unterschiedliches Sättigungsverhalten bewirkt. Dies bewirkt an der Sekundärwicklung eine messbare Wechselspannung. Der Betrag der messbaren Wechselspannung verhält sich proportional zum Einfallswinkel des Messfeldes [Kni18, S. 43]. Mit Hilfe von Fluxgate-Magnetometern können magnetisch Feldstärken von $1nT$ bis $1mT$ gemessen werden. Auf Grund der Verwendung magnetischer Spulenkerne sind der Miniaturisierung von Fluxgate-Magnetometern Grenzen gesetzt. Neben dem Einsatz in der Raumfahrt wie zum Beispiel beim *CHAMP* Satellien, Abbildung 16, zur Messung von weit schwächeren Magnetfeldern als dem Magnetfeld der Erde

werden Fluxgate-Magnetometer in der Industrie sowie als Fluxgate-Kompass in der Schifffahrt verwendet. Zunehmend werden diese jedoch auch in diesen Einsatzberei-

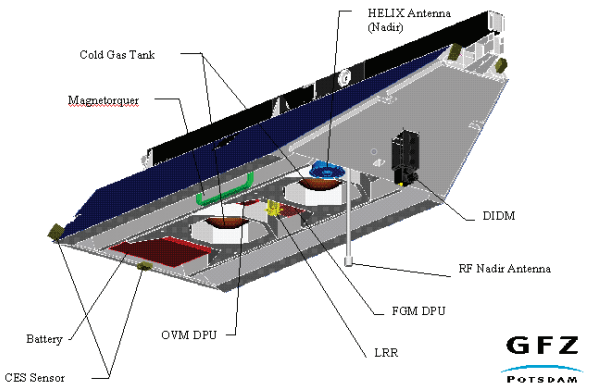

Abbildung 16: *CHAMP*-Satellit mit Fluxgate-Magnetometer[10]

chen durch XMR-Sensoren verdrängt.

Hall-Sensoren nutzen den Hall-Effekt, benannt nach seinem Entdecker Edwin Hall, zur Messung von Magnetfeldern aus. Auf Grund der Tatsache, dass sich Halbleiter

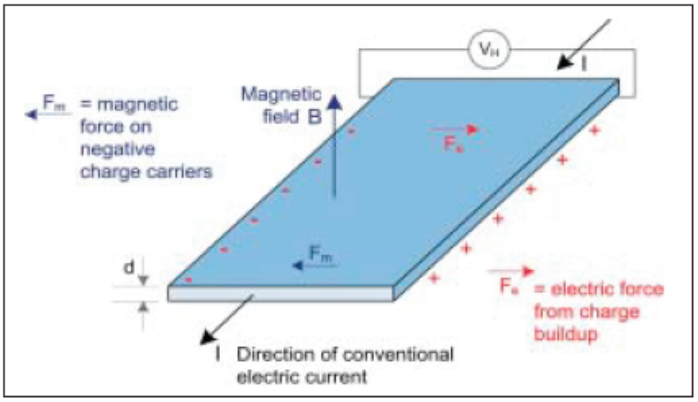

Abbildung 17: Prinzip des Hall-Effekts [Jan06]

sehr gut als Basismaterial für Hallelemente eignen hat sich die Fertigung von Chip basierten Hallschaltkreisen zur Magnetfeldmessung sehr frühzeitig durchgesetzt. Hall-Sensoren setzten sich somit im industriellen Messumfeld sehr schnell und umfassend durch. Mit Hall-Sensoren können Magnetfelder in einem Bereich von $100nT$ bis $10T$ gemessen werden. Eine Messung des Erdmagnetfeldes mit einer Feldstärke von 30 bis $60\mu T$ ist somit gegeben. In den letzten 20 Jahren hat jedoch die Entwicklung von XMR-Sensoren einen deutlichen Schub erfahren. Auf Grund höheren Sensitivität sowie einem stärkeren Potential zur Miniaturisierung haben sich XMR-Sensoren immer weiter durch gesetzt ohne jedoch die Hall-Sensoren verdrängen zu können. Im Bereich der

[10]Quelle: `http://op.gfz-potsdam.de/champ/systems/index_SYSTEMS.html`

MEMS werden für die Messung des Erdmagnetfeldes mittlerweile vorwiegend XMR-Sensoren verwendet [Zwe97, S. 61–62].

Die Gruppe der XMR-Sensoren fasst Sensoren zusammen, die nach unterschiedlichen Verfahren eine Änderung des Widerstandswertes leitender Materialien durch ein Magnetfeld hervorrufen. Zu den magnetoresistiven Verfahren zählen

- *Anisotropic Magneto Resistance* (AMR),
- *Colossal Magneto Resistance* (CMR),
- *Extraordinary Megneto Resistance* (EMR),
- *Giant Magneto Resistance* (GMR),
- *Tunneling Magneto Resistance* (TMR).

Tabelle 2 zeigt eine Übersicht der eingesetzten Verfahren mit einer kurzen Beschreibung ihrer Wirkungsweise. Auch, wenn die magneto resistiven Sensoren nicht die Genauigkeit der Fluxgate-Magnetometer erreichen zeichnen sie sich durch den Vorteil einer sehr preiswerten Produktion und guter Miniaturisierung auf Basis von Halbleiter- und Dünnschichttechnologien aus. Immer stärkere Verbreitung finden AMR-Sensoren in der Schaltkreis basierten Sensortechnologie. Ein Beispiel für den Einsatz von AMR-Sensoren als Kompass stellt die Verwendung des Chips *IST8310* [Ise15] des Herstellers *Isentek Technologies* im IMU-Cube des Autopiloten *Pixhawk4* für UAV dar. Der Sensor ist als dreidimensionaler Magnetsensor ausgelegt. Abbildung 18 zeigt das Blockschaltbild sowie die Orientierung der magnetischen Sensoren des *IST8310*. Die Empfindlichkeit ist mit $3,3\mu T$ und die Auflösung mit $0,3\mu T$ angegeben. Im Gegensatz zur konventionellen Luftfahrt wird der Kompasswert in UAS nicht für den Flug auf einem Kursvektor sonder für die Bestimmung der Ausrichtung des UAV um die Hochachse und deren Lagestabilisierung verwendet.

Verfahren	**Beschreibung**
AMR	Ausnutzung richtungsgebundener elektrischer Widerstandsänderung in Materialstrukturen von Legierungen (anisotropes Verhalten).
CMR	Nutzt die Widerstandsänderung in mikroskopisch dünnen Magnesiuoxidschichtfilmen aus.
EMR	Nutzt die hohe Empfindlichkeit von Halbleitern und elektrisch leitenden Materialien gegenüber kurzen magnetischen Impulsen aus.
GMR	Ausnutzung der Änderung der Leitungsmechanismen auf Basis des Elektronen-Spin-Effektes in metallischen Dünnschichtsystemen.
TMR	Nutzt die ausrichtungsabhängige Widerstandsänderung durch Tunneling von Elektronen durch Isolationsünnschichtstrukturen.

Tabelle 2: Übersicht XMR-Verfahren

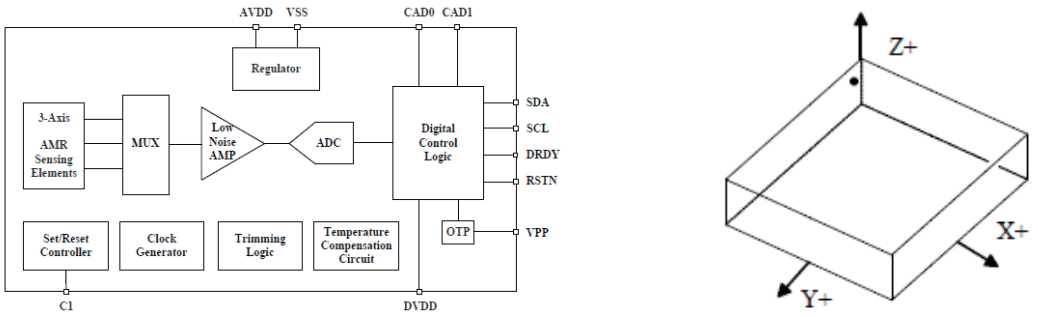

Abbildung 18: Block Diagramm und Orientierung des AMR-Sensor *IST8310* [Ise15, S. 4–5]

3.3. Höhenmesser

In der Luftfahrt werden Höhenmesser sowohl für die Stabilisierung der Flughöhe wie auch zur Kontrolle der Höhe über Grund eingesetzt. Höhenmesser können nach der Messmethode in folgende Kategorien unterteilt werden

- Barometrischer Höhenmesser,
- Höhenmessung auf der Basis von GNSS,
- Höhenmesser nach dem Prinzip der Abstandsmessung.

Höhenmesser nach dem Prinzip der Abstandsmessung basieren auf folgenden Verfahren

- Radar,
- Ultraschall,
- optische Verfahren.

Das älteste Verfahren zur Höhenmessung in der Luftfahrt stellt das barometrische Verfahren dar. Die Berechnung der Höhe richtet sich nach der durch die ICAO definierten Standardatmosphäre *International Standard Atmosphere* (ISA) [ICA93]. Für die aktuellen Betrachtungen ist der Höhenbereich der Troposphäre ausreichend. Diese erstreckt sich bis zu einer Höhe von $11km$. Laut ISA ist der vertikale Temperaturgradient in diesem Bereich konstant und liegt bei $0,0065K/m$ [ICA93, S. E-xii]. Für die Höhe 0 ist die Referenztemperatur T_0 auf $288,15K$ sowie der Referenzdruck P_0 auf $1013,25hPa$ festgelegt [ICA93, S. E-viii]. Mit Hilfe der internationalen Höhenformel 12 für die Troposphäre kann somit die Höhe über Normalnull berechnet werden [Flü12, S. 282].

$$p\left(h\right) = P_0 \cdot \left(1 - \frac{0,0065\frac{K}{m} \cdot h}{T_0}\right)^{5,255} hPa \tag{12}$$

$$h = \frac{P_0}{0,0065\frac{K}{m}} \cdot \left(1 - \left(\frac{p\left(h\right)}{1013,25hPa}\right)^{\frac{1}{5,255}}\right) m \tag{13}$$

Klassische Höhenmesser aus der Luftfahrt basieren in der Regel auf einem Barometer. Die Skala des Anzeigeinstrumentes ist jedoch nicht für eine Anzeige des Druckes, son-

Abbildung 19: barometrischer Höhenmesser *Winter 4FGH10* [Win17]

dern für eine entsprechende Höhe bezogen auf *Normalhöhennull* (NHN) für den Luftdruck P_0 gemäß ISA. Da sowohl die aktuelle Temperatur sowie der aktuelle Luftdruck auf Grund atmosphärischer Schwankungen in der Regel nicht den Standard Werten entsprechen, befindet sich am Anzeigeinstrument ein Drehregler zur Korrektur. In der konventionellen Luftfahrt wird sowohl mit einer Flughöhe nach NHN oder bezogen auf Flugplatzhöhe geflogen.

Für den Einsatz auf UAV ist die Messung einer Flughöhe nach NHN nicht relevant. Auf Grund der geringen Flugzeiten und -Höhen wird nach Höhe über Grund gemessen. Auf der Basis einer Luftdruckmessung wird zur Bestimmung des Referenzdruckes vor Start des UAV eine Referenzmessung durchgeführt. Der somit ermittelte Referenzdruck gilt in Folge als Referenzdruck für die Höhe über Bodenniveau. Mit der Entwicklung von integrierten Luftdruckmessern in Schaltkreistechnologie hat die barometrische Höhenmessung neue Einsatzgebiete erschlossen. Der Drucksensor *MS5611*, verbaut in der IMU-Einheit des Autopiloten *Pixhawk4* für UAV, basiert auf einer piezoresitiven Messzelle. Eine Beschreibung zur Absolutdruckmessung auf Basis von Siliziumsensoren und

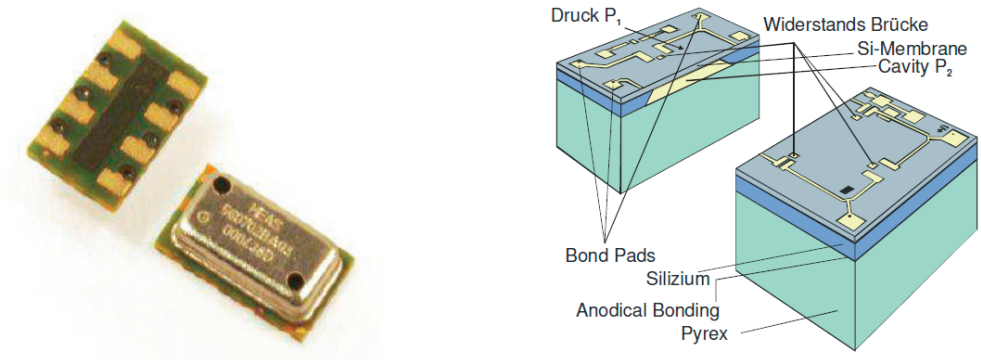

Abbildung 20: Absolutdrucksensor *AMSYS MS5611* [AMS17, S. 1] und Aufbau einer Silizium-Messzelle zur Absolutdruckbestimmung [AMS09, S. 2]

piezoresistiven Widerständen ist in [AMS09] zu finden.

Barometrische Druckmesssensoren werden in der UAV Technik in der Regel ausschließlich für die Stabilisierung der Flughöhe eingesetzt. Dies ist auch ein Grund dafür, warum barometrische Drucksensoren in die IMU-Einheiten der Autopiloten integriert sind. Der Einsatz für die Bestimmung der Höhe über Normal Null zur Einhaltung einer vorgeschriebenen Flughöhe ist hier nicht relevant. Bezug nehmend auf die neuen Bestimmungen zur Einhaltung einer maximalen Höhe von $100m$ über Grund, gemäß LuftVO §21b Absatz 8 [BB17], werden barometrische Höhenmesswerte auch auf UAV verwendet.
Die Bestimmung der Flughöhe für UAV ist vorrangig im Sinne einer Höhe über Grund relevant. Auf Grund der benannten rechtlichen Rahmenbedingungen zur Einhaltung einer Maximalflughöhe wie auch durch die prädestinierten Einsatzszenarien von UAS erfolgen in der Regel Flüge in geringer Höhe. Die Gefahr des unbeabsichtigten Bodenkontaktes ist somit naturgemäß eher gegeben als bei der konventionellen Luftfahrt. Aus diesem Grund besitzen Messsysteme zur Bestimmung der Höhe über Grund eine bedeutend größere Relevanz für UAS. Hierfür bedient man sich sowohl klassischer, wie auch neuer Verfahren, die in der Luftfahrt eher wenig gebräuchlicher sind.
Radarhöhenmesser sind bereits aus der konventionellen Luftfahrt bekannt. Hier finden sie jedoch nur begrenzten Einsatz wie zum Beispiel im Landeanflug, auch in Kombination mit *Instrument Landing System* (ILS), oder als Warnsysteme für das Unterschreitung einer eingestellten Mindestflughöhe über Grund.
Für den Einsatz auf UAS kommen Verfahren zur Höhenbestimmung über Grund zur Anwendung, die in der konventionellen Luftfahrt keine Verwendung finden. Oft sind dies auch Verfahren, die ihr Haupteinsatzfeld in weiter reichenden Anwendungsgebieten für UAS als nur der Höhenmessung finden. Dazu zählen Ultraschall-, LIDAR oder Kamera-Sensoren. In Kapitel 3.5, 3.4 und 3.6 wird auf den Einsatz dieser Sensorarten ausführlicher eingegangen.

3.4. LIDAR Sensoren

LIDAR Sensoren oder Laserscanner erfassen ihre Umgebung nach dem Prinzip der Laserabstandsmessung. Als Messprinzip kommt hierbei die Lichtlaufzeit- oder Phasendifferenzmessung zum Einsatz. Ein Laserimpuls wird in eine definierte Richtung ausgesendet und deren Reflexion durch eine optische Sensoreinheit erfasst.
Für die Messung kommen

- 1D-Laserentfernungsmesser,
- 2D-Laserscanner sowie
- 3D-Laserscanner

zum Einsatz. Unter Verwendung von 1D-Laserentfernungsmessern können nur Objekte in Montagerichtung gemessen werden. Das Erstellen von räumlichen Messbildern

ist somit nur schwer möglich. 2D- und 3D-Laserscanner implementieren mechanische Einrichtungen zur kontinuierlichen schrittweisen Ausrichtung des Lasers. Somit ist es möglich in einer festen Position des Scanners räumlich verteilte Messungen vorzunehmen. Hierzu wird ein rotierender Spiegel vor der Lasereinheit oder die Lasereinheit direkt bewegt. 3D-Laserscanner kombinieren in der Regel beide Prinzipien.
Die durch 2D-Laserscanner erfassten Raumpunkte werden in Polarkoordinaten mit Angabe des Winkels φ und der Entfernung r abgebildet. 3D-Laserscanner bilden die Raumkoordinaten als Kugelkoordinaten mit den Winkeln Θ, φ und der Entfernung r ab. Die gegebenen Polarkoordinaten werden in kartesische Koordinaten für zweidimensionale Messungen mit

$$x = r \cdot \cos(\varphi) \tag{14}$$

$$y = r \cdot \sin(\varphi) \tag{15}$$

sowie für dreidimensionale Messungen mit

$$x = r \cdot \sin(\Theta)\cos(\varphi) \tag{16}$$

$$y = r \cdot \sin(\Theta)\sin(\varphi) \tag{17}$$

$$z = r \cdot \cos(\Theta) \tag{18}$$

umgerechnet.
Zur Minimierung der Datenübertragung werden der minimale Winkel, der maximale Winkel des Scannbereiches sowie die Schrittweite als Konstanten vorgegeben. Als Rohdaten werden in diesem Fall nur noch die Entfernungen r als Array zusammengefasst. Mit Hilfe des Index der Entfernung im Array kann der zugehörige Winkel der Polarkoordinate errechnet werden. Für eine qualitative Bewertung des Messsignales wird zusätzlich ein Intensitätswert in einem weiteren Array zur Verfügung gestellt. Mit Hilfe des Intensitätswertes können Bewertungen zur Qualität des Messsignales und somit eine Differenzierung der Ergebnisse vorgenommen werden.
In der Vermessung zum Einsatz kommende Laserscanner erzielen hierbei Genauigkeiten im Millimeterbereich. Diese sind jedoch für den terrestrischen Einsatz konzipiert. Laserscanner für den Einsatz auf UAV erreichen auf Grund ihrer kleinen Bauform Messgenauigkeiten im Zentimeterbereich.

3.5. Ultraschall Sensoren

Ultraschallsensoren senden hochfrequente Schallwellen aus und messen die Distanz zu Objekten auf Basis der Laufzeit des Schallimpulses. Ultraschallsensoren arbeiten weitestgehend unabhängig von der Materialbeschaffenheit der Reflexionsobjekte. Sie weisen im Medium Luft jedoch eine geringe Reichweite von einigen Metern auf. Im Gegen-

satz zu LIDAR-Sensoren können mit Ultraschallsensoren keine, auf wenige Millimeter, gebündelten Impulse ausgesandt werden. Ultraschallsensoren eignen sich somit vorrangig für die Bereichsüberwachung wie zum Beispiel in der Hinderniserkennung. Für eine Vermessung von Raumkoordinaten sind diese nicht geeignet. Zusätzlich besitzen Ultraschallsensoren eine Blindbereich für kurze Entfernungen und somit eine minimale Erfassungsschwelle für die Entfernung von Objekten. Einen preiswerten Sensor stellt der Ultraschallsensor *HC-SR04* wie in Abb. 21 dar. Die Entfernung zum Objekt wird

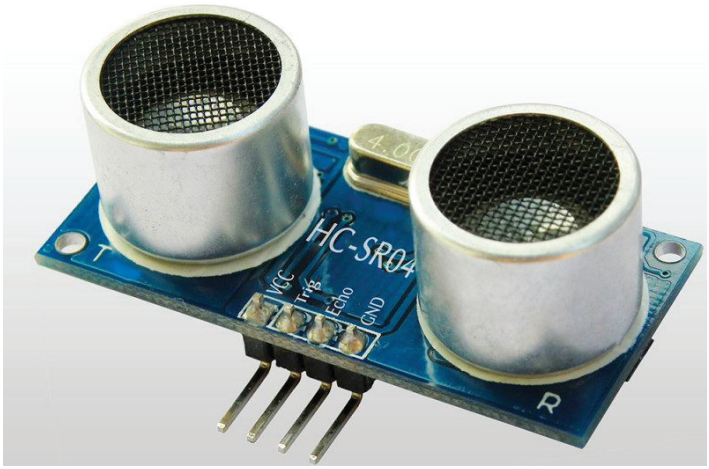

Abbildung 21: Ultraschallsensor *HC-SR04*[11]

auf Basis der Schallgeschwindigkeit im Medium Luft berechnet. Dabei ist zu berücksichtigen, dass die Umgebungstemperatur Einfluss auf die Schallgeschwindigkeit hat. Die Berechnung der Schallgeschwindigkeit kann unter Einbeziehung der Temperatur ϑ in $°C$ mit hinreichender Genauigkeit wie folgt berechnet werden

$$c_{Luft} = (331,5 + 0,6 \cdot \vartheta)\,\frac{m}{s} \tag{19}$$

Womit sich die Entfernung s in Meter für eine Laufzeit t in Sekunden mit

$$s = \frac{c_{Luft} \cdot t}{2} \tag{20}$$

ergibt.

3.6. Kamera Systeme

Mit Hilfe geeigneter Kamerasysteme können 3D Rauminformationen mittels unterschiedlicher Verfahren gewonnen werden.
Stereo-Kameras generieren 3D-Rauminformationen mittels definierter Anordnung zweier Kameras in einer Ebene. Mit der zeitgleichen Aufnahme von zwei Bildern einer Szene

[11] Quelle: `https://www.mikrocontroller-elektronik.de/ultraschallsensor-hc-sr04`

mit definiertem Kameraabstand kann eine Tiefeninformation aus den korrespondierenden Bildpunkten beider Bilder berechnet werden[HLN12, S. 58]. Die Differenz der kor-

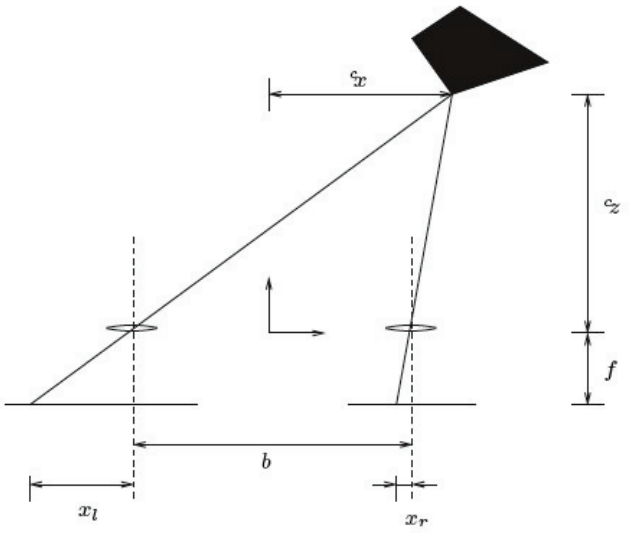

Abbildung 22: Kanonische Stereogeometrie[HLN12, S. 58]

respondierenden Bildpunkte stellt die *Disparität* d dar. Mit Hilfe des Abstandes b sowie der Brennweite f der Kameras werden die Tiefe des Punktes z sowie die Koordinaten x und y berechnet.

$$d = x_l - x_r \tag{21}$$

$$z = \frac{b \cdot f}{d} \tag{22}$$

$$x = \frac{b}{2} \cdot \frac{x_l - x_r}{d} \tag{23}$$

$$y = \frac{b \cdot h}{d} \tag{24}$$

Für die Genauigkeit der Koordinatenberechnung ist sowohl der Kameraabstand b, die Brennweite f sowie die Pixelauflösung der Kameras entscheidend. Mit steigendem Abstand des Objektes zur Kamera steigt der Fehler der Berechnung. Ein großer Kameraabstand sowie eine hohe Auflösung verbessern das Ergebnis. Hier ist jedoch zu beachten, dass Bereiche die nicht durch beide Kameras erfasst werden nicht räumlich abgebildet werden können. Die Reflektionseigenschaften der betrachteten Materialien beeinflussen zusätzlich die Messreichweite.

4. CAD Formate

Mit Sicht auf die verwendeten CAD-Systeme in den relevanten Industriebereichen ist die Frage zu klären, welche Formate durch einzelne Softwareprodukte unterstützt werden und welche Austauschformate für den Anwendungsfall geeignet erscheinen. In erster

Linie gilt es, die gängigen Formate namhafter Softwarelieferanten, wie *Autodesk* mit den Produkten *AutoCAD* und *Inventor* oder *Dassault Systèmes* mit den Produkten *Solid Works* und *CATIA*, zu betrachten. Des weiteren haben sich gängige Austauschformate wie *Initial Graphics Exchange Specification* (IGES), *Standard for the Exchange of Product model data* (STEP) oder STL etabliert.
Für die Aufgabenstellung werden nur 3D-, bei Eignung auch 2,5D-Formate herangezogen. Für die Positionserkennung und Antikollision sind zusätzliche Informationen der physikalischen Eigenschaften der modellierten Körper nicht relevant. Mit Sicht auf weiter zukünftige Aufgaben, wie zum Beispiel die Berücksichtigung von Reflexions- und Absorbtionseigenschaften von Funkwellen der Objekte als physikalische Eigenschaften zur Simulation von Funkausleuchtungen, sollten jedoch Metainformationen über die geometrischen Eigenschaften hinaus Berücksichtigung finden.
Für die Erstellung und Ableitung eines geeigneten Datenmodelles für den Drohneneinsatz ist zu prüfen, welche Datenquellen und Softwareprodukte in den relevanten Einsatzbereichen zur Anwendung kommen. In der Aufgabenstellung wurden sowohl industrielle Unternehmen der Produktion von Großstrukturen wie Werften, Stahlbauunternehmen und deren Kunden wie Reeder, Energieversorger wie auch Errichter und Betreiber von Bauwerksanlagen und Gebäuden identifiziert. Hier ist zu berücksichtigen, welche CAD-Softwareprodukte vorwiegend zum Einsatz kommen. Im Maschinen-, Schiffs-, Flugzeug- und Großstrukturenbau kommen vorwiegend CAD-Softwareprodukte wie:

- *AutoCAD* und *Inventor* der Firma *Autodesk*,
- *SolidWorks* und *CATIA* der Firma *Dassault Systèmes* und
- *Siemens PLM* und *Solid Edge* der Firma *Siemens* (ehemals *Unigraphics*)

zum Einsatz.
Neben den Herstellerspezifischen Dateiformaten haben sich CAD-Austauschformate etabliert. Die gängigsten Austauschformate im 3D-CAD Bereich können in zwei grundlegende Kategorien unterteilt werden:

- Formate für Körpermodelle (auch als Volumenmodelle bezeichent) wie
  - IGES,
  - STEP und
- Formate für Oberflächenmodell wie
  - STL,
  - *Boundary Representation* (BREP) und
  - *Wavefront* (OBJ).

Auf Grund seiner starken Verbreitung hat sich das *DWG-Format* als herstellerspezifisches Dateiformat von *Autodesk* teilweise ebenfalls als ein gebräuchliches Austauschformat durchgesetzt. Dies weniger auf Grund eines etablierten Standards als durch seine frühe Verbreitung. Hersteller anderer CAD-Softwareprodukte bieten aus diesem Grund Importfiltern für das *DWG-Format* an. Aus diesem Grund soll das *DWG-Format* hier mit betrachtet werden.
Die Austauschformate für Oberflächenmodelle IGES und STEP können sowohl als Text- wie auch Binär-Formatdateien vorliegen. Mittlerweile wird für das IGES Format jedoch nur noch das Textformat unterstützt. Dies macht die Dateien sehr groß. Für die Verwendung erscheint das STEP Format derzeit geeigneter.
Das Wavefront Format ist in der Computergrafik weit verbreitet. Dies vorwiegend im Bereich der Entwicklung von Computerspielen. Durch die Implementierung der zusätzlichen Materialdateien ist neben dem Oberflächenmodell die einfache Weitergabe von Metainformationen zu den Geometrieobjekten möglich. Das Format ist einfach lesbar und somit unkompliziert in der Implementierung.

4.1. DWG Format

Das *Drawing* (DWG) stellt ein binäres Dateiformat für die Speicherung von 2D- oder 3D-Zeichnungsdaten dar. Es wurde von der Firma *Autodesk* als eigenes Speicherformat ihrer CAD-Produktfamilie entworfen. Das *Drawing Interchange File Format* (DXF) Format stellt ein *American Standard Code for Information Interchange* (ASCII) basiertes Dateiformat dar und wurde als vorrangiges 2D Austauschformat alternativ zum DWG Format entwickelt. *Autodesk* hat das DWG Format als nicht öffentliches Format entwickelt und im Gegensatz zum DXF Format nicht als Austauschformat vorgesehen. Ungeachtet dessen wurde von diversen Mitbewerbern jedoch das *DWG* Format als Importformat eingebunden. Die *Free Software Foundation* hat beginnend mit dem Projekt *LibreDWG* begonnen das DWG Format zu beschreiben und eine freie Bibliothek unter der GNU *General Public License* (GPL) Lizenz zu veröffentlichen. Eine Beschreibung des Formates ist in [ODA18] veröffentlicht.
Die freie Softwarebibliothek *LibreDWG* stellt eine Weiterführung des *OpenDWG* Projektes dar, ist jedoch seit 2011 aus mehreren Gründen eingestellt. Im Jahre 2013 wurde das Projekt durch *LibDWG* unter der *GNUv3* reaktiviert. *LibDWG* [Cas19] implementiert in seiner aktuell letzten Version 0.6 das Lesen der Formate *R13*, *R14*, *R2000* und *R2004*. Die Bibliothek steht als C-Bibliothek zur Verfügung.

Eine weitere Alternative unter der *GPLv2* und *GPLv3* Lizenz stellt das freie Projekt *libdxfrw* dar. Es können DXF Files gelesen und geschrieben werden. DWG Files sind ab der Version *R14* bis zur Version 2015 implementiert. *libdxfrw* liegt mit Stand 19.10.2015 als C++ Bibliothek vor. Aktuell stellt somit das Paket *libdxfrw* den aktuellsten Stand der Entwicklung freier Softwarebibliotheken für das Lesen von DWG Files dar.

4.2. IGES Format

Das IGES Format ist 1996 als *American National Standards Institute* (ANSI) Standard definiert worden. Obwohl in der Standardisierung das binäre Datenformat vorgesehen ist werden IGES Dateien heute nur noch im ASCII Format gespeichert. Damit besteht zwar die Möglichkeit die Dateien mit einem einfachen ASCII-Editor zu lesen, machte diese aber auch unnötig groß. Das Format definiert geometrische Primitive wie Linien, Flächen und weitere 2D Geometrien. Ab der Version 5 werden auch Körper-Primitive berücksichtigt.

4.3. STEP Format

Das STEP Format ist in der *ISO-Norm 10303* standardisiert worden. Es liefert die Möglichkeit CAD-Austauschdateien im ASCII- und Binär-Format zu speichern. **STEP** bietet neben der Speicherung der reinen Geometriedaten zusätzlich die Möglichkeit Produktdateninformationen über den gesamten Lebenszyklus zu speichern. Auf Grund der frühzeitigen Standardisierung hat sich das Format als weit verbreitetes Austauschformat zwischen CAD-Programmen etabliert. Auf der Homepage des *National Institute of Standards and Technology*[12] können sowohl die Spezifikationen eingesehen, wie auch diverse Links zu Standardbibliotheken gefunden werden[13]. Zusätzlich wird eine C++ Klassenbibliothek *STEP Class Library* (SCL)[14] zur Verfügung gestellt. Die Klassenbibliothek ist als *Open Source Project*[15] verlinkt. Die SCL stellt folgende Komponenten zur Verfügung

- **data access interface library** C++ Klassen für die Implementierung des SDAI Session Schema
- **Data Probe** Eine graphische Anwendung der SCL zur Visualisierung des EXPRESS schema und der Möglichkeit dem Nutzer das Erstellen oder Bearbeiten der enthaltenen Entities zu erlauben.
- **editor library** C++ Klassen zur Implementierung eines Editors für die Entities im EXPRESS schema.
- **exppp** Drucker für die Formatierung von EXPRESS.
- **fedex** Syntax und Schema Prüfer für EXPRESS.
- **fedex_plus** Parst ein EXPRESS Datei und erstellt eine C++ Klassenbibliotheks-Objekt für eine *STEP Data Access Interface* (SDAI) Repräsentanz.

[12] `https://www.nist.gov/`
[13] `https://find.nist.gov/search?utf8=true&affiliate=nist-search&query=STEP&commit=Search`
[14] `https://www.nist.gov/services-resources/software/step-class-library-scl`
[15] `http://stepcode.org/`

- **fedex_idl** Übersetzt ein EXPRESS in ein idl based on the STEP Data Access Interface IDL language binding (ISO-10303-26).
- **fedex_os** Übersetzt ein EXPRESS in C++ Objekte, die zum Hinzufügen von Persistenz zu den Klassen, die von fedex_plus ausgegeben werden, erforderlich sind.
- **mkProbe** Stellt ein EXPRESS File in einem graphischen Browser-Schema dar und ermöglicht es dem Nutzer die Eigenschaften der Entities zu editieren oder neue Entities zu ersellen.
- **NIST EXPRESS Pretty Printer Toolkit** Liest ein EXPRESS File und stellt dieses in einem gut formatierten Schema dar.
- **NIST EXPRESS Toolkit** Liest ein EXPRESS File und bildet es als interne C-Strukturen ab. Es stellt die Basis der fedex Tools dar.
- **probe-ui library** Bibliothek von C++ Klassen, die InterViews-Benutzeroberflächenobjekte zur Implementierung eines Editors von EXPRESS-Instanzen bereitstellen.
- **shtolo** Ein Tool zur Erstellung eines eigenständigen "LangformËXPRESS Schemas aus einem "KurzformSSchema mit Schnittstellenspezifikationen.
- **STEP core library** Bibliothek von C++ Klassen, die STEP-bezogene Funktionalität bereitstellen, einschließlich des frühen und spätgebundenen Attributzugriffs, Lesen und Schreiben von STEP Part 21-Dateien sowie Klassen, die EXPRESS-Basistypen implementieren.
- **utils library** Bibliothek von C++ Klassen, die generische C++ Funktionalität bieten. Ivfasd-Bibliothek - Bibliothek von C++ Klassen, die universelle InterViews-Benutzeroberflächenobjekte bereitstellen.

Eine Implementierung der Klassenbibliothek ist unter den Sprachen C, C++ und Python realisierbar. Das SCL stellt aktuell die stärkste Implementierung des Standards in einer Klassenbibliothek dar.

4.4. STL Format

Das STL Austauschformat wurde von der Firma *3D Systems Inc.* entwickelt und im Jahr 1989 veröffentlicht. Das Format stellt ein reines Oberflächenformat zur Abbildung von Volumenkörpern dar. Es beschränkt sich ausschließlich auf die Darstellung von Oberflächen in einer Dreiecksvermaschung. Die Qualität nicht ebener geometrischer Oberflächen ist von der Dreiecksauflösung ihrer Abbildung abhängig. Das STL Format findet aktuell weite Verbreitung im 3D Druck und hat sich dort als gängiges Austauschformat etabliert.

4.5. Boundary Representation Format

Das BREP Format stellt ein reines Oberflächenformat dar. Damit ist es in seiner Komplexität für CAD-Konstruktionszwecke gegenüber den beschriebenen Körpermodellformaten unterlegen. Jedoch stellt es eine sehr kompakte Datenform für die Darstellung von Körpermodellen in der 3D Grafik dar.

4.6. Wavefront Format

Das Wavefront Format wurde 1989 von der Firma *Wavefront Technologies* entwickelt. In seiner ASCII basierten Dateiform, auch als OBJ Format benannt, wird dieses Format in weiten Bereichen der Computergrafik eingesetzt. Fast alle gängigen 3D Computergrafikprogramme können das Format importieren. Es stellt ein Oberflächenmodel dar, dass durch zusätzliche Materialdateien ergänzt wird.
Das Format stellt folgende geometrische Objekte zur Verfügung

- Punkt,
- Linie,
- Kurve,
- 2D Kurve,
- Fläche und
- Freiformflächen.

Somit ist es nicht, wie das STEP Format, auf eine Dreiecksvermaschung beschränkt. Eine ausführliche Beschreibung des Formates ist unter `http://www.martinreddy.net/gfx/3d/OBJ.spec` zu finden. Starke Verbreitung findet das Format in der Nutzung für Objektmodelle in modernen Game Engines. Durch die Bereitstellung unterschiedlicher Materialdateien, spezifisch zu den beschriebenen Objekten, ist eine Verwendung im Bereich der 3D Grafik mit ausreichend realistischer Abbildung der darzustellenden Objekte möglich. Eine Einbindung der verwendeten 3D Umgebungsmodelle in eine Engine wie *Unity* zur Visualisierung ist somit in guter graphischer Darstellungsform gegeben.
Mit der Auswahl dieses Austauschformates ist somit möglich, eine Datenbasis sowohl für die Einbindung von 3D Modellen im Rahmen dieses Arbeit wie auch einer externen realitätsnahen Visualisierung ohne Konvertierung in andere Datenformate zu verwenden.

Teil II.
Umsetzung

5. Testsystem

5.1. Quadrocopter Matrice 100

Für den vorgesehenen Einsatz auf Drohnen stehen die verbauten Flug-Controller als erste Datenquelle für aktuelle IMU Daten zur Verfügung. In näherer Zukunft werden Tests auf einer verfügbaren Drohne vom Typ *Matrice 100*[16] des Herstellers *DJI* sowie auf einer Eigenbau-Plattform erfolgen. In der *Matrice 100* ist ein Flug-Controller vom Typ *N1* verbaut. Für die Eigenbau-Plattform steht ein Flug-Controller vom Typ *Kakute F4*[17] zur Verfügung.
Für Tests im Rahmen dieser Arbeit wurde die Drohne vom Typ *Matrice 100* verwendet. Zusätzlich wurden als Sensoren das System *Guidance* des Herstellers *DJI* sowie der Laserscanner *Sweep V1.0* des Herstellers *Scanse* verbaut. Als Rechnersystem wurde ein *AscTec Mastermind* des Herstellers *AscTec* mit i7-Prozessor integriert. In Abb. 23

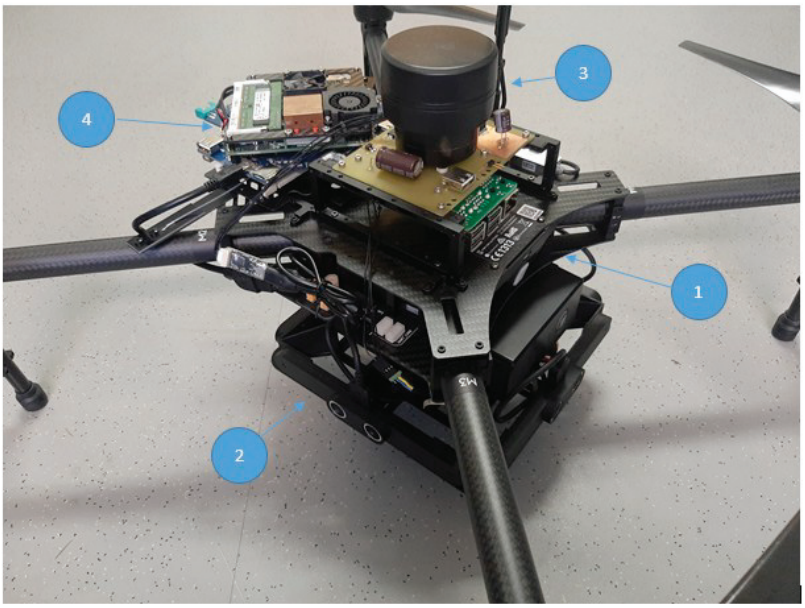

Abbildung 23: Teststem Drohe *Matrice 100* mit Sensorik und Rechnersystem

sind die Komponenten wie folgt angeordnet:

1. Drohne *Matrice 100*,
2. Sensorsystem *Guidance*,

[16] https://www.dji.com/matrice100?site=brandsite&{}from=nav
[17] http://www.holybro.com/product/47

3. Laserscanner *Sweep V1.0* und
4. Rechnersystem *AscTec Mastermind*.

Die *Matrice 100* zeichnet sich durch folgende Merkmale aus[18]:

- physikalische Eigenschaften:
  - Diagonaler Radstand 650 mm
  - Gewicht (mit TB47D Batterie) 2431 g
  - Max. Startgewicht 3400 g
  - Betriebstemperatur $-10°C$ bis $40°C$
- Leistung:
  - Schwebegenauigkeit (P-Modus mit GPS) Vertikal: 0.5m, Horizontal: 2.5m
  - Max. Winkelgeschwindigkeit Steigung: $300°/s$, Yaw: $150°/s$
  - Max. Neigungswinkel 35°
  - Max. Aufstiegsgeschwindigkeit 4 m/s
  - Max. Wind-Widerstand 10 m/s
  - Max. Geschwindigkeit:
    - 22 m/s (ATTI Modus, keine Nutzlast, kein Wind)
    - 17 m/s (GPS Modus, keine Nutzlast, kein Wind)

    item Schwebezeit (mit TB47D Akku) Keine Nutzlast: 28 min; 500g Nutzlast: 20 min; 1kg Nutzlast: 16 min
- Flugkontroller Model N1
- Batterie:
  - Modell TB47D
  - Kapazität 4500 mAh
  - Spannung 22.2 V
  - Typ LiPo 6S
  - Energie 99.9 Wh
  - Nettogewicht 600 g
  - Betriebstemperatur $-10°C$ bis $40°C$

[18] Quelle: `https://www.dji.com/de/matrice100/info#specs`

5.2. Sensorik

5.2.1. Sensorsystem DJI Guidance

Neben den integrierten IMU's der Flug-Controller besteht zusätzlich die Möglichkeit IMU Daten der Sensoreinheit *Guidance* zu nutzen. Die Sensoreinheit wird aktuell für Tests zur Realisierung einer Antikollisionslösung auf der *Matrice 100* verwendet. Abbildung 24 zeigt das System in der Ausbaustufe für fünf Guidance Sensoreinheiten. Jede Sensoreinheit verfügt über einen Ultraschallsensor sowie einer Stereo-Kameraeinheit.

Abbildung 24: Sensorsystem *Guidance*[19]

Neben den fünf Sensoreinheiten ist das System mit einer eigene Prozessoreinheit mit integrierter IMU ausgestattet. Hierfür steht ein SDK des Herstellers zur Verfügung. Über die *Universal Asynchronous Receiver Transmitter* (UART) oder *Universal Serial Bus* (USB) Hardware Schnittstellen des Prozessors kann mittels vorliegendem SDK[20] auf die Datenverarbeitung des Prozessors zugegriffen werden. Das *Guidance* System soll zukünftig auch auf der Eigenbau Plattform getestet werden. Somit steht eine vom Flug-Controller unabhängige IMU zu Testzwecken zur Verfügung. Die ersten Tests zur Aquise von IMU Daten erfolgen auf der Basis des Systems *Guidance*. Das Sensorsystem besitzt folgende Merkmale und Eigenschaften[21]:

- physikalische Eigenschaften:
  - Abmessungen:
    - Guidance Kern: 78,5 x 53,5 x 14 mm
    - Guidance Sensor: 170 x 20 x 16,2 mm
    - VBUS-Kabel: 200mm

[19] Quelle: `https://www.dji.com/de/guidance`
[20] `https://github.com/dji-sdk/Guidance-SDK`
[21] Quelle: `https://www.dji.com/de/guidance/info#specs`

  - Gewicht:
    * Guidance Kern: 64 g
    * Guidance Sensor (Single): 43 g
    * VBUS-Kabel (Single): 11.6 g

- Geschwindigkeitserkennung Reichweite 0 16 m/s (vom Boden 2m) (die Messung ist maßgebend)
- Geschwindigkeit-Erkennungsgenauigkeit 0,04 m/s (vom Boden 2m)
- Positionierungsgenauigkeit 0,05m (über dem Boden 2m)
- Effektive Sensor Reichweite 0,20m 20m
- Externe Anforderungen Gute Beleuchtung; Oberfläche mit viel Textur und klaren Mustern

5.2.2. Laserscanner

Der Scanner *UST-20LX* stellt einen einkanaligen Laserscanner mit einer Reichweite bis maximal 60 Meter dar. Vorteilhaft für seinen Einsatz ist das geringe Gewicht gepaart mit geringen Abmessungen. Der Scanner liefert in einem Frame 1080 Messpunkte. Je-

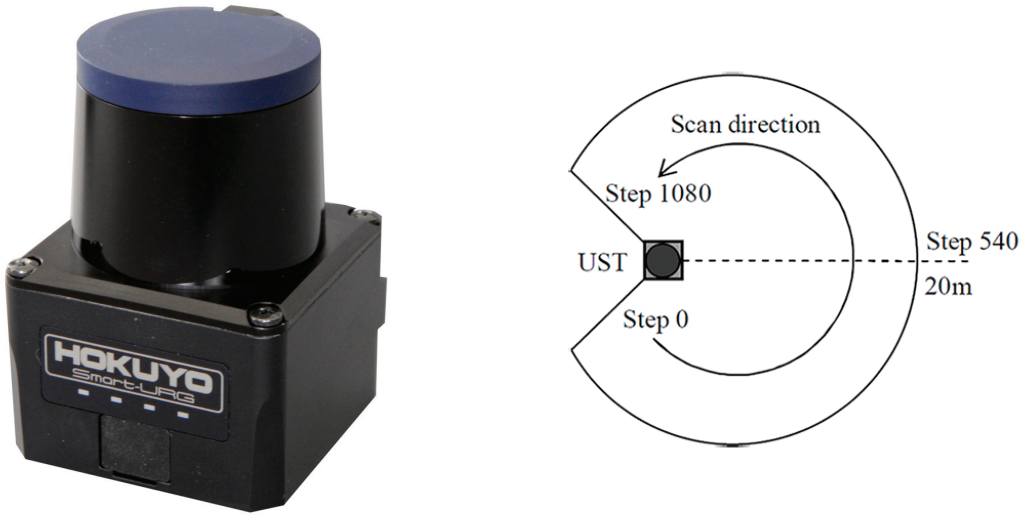

Abbildung 25: Scanner *UST-20LX*[22]

der Messpunkt ist mit den Werten Azimut und Entfernung besetzt. Eine Überführung in kartesische Koordinaten des Scanner Koordinatensystems ist somit problemlos möglich. Der horizontale Erfassungsbereich ist jedoch auf 270° beschränkt. Die Scannzeit für einen Frame beträgt 24ms. Im Folgenden sind die wichtigsten technischen Daten des *UST-20LX* zusammengefasst.

[22]Quelle: `https://www.hokuyo-aut.jp/search/single.php?serial=167`

technische Daten[23]:

- Größe: 50mm x 50mm x 70mm
- Gewicht: 130g
- Stromversorgung: 12/24V DC; 150mA (Start 450mA)
- Genauigkeit: $\pm 40mm$
- Wiederholgenauigkeit: $\sigma < 30mm$
- Scan-Range: 270°
- Scan-Geschwindigkeit: 25ms
- horizontale Winkelauflösung: $0,25°$
- Schutzklasse: IP65
- Interface: Ethernet 100BASE-TX; Open Collector Optokoppler synchroner Datenausgang
- Temperaturbereich: $-10°C$ bis $+50°C$
- Luftfeuchtigkeit: max. 85%
- Laser: 905nm, class 1
- maximale Reichweite: 60m
- nominale Reichweite: 0,06m bis 20m
- Messbereich incl. Reflectance: 0,06 bis 8m

Als Datenschnittstelle steht sowohl ein Ethernet Anschluss wie auch ein synchroner Datenausgang in potentialfreier Ausführung zur Verfügung. Der Scanner wurde in einem TCP/IP Netzwerk in Betrieb genommen. Die Aufzeichnung von Modelldaten erfolgte mit Hilfe der freien Software *URG Viewer*. Abbildung 26 zeigt einen Screenshot der Benutzeroberfläche. Mit Hilfe der Software können die Scanndaten in Dateiform gespeichert werden und stehen somit für eine Offline-Verarbeitung zur Verfügung. Eine direkte Anbindung des Scanners über die zur Verfügung stehenden Schnittstellen auf Softwareebene ist ebenfalls möglich. Hierzu liefert der Hersteller eine entsprechende Protokollbeschreibung . Für die Datenanbindung mittels TCP/IP verfügt der Scanner über eine Ethernet 100BASE-T Schnittstelle sowie einen Socket-Server mit folgenden Default Einstellungen:

[23]Quelle: `https://www.hokuyo-aut.jp/search/single.php?serial=167`
[24]Quelle: `https://sourceforge.net/p/urgnetwork/wiki/urg_viewer_en`

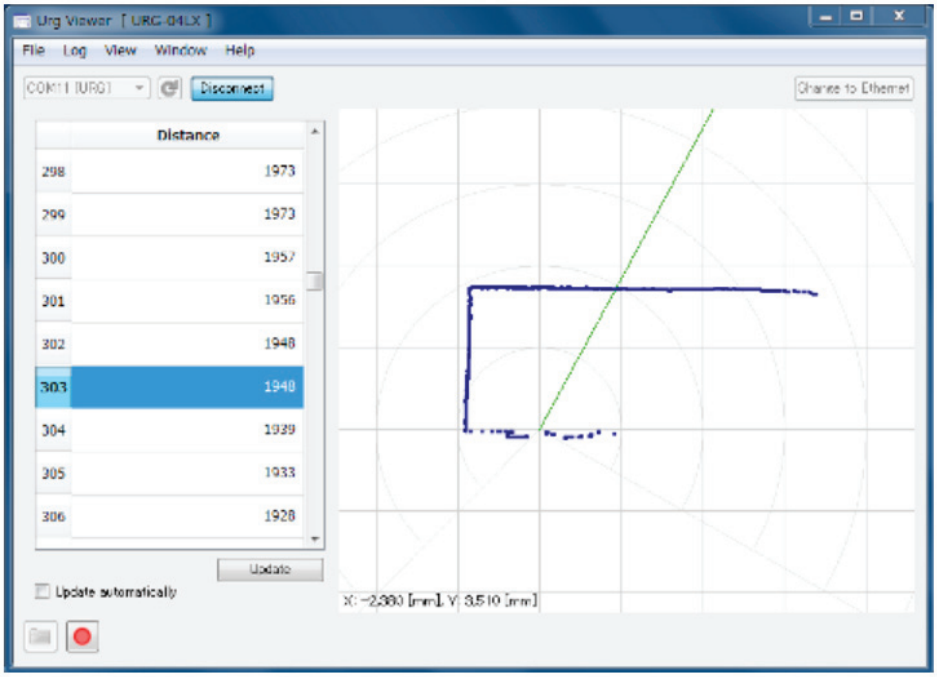

Abbildung 26: Software *URG-Viewer*[24]

IP adress	: 192.168.0.10
Subnet mask	: 255.255.255.0
Default Gateway	: 192.168.0.1
Port number	: 10940 (fixed)

Der Verbindungsaufbau eines Software Socket Clients auf den zur Verfügung gestellten Socket Server ermöglicht dem Anwender den Software seitigen Zugang zu den Scannerdaten.
Für die Kommunikation stehen zwei Kommunikationsmodelle zur Verfügung

1. *response per request*,
2. *multiple responses per request*.

Das erste Modell wird als „Handshake“, das zweite als „Continuous“ bezeichnet. Das Protokoll implementiert folgende Kommunikationskommandos:
Kommandos für die Abfrage von Messwerten

- Distanz Messkommando (GD, MD),
- Distanz und Intensität Messkommando (GE, ME),

Kommandos für die Abfrage von Systeminformationen

- Version Abfrage (VV),
- Abfrage der Parameter der Sensoren (PP),

- Abrage der Statusinformationen der Sensoren (II),

Kommandos zum Ändern des Sensorstatus

- Übergang in den Messzustand (BM),
- Stoppen der kontinuierlichen Messung und Übergang in den Warte Modus (QT),
- Sensor Initialisierung (RS, RT),
- Sensor Neustart (RB) und
- Justieren der internen Uhr (TM).

Der Scanner *sweep v1.0* stellt ebenfalls einen einkanaligen Laserscanner dar. Er verfügt über eine maximale Scannreichweite von 40 Metern. Die Funktionsweise ähnelt

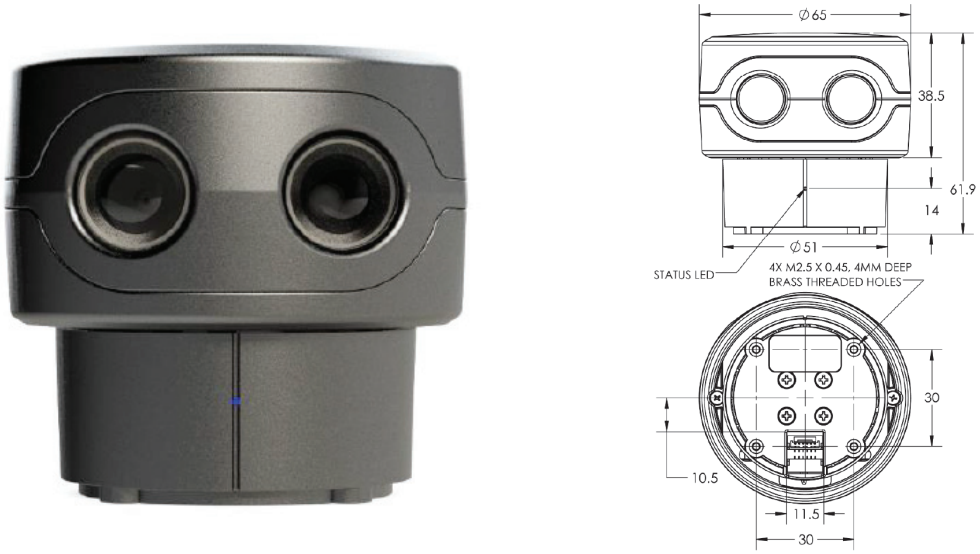

Abbildung 27: Scanner *sweep v1.0*[LLC17]

der des Scanners *UST-20LX*. Abweichend zum *UST-20LX* wird für den Scanner jedoch nur eine Angabe zur Sample Frequenz geliefert. Bei genauer Untersuchung stellt sich heraus, dass die angegebene Sample Frequenz als feste Größe betrachtet werden muss. Die zusätzliche Angabe einer Framerate verändert nicht die Sample Frequenz. Wird ein Frameintervall von 1 Sekunde konfiguriert, so liefert der Scanner 1075 diskrete Messwerte im Zeitintervall von einer Sekunde für einen Vollkreis von 360°. Mit dieser Einstellung ist somit die kleinst mögliche Winkelauflösung des Azimut erreichbar. Die maximal mögliche Framrate von 10Hz liefert somit 10 Frames pro Sekunde mit jeweils $\frac{1075}{10}$ diskreten Datenwerten für einen Vollkreis. Somit sinkt für einen Frame die Winkelauflösung des Azimutes ebenfalls auf $\frac{360°*10}{1075}$. Eine Synchronisation auf feste Winkelwerte für einen Vollkreis erfolgt nicht.
Für den Scanner werden folgende technische Daten durch den Hersteller veröffentlicht.

technische Daten[LLC17]:

- Größe: 65mm x 65mm x 62mm
- Gewicht: 120g
- Stromversorgung: 5V DC; 450mA nominal, 650mA maximal
- Genauigkeit: $10mm$
- Scan-Range: 360°
- Sample Frequenz: max. 1075Hz
- horizontale Winkelauflösung: max. $0,36°$
- Interface: UART
- Temperaturbereich: $-10°C$ bis $+60°C$
- Laser: class 1
- maximale Reichweite: 40m

Dem Lieferumfang des Scanners ist ein UART-USB Adapter beigelegt. Somit ist eine erste Inbetriebnahme problemlos per USB Anbindung möglich. Unter Verwendung der frei beim Hersteller verfügbaren Software *Sweep Visualizer* (Abbildung 28) kann für die erste Inbetriebnahme ein unproblematischer Zugriff auf den Scanner erfolgen. Neben

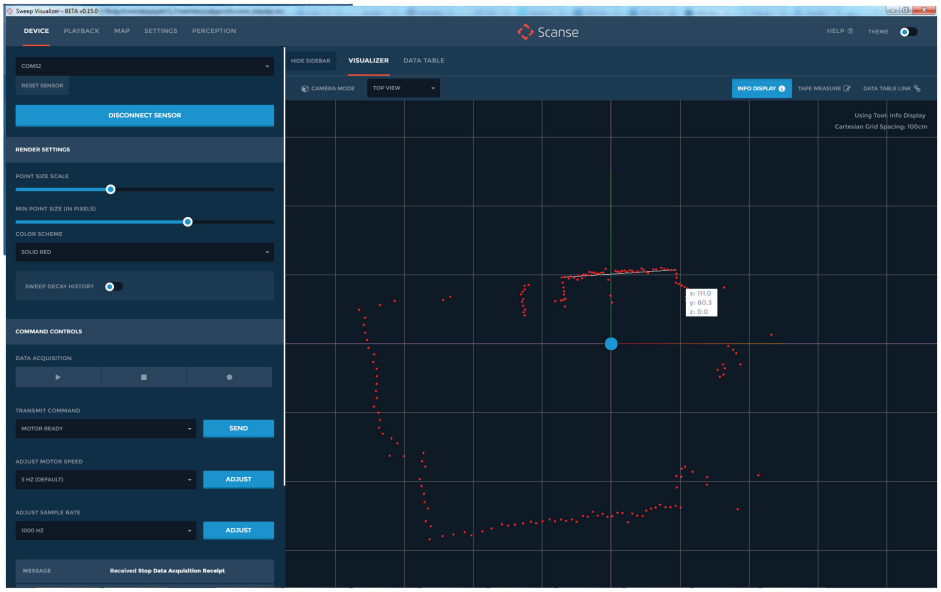

Abbildung 28: Software *Sweep Visualizer*

diversen Einstellungsmöglichkeiten wie der Scann Rate und Sample Frequenz können die eingehenden Daten in Dateiform gespeichert werden. Das Erstellen von Dateien mit

Daten zur Offline Verarbeitung ist somit ebenfalls problemlos möglich.
Eine hardwareseitige Anbindung des Scanner ist ausschließlich über die angegebene UART Schnittstelle möglich. Die Schnittstellenparameter sind wie folgt angegeben:

Bit Rate	: 115.200Baud
Parity	: None
Data Bits	: 8
Stop Bits	: 1
Flow Control	: None

Im Rahmen des durch den Hersteller zur Verfügung gestellten User Manuals liegt eine ausführliche Protokollbeschreibung für den Datenzugriff auf den Scanner für eine eigene Softwareanbindung vor. Ähnlich wie in der Protokollbeschreibung zum Scanner *UST-20LX* stehen sowohl Handshake Kommandos wie auch Protokolle und Kommandos für die kontinuierliche Datenübertragung zur Verfügung. Das Kommunikationsprotokoll implementiert folgende Komminikationskommandos:
Kommandos für die Übertragung von Messwerten

- Start Datenaquise (DS),
- Stopp Datenaquise (DX),

Kommandos für die Abfrage von Systeminformationen

- Motor Ready (MZ),
- Motor Speed Info (MI),
- LiDAR Sample Rate Info (LI),
- Version Info (IV),
- Device Info (ID),

Kommandos zum Ändern des Sensorstatus

- Adjust Motor Speed (MS),
- Adjust LiDAR Sample Rate (LR) und
- Reset Device (RR).

5.3. Embedded System AscTec Mastermind

Für die Onboard Prozessierung der Daten kommt ein Entwicklungsrechner vom Typ *AscTec Mastermind* zum Einsatz. Der Rechner ist mit einem Intel® Core™ i7-3612QE (4 x 2.1 GHz), 4 GB DDR3 RAM ausgestattet und besitzt folgende Schnittstellen:

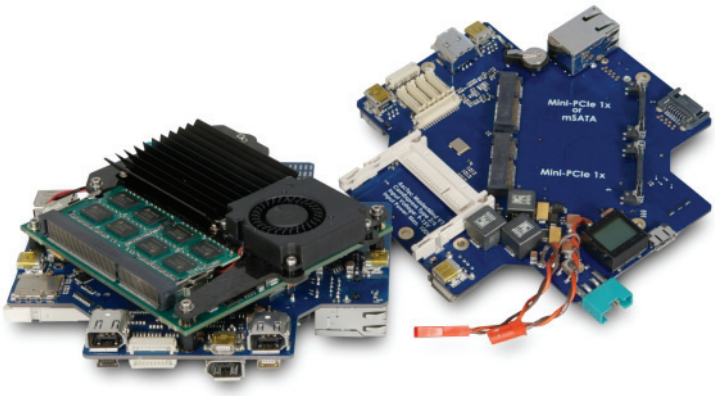

Abbildung 29: Embedded System AscTec Mastermind[25]

- 1 x VGA display port, CRT up to QXGA (2048x1536)
- 5 x USB 2.0; max. current of each USB 2.0 port is 1.0A
- 6 x Serial port
- 1 x MDI-X Ethernet: 10/100/1000 Mbit
- 1 x GPIO
- 3 x FireWire (IEEE 1394): 800 Mbit/s
- 1 x CFast card slot
- 1 x Micro-SD card slot: Micro-SD cards up to 32 GB
- 1 x SerialATA 300 slot (AHCI; RAID 0, 1)
- 1 x Mini-PCI-Express slot
- 1 x mSATA slot or Mini-PCI-Express slot (shared)

Als Betriebssystem ist ein AscTec Mastermind Ubuntu Linux 14.04 installiert. Zusätzlich wird ein vorinstalliertes *ROS Indigo* zur Verfügung gestellt.
Das Board kann in einem Spannungsbereich von 11 bis 15 Volt betrieben werden und hat eine maximale Stromaufnahme von 4,5 Ampere.

6. Lösungsansatz

Trotz der bereits sehr hohen Entwicklungsreife und Verfügbarkeit von UAS für den professionellen zivilen Markt sind diese jedoch ausschließlich für den manuellen oder

[25] Quelle: `http://wiki.asctec.de/display/AR/AscTec+Mastermind`

autonomen Flug im Outdoor Bereich verfügbar. Im Bereich der Navigation für den autonomen Flug beschränken sich die Lösungen auf folgende Strategien

1. Bestimmung der absoluten Position unter Verwendung von GNSS Sensoren,
2. Bestimmung der Ausrichtung um die Hochachse unter Verwendung von Kompass Sensoren,
3. Bestimmung einer relativen Position (Entfernung und Ausrichtung) zu einem Ziel auf der Basis der fortlaufenden Übertragung eines GNSS basierten Positionswertes des Zieles an das UAV,
4. Bestimmung einer relativen Position (Entfernung und Ausrichtung) zu einem Ziel auf der Basis einer optischen Objekterkennung des Zieles über ein geeignetes Kamerasystem mit Tiefeninformation.

Die Lösungsansätze 3 und 4 sind allgemein auch unter dem *Follow Mode* bekannt. Die aufgeführten Lösungen sind jedoch für die Aufgabenstellung dieser Arbeit nicht verwendungsfähig.
Die Sensoreinheit *Guidance* liefert bereits an ihren Datenschnittstellen ein Protokoll für eine gerechnete Position und Orientierung. Leider konnten keine Informationen darüber beschafft werden, aus welchen Sensordaten und mit welchen konkreten Verfahren diese Daten berechnet werden. Es ist jedoch zu vermuten, dass diese Positions- und Orientierungswerte aus den Winkelbeschleunigungswerten der integrierten IMU korrigiert durch Daten der Gyroskop Sensoren sowie der gerechneten Bewegungsdaten mittels optischem Fluss aus den Daten der Kamerasysteme berechnet werden. Praktische Tests haben gezeigt, dass diese Positionsdaten der erwarteten starken Drift durch steigende Zunahme des Fehlers der gerechneten gegenüber der tatsächlichen Position und Orientierung unterliegen. Eine alleinige Nutzung dieser Positions- und Orientierungsinformationen für eine Navigation ist somit nicht zielführend.
Mit Hilfe geeigneter Sensorik auf dem UAV lassen sich spatiale Messdaten gewinnen. Aus diesen Daten kann ein temporäres Abbild des umgebenden Raumes erstellt werden. Dieses temporäre Raumabbild ist im Ausgangskoordinatensystem des UAV orientiert. Auf dem Rechner des UAV ist ein hinreichend genaues und vollständiges spatiales Modell der Umgebung (3D Modell) gespeichert. Kann dieses temporäre Raumabbild eindeutig in das Zielsystem transformiert werden, so bilden die transformierten Ursprungskoordinaten des Ausgangssystems die aktuelle Position des UAV im Zielsystem. Durch die Transformation wird zusätzlich die Lage des Ausgangskoordinatensystems zum Zielkoordinatensystem in Einklang gebracht. Somit wird auch die Orientierung des UAV im Zielsystem bestimmt. Der translative Anteil der Transformation bestimmt die Position und der Rotationsanteil der Transformation die Orientierung im Zielkoordinatensystem. Mathematisch kann dieser Vorgang als Transformation im R3 Raum

beschrieben werden [Nie01, S. 313–315].

$$\begin{bmatrix} x_g \\ y_g \\ z_g \end{bmatrix} = \mathbf{T} + m \cdot \mathbf{R}\left(\Phi, \Theta, \Psi\right) \cdot \begin{bmatrix} x_f \\ y_f \\ z_f \end{bmatrix} \tag{25}$$

Hierbei stellt der Vektor $[x_f, y_f, z_f]^T$ die Koordinaten des Ursprungs des Ausgangskoordinatensystems des UAV aus Abbildung 1 dar. Der Vektor $[x_g, y_g, z_g]^T$ als Ergebnis der Transformation entspricht der berechneten Position des UAV im Zielkoordinatensystem des 3D Modell. Die Matrix T bildet die Translationsmatrix $T = [t_x, t_y, t_z]^T$ der Transformation ab. Der Skalar m bestimmt den Skalierungsfaktor. Die Rotationsmatrix $R\left(\Phi, \Theta, \Psi\right)$ der Transformation beschreibt die Rotation um alle drei Achsen. Die dreiachsige Rotationsmatrix kann in die Teilkomponenten ihrer Einzelrotationen zerlegt werden.

$$R\left(\Phi, \Theta, \Psi\right) = R\left(\Phi\right) \cdot R\left(\Theta\right) \cdot R\left(\Psi\right) \tag{26}$$

$$R\left(\Phi\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\Phi & \sin\Phi \\ 0 & -\sin\Phi & \cos\Phi \end{bmatrix} \tag{27}$$

$$R\left(\Theta\right) = \begin{bmatrix} \cos\Theta & 0 & -\sin\Theta \\ 0 & 1 & 0 \\ \sin\Theta & 0 & \cos\Theta \end{bmatrix} \tag{28}$$

$$R\left(\Psi\right) = \begin{bmatrix} \cos\Psi & \sin\Psi & 0 \\ -\sin\Psi & \cos\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{29}$$

Die Eulerwinkel Φ, Θ, Ψ beschreiben die Rotationswinkel um die x-, y- und z-Achse [BAL11, S. 49]. Mit Hilfe der Rotationsmatrix kann der Richtungsvektor des Ursprungskoordinatensystem des UAV $\vec{o_f}$, in der Regel der nach vorne gerichtete Vektor und somit der X-Achsenvektor, in die korrekte Lage im Zielkoordinatensystem überführt und somit der Orientierungsvektor $\vec{o_g}$ des UAV im Zielsystem berechnet werden.

$$\vec{o_g} = R \cdot \vec{o_f} \tag{30}$$

Der in Kapitel 5.2.1 beschriebene Sensorring *Guidance* stellt an seinen Schnittstellen Daten der implementierten Stereo Kameras zur Verfügung. Über das vorhandene SDK besteht gleichfalls die Möglichkeit auf diese Daten zuzugreifen. Als Kameradaten werden sowohl Graustufenbilder wie auch Tiefenbilder zur Verfügung gestellt. Die

gelieferten Tiefenbilder kodieren in den Grauwertdaten des Pixels den zugehörigen Tiefenwert an dieser Bildposition. Somit besteht die Möglichkeit, unter Verwendung der Kameraparameter, diese Bildinformationen in spatiale Datenwerte im Koordinatensystem des UAV zu überführen. Durch den geringen Abstand der Kameras der Stereo Kamera Paare ist die Tiefenreichweite jedoch stark begrenzt. In der Praxis ist mit einem Einsatz auch in größeren Raumstrukturen zu rechnen. Aus diesem Grund wird zusätzlich ein LIDAR Messsystem wie in Kapitel 5.2.2 beschrieben eingesetzt. Dieses liefert spatiale Daten für größere Reichweiten. Jedoch in einer ungünstigen spatialen Verteilung auf Grund der Auslegung als einkanaliger Scanner. Mit der Fusion der Daten beider Systeme kann jedoch eine Verbesserung der spatialen Basisdatenmenge erreicht werden. Zusätzlich liefern die Scanner zuverlässige Daten auch bei schlechten Beleuchtungsverhältnissen. Diese fusionierten Daten stellen als Ergebnis das geforderte temporäre Raumabbild als spatiale Datenmenge im Ursprungskoordinatensystem dar. Mit Hilfe des temporären Raumabbildes sowie dem vorhandenen 3D Modell liegen jetzt alle Datenvoraussetzungen für eine Bestimmung der Parameter einer Transformation in das Zielkoordinatensystem vor.

Für eine Lösung ergeben sich jedoch unterschiedliche Probleme. Die Datenmenge des temporären Raumabbildes ist in der Regel wesentlich geringer als die des vorliegenden 3D Modelles mit einer spartial engen Verteilung. Die Wahrscheinlichkeit das keine eindeutige oder auch generell keine Lösung für die notwendigen Transformationsparameter berechnet werden können ist somit sehr hoch. In erster Linie müssen zur Verbesserung der Lösungswahrscheinlichkeit des Problems näherungsweise Schätzparameter für eine Vororientierung ermittelt werden. Eine fortlaufende Positionsbestimmung stellt einen Prozess einer zeitlich diskreten Abfolge aneinandergereihter Einzel-Positionsbestimmungen dar. Liegt für hinreichend kurze Zeitintervalle eine Parametersatz der letzten Transformation vor, kann dieser für eine Vororientierung verwendet werden. Wird bei Start der Prozesskette bereits eine näherungsweise Position und Orientierung vorgegeben können auch die Start-Transformationsparameter für das System ermittelt werden. Nach dem Start up des Systems wird als Grundbedingung eine gültige Position und Ausrichtung des UAV im Zielsystem in grober Näherung als erste gültige Initialisierung vorgegeben. Praktisch bedeutet dies, dass dem System nach dem Einschalten die aktuelle Standortposition und Ausrichtung als Initialisierungswerte übertragen werden oder das UAV auf dem ersten Wegpunkt mit korrekter Ausrichtung zum nächsten Wegpunkt positioniert wird. Somit kann der erste Wegpunkt der gespeicherten Route sowie der erste Richtungsvektor der Route als Startparameter verwendet werden. Dies muss jedoch mit einer praktikablen zulässigen Abweichung möglich sein. Die Werte der Initialisierungsparameter können somit einer groben Näherung der aktuellen Position und Ausrichtung entsprechen.

Die aktuell vorliegenden Raummodelle zeigen, dass für einen zukünftigen Einsatz mit eher großen Modelldatenmenge des Zielsystems zu rechnen ist. Für eine Datenverar-

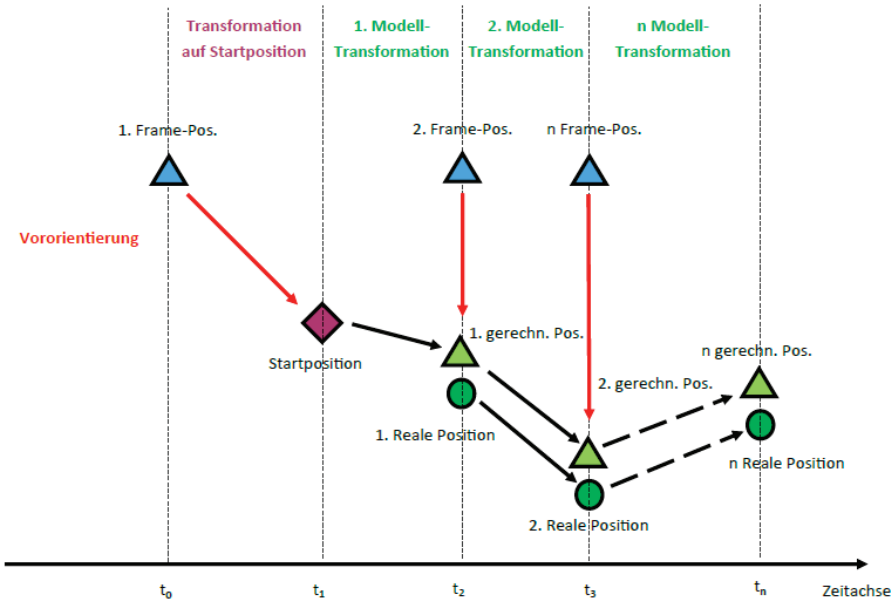

Abbildung 30: Prozesskette der zeitdiskreten Transformation zur Bestimmung der Position und Orientierung

beitung auf UAV tragfähigen Rechnersystemen als embedded Systeme ist von einer begrenzten Leistungsfähigkeit bezüglich Speicher und Prozessorleistung auszugehen. Unabhängig vom gewählten Verfahrens zur Ermittlung der Transformationsparameter ist eine Reduktion der Modelldatenmenge förderlich. Zur Vermeidung mehrdeutiger Lösungen teilweise sogar erforderlich. Bei Einhaltung der gezeigten Prozesskette zur Bestimmung von Position und Orientierung liegen zu jedem Zeitpunkt der Datenverarbeitung gültige Schätzwerte einer näherungsweisen Position vor. Somit ist es sinnvoll und möglich die Datenmenge der Modelldaten des Zielsystems zu minimieren. Aus dem Bereich der Comuptergrafik sind unterschiedliche Verfahren zum Thema Sichtbarkeit als Maßnahme der Reduktion von Prozessdaten bekannt. Aus den Ausführungen zur Erstellung des temporären spatialen Raumabbildes aus Sensordaten ist zu erkennen, dass alle Datenwerte dem Kriterium der direkten Sichtbarkeit unterliegen. Somit besitzen auch nur die Modelldaten des Zielsystems zur Berechnung Relevanz, die vom aktuellen Standort sichtbar sind. Modelldaten außerhalb des Sichtbarkeitsbereiches der Sensorik können somit temporär für den aktuellen Prozessschritt aus dem Modell entfernt werden. Das *Occlusion Culling* aus dem Bereich der Grafikprogrammierung stellt hierbei ein mögliches Verfahren zur Minimierung der Prozessdatenmenge dar [NFHS11, S. 451–454]. Hierbei werden alle Objekte des Zielmodelles entfernt, die nicht im *View Frustum* liegen oder durch Modellobjekte in der direkten Sicht verdeckt werden. Der *View Frustum* stellt den Sichtkegel der Kamera in der Computergrafik dar [NFHS11, S. 448–450]. Im Vorliegenden Fall ist der *View Frustum* der Sichtkegel der Sensorik des UAV. Die Verwendung der letzten gültigen Position und Ausrichtung als Schätzwerte

[26] Quelle: `https://blogs.unity3d.com/2014/01/02/occlusion-culling-in-unity-4-3-troubleshooting/`

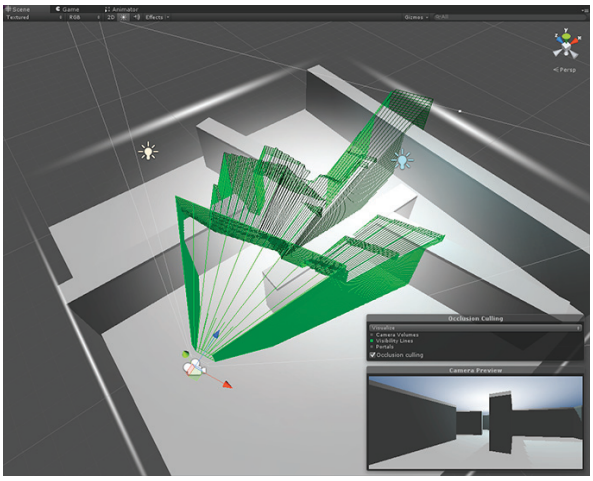

Abbildung 31: Occlusion Culling der Modelldaten[26]

zur Vororientierung im nächsten Prozessschritt stellt einen ersten Schritt zur Ermittlung von Schätzparametern dar. Gelingt es jedoch diese Werte weiter zu verbessern steigt somit die Wahrscheinlichkeit einer eindeutigen Lösung. Zusätzlich sinkt der Prozessierungsaufwand für iterative Verfahren. Die Datenmenge der spatialen Daten der LIDAR- und Kamera Sensoren übersteigt die Datenmenge der gerechneten Positions- und Orientierungsdaten der IMU, hier im weiteren *Motion Data* genannt, um ein Vielfaches. Teilweise werden IMU basierte Prozessdaten in einer höheren Sampelrate gegenüber den Bilddaten geliefert. Zusätzlich muss ein hinreichender Zeitraum für die Prozessierung der Transformation berücksichtigt werden. Daraus folgt, dass zwischen den Prozessschritten der Transformationen in erheblich kürzerer Zeitfolge *Motion Data* übertragen werden und somit Daten für eine Verbesserung der Schätzwerte zur Verfügung stehen. Auch wenn diese Werte einem Fehlertrend unterliegen ist dieser eher für größere Zeiträume relevant. Somit können die Startparameter P_i der Transformation zum Zeitpunkt i mit Hilfe der *Motion Data* weiter angenähert werden. Der Korrekturwert der Position $\triangle P_{md}$ und Orientierung $\triangle O_{md}$ ergibt sich als Differenz der *Motion Data* des letzten Intervalles.

$$P_{korr} = \triangle P = P_{md_i} - P_{md_{i-1}} \tag{31}$$

$$O_{korr} = \triangle O = O_{md_i} - O_{md_{i-1}} \tag{32}$$

mit	P_{korr}, O_{korr}	Korrekturwert der Position und Orientierung
	P_{md_i}, O_{md_i}	Position und Orientierung *Motion Data* Zeitpunkt i
	$P_{md_{i-1}}$, $O_{md_{i-1}}$	Position und Orientierung *Motion Data* Zeitpunkt $i-1$

Mit der Annahme, das die Fehlerfortpflanzung der gerechneten IMU Daten als *Motion*

Data-Korrekturwerte P_{korr} und O_{korr} kleiner als die Positionsabweichung im Zeitintervall der Positionen P_{i-1} und P_i gemäß Formeln 33 bis 38 ausfällt.

$$\varepsilon_{P_{korr}} = |P_r - P_i - P_{i-1} + P_{korr}| \tag{33}$$

$$\varepsilon_{P_{Pos}} = |P_r - P_i - P_{i-1}| \tag{34}$$

$$\varepsilon_{P_{korr}} < \varepsilon_{P_{Pos}} \tag{35}$$

$$\varepsilon_{O_{korr}} = |O_r - O_i - O_{i-1} + O_{korr}| \tag{36}$$

$$\varepsilon_{O_{Pos}} = |O_r - O_i - O_{i-1}| \tag{37}$$

$$\varepsilon_{O_{korr}} < \varepsilon_{O_{Pos}} \tag{38}$$

mit	$\varepsilon_{P_{korr}}$, $\varepsilon_{P_{Pos}}$	Fehler der Position mit und ohne Korrektur
	$\varepsilon_{O_{korr}}$, $\varepsilon_{O_{Pos}}$	Fehler der Orientierung mit und ohne Korrektur
	P_r, O_r	Reale Position und Orientierung
	P_i, O_i	gerechnete Position und Orientierung zum Zeitpunkt i
	P_{i-1}, O_{i-1}	gerechnete Position und Orientierung zum Zeitpunkt $i-1$

In der Firmware des UAV sind bereits wesentliche Funktionalitäten implementiert. Diese können wie folgt Kategorisiert werden:

- Flugsteuerung des Piloten,
- Fluglagestabilisierung durch den Autopiloten auf der Basis von Sensordaten der integrierten IMU,
- Antikollisionslösung bei Verwendung des Sensors *Guidance*,
- integrierte Navigationslösung auf der Basis von GPS-Daten für Flugrouten auf der Basis von Way-Points sowie Basisfunktionen wie *Return to Home*, *Follow Me* oder automatisierte Landung.

Soll eine Positions- und Lagebestimmung innerhalb von geschlossenen baulichen Strukturen ohne GNSS-Empfang implementiert werden, stellt sich die Frage der Integration in das vorhandene Steuerungskonzept. Mit der Bestimmung einer Position und Ausrichtung im 3D-Modellkoordinatensystem steht ein Bezugssystem für die Navigation zur Verfügung. Erfolgt die Planung einer Route in diesem Koordinatensystem, so können die berechneten Koordinaten direkt für die Navigation verwendet werden. Das Gleiche

gilt für die Orientierung im Bezugssystem. Im aktuellen System wird diese durch den in der *DJI GPS-Compass Pro Plus* Baugruppe integrierten Kompass geliefert. Wird die Baugruppe vom System getrennt, können die berechneten Positionsdaten die Daten der GPS-Baugruppe sowie die berechnete Orientierung die Daten des Kompass ersetzen. Gelingt es diese an den Autopilot alternativ zur *DJI GPS-Compass Pro Plus* Baugruppe zu übertragen, so sind keine Eingriffe in die werksseitige Navigationslösung erforderlich. Abb. 32 stellt die Einbindung der in dieser Arbeit vorgestellten Lösung in das bestehende Steuerungskonzept dar.

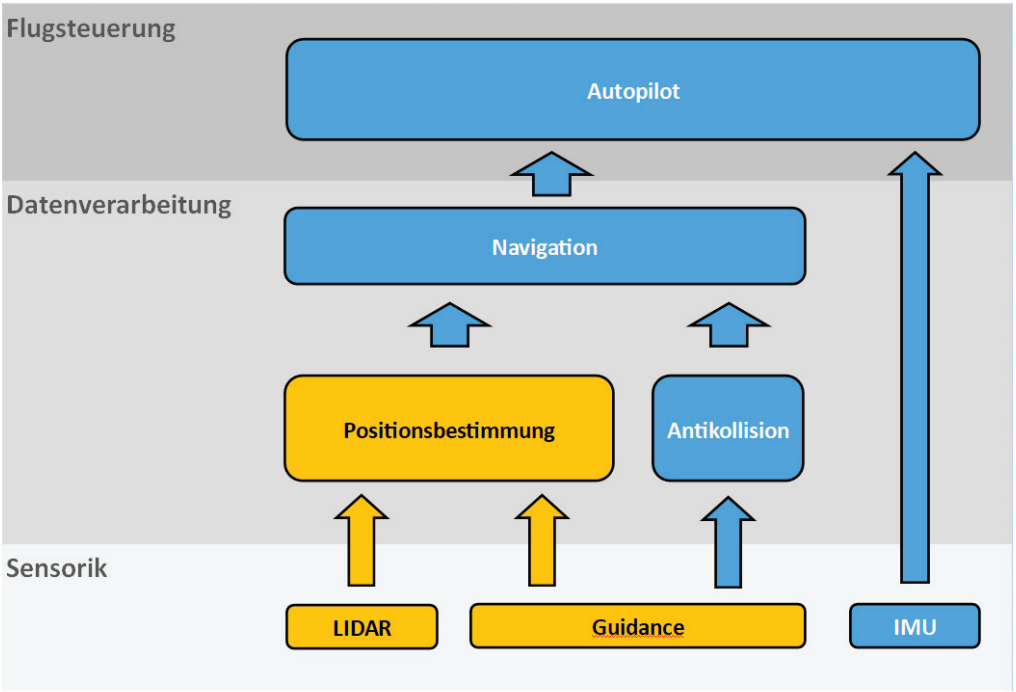

Abbildung 32: Einbindung der Positionsbestimmung in das Gesamtkonzept

7. Software Developer Kits und Bibliotheken

7.1. Robot Operation System - ROS

Das ROS stellt ein weit verbreitetes Open Source Framework für die Programmierung von Robotersystemen dar. Es unterstützt die Einbindung von Sensoren und Aktoren auf einem einzelnen Rechner wie auch in einem Netzwerk. Auf der Basis des Prinzips verteilter Anwendungen sowie einer zentralen Management-Software, dem ROS-Core, ist es möglich mit verhältnismäßig geringem Aufwand komplexe Anwendungen zu erstellen. Das ROS stellt hierfür spezielle Mechanismen und Werkzeuge zur Verfügung. ROS wurde für den Einsatz auf diversen LINUX Distributionen entworfen, steht aber auch für das Betriebssystem MAC OS X zur Verfügung.

Ein umfangreiches WIKI ist unter der URL `http://wiki.ros.org/de` zu finden. Quellcodes werden unter dem GIT Repository zur Verfügung gestellt.
Die ROS-Distribution startete im Jahr 2010 mit *ROS Box Turtle* bis zur aktuellen Version *ROS Melodic* ab den Jahr 2018. Die Einführung von *ROS Noetic* ist für das Jahr 2020 geplant. Aktuell werden die Distributionen *ROS Kinetik* aus dem Jahr 2016 sowie *ROS Melodic* supported. Der aktuelle Stand der supporteten Distributionen kann unter der URL `http://wiki.ros.org/Distributions` abgefragt werden. Für die Code-Entwicklung in ROS stehen den Entwicklern mehrere Implementierungen zur Verfügung. Hier können Programmiersprachen wie Python, C++, LISP, Java, JavaScript und weitere verwendet werden. Einige sind jedoch nur als experimentelle Bibliotheken implementiert. Aktuell werden die Entwicklungen vorrangig in C++ und Python betrieben.
ROS kann als Meta-Betriebssystem verstanden werden, welches ein Dateisystem sowie eine Peer-to-Peer Netzwerk von Prozessen zur Verfügung stellt sowie deren Kommunikation regelt. Das ROS Dateisystem stellt Datei basierte Ressourcen wie

- Packages,
- Manifests,
- Stacks,
- Stack Manifests,
- Message Beschreibungsdateien,
- Services

und weitere zur Verfügung. Für die Softwareentwicklung stellen die Packages das wesentliche Element dar. In den jeweiligen Packages werden alle Code-, Konfigurations-, Bibliotheks- sowie alle weiteren Dateien für die Entwicklung eines konkreten Laufzeitprozesses (Node) zusammengefasst. Das Peer-to-Peer Netzwerk von ROS-Prozessen (ROS Computation Graph) beinhaltet folgende Komponenten

- Nodes,
- ROS Master (auch als ROS Core bezeichnet),
- Parameter Server,
- Messages,
- Topics und
- Bags.

Der ROS Master stellt das zentrale Element der ROS Kommunikation dar. Er registriert die Namen der beteiligten Elemente und stellt diese den Elementen für die Kommunikation untereinander zur Verfügung. Ein ROS Master muss in jedem Netzwerk zwangsläufig gestartet sein.
In der Netzkommunikation spielen Namen eine wichtige Rolle. Die Identifikation und Kommunikation innerhalb des ROS Netzwerkes erfolgt in erster Linie über die Namensvergabe. Sowohl Nodes wie auch Messages und Topics werden über ihre Namen identifiziert. Neben den Nodes stellen Topics und Messages weitere bedeutende Elemente in einem ROS Netzwerk dar. Topics bezeichnen Themen unter denen Nodes Nachrichten, also Messages austauschen. Nodes registrieren sich selbst sowie alle durch sie veröffentlichten oder abonnierten Topics nach dem Start beim ROS Master.
Der Austausch von Nachrichten erfolgt grundsätzlich unter einem zugehörigen Thema (Topic). Hierbei stellen Nodes Nachrichten unter einem Thema zur Verfügung (publish) oder empfangen Nachrichten unter diesem Thema (subscribe). Die Zuordnung der benötigten Topics und Messages erfolgt bei Start eines Node durch den ROS Master, die weitere Kommunikation der Nodes jedoch ohne Beteiligung des ROS Masters. Da Nodes Messages, also Daten, unter einem Thema veröffentlichen sowie Daten unter einem Thema empfangen erfolgt die Kommunikation hierbei nicht direkt zwischen den Nodes. Die beteiligten Nodes müssen somit nicht wissen welche weiteren Nodes an der Kommunikation beteiligt sind. Somit wird die Bereitstellung sowie die Verarbeitung von Daten voneinander entkoppelt.

Die ROS Plattform stellt dem Entwickler bereits mehrere wichtige Stacks für die

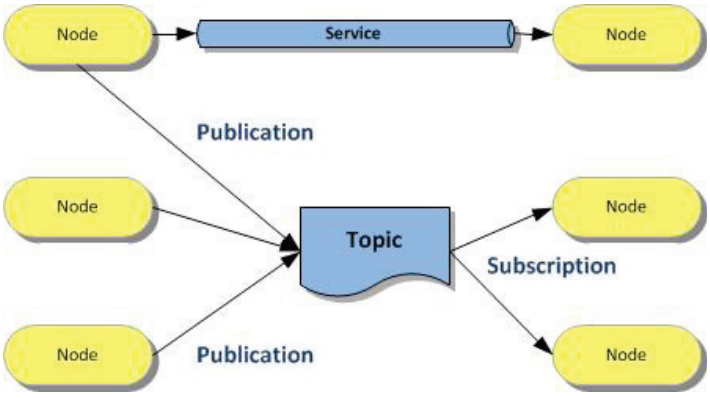

Abbildung 33: ROS Kommunikationskonzept[27]

Nutzung zur Verfügung. Hierzu zählen Packages wie

- common_msgs[28] für die wichtigsten Nachrichten die unterschiedliche Packages verwenden,

[27] Quelle: `https://www.generationrobots.com/blog/de/ros-robot-operating-system/`
[28] `http://wiki.ros.org/common_msgs`

- geometry[29], ein Meta Package für unterschiedlichste mathematische und physikalische Funktionen,
- tf[30] für Koordinatentransformationen, oder
- urdf[31] für Robot-Beschreibungsformate.

Mit den bereitgestellten Packages sind bereits viele wichtige Funktionalitäten für den Entwickler in ROS implementiert. Zusätzlich werden Tools für die Simulation und Visualisierung von Daten und Nachrichten bereit gestellt. Hierzu zählen sowohl das Simulationstool *Gazebo*, basierend auf der Game Engine *Unity*, wie auch die Visualisierungstools *rviz* oder *rqt*. Mit Hilfe dieser Werkzeuge ist es dem Entwickler möglich, seine Nodes in einer umfassenden Simulation zu testen sowie den Datenstrom oder Kommunikationsablauf zu visualisieren.
Für die Softwareentwicklung im ROS System wird vorwiegend das *catkin* Build System verwendet. *Catkin* stellt jedoch keine Entwicklungsumgebung zur Verfügung. Der Softwareentwickler erstellt seine Code Basis in seiner favorisierten Entwicklungsumgebung und nutzt für die Erstellung der Zielanwendungen das *catkin* Build System. Für unterschiedlichste Entwicklungsumgebung stehen ROS Plugins zur Verfügung. Mit Hilfe dieser Plugin ist es möglich, den aktuellen Workspace zu laden, das Build direkt aus der Entwicklungsumgebung zu starten oder den erstellten Node in der Entwicklungsumgebung zu debuggen.
Sollte die ROS Distribution nicht auf dem System installiert sein, so kann diese auf der Konsole mittels

```
1 $ sudo apt-get update
2 $ sudo apt-get install ros-melodic-catkin
```

Listing 1: Installation der ROS Distribution

installiert werden. Zusätzlich sollte in der *bashrc* Startdatei das automatische Sourcen der Distribution eingetragen werden. Mit diesem Eintrag entfällt das jeweilige Sourcen der Distribution beim Öffnen eines neuen Konsolenfensters oder der Entwicklungsumgebung. Hierzu wird in der Konsole die Datei zur Bearbeitung *bashrc* geöffnet und der notwendige Eintrag hinzugefügt.

```
1 $ gedit ~/.bashrc
```

Listing 2: Editieren der bashrc

Am Ende der Datei wird die Zeile

```
1 source /opt/ros/melodic/devel/setup.bash
```

Listing 3: Sourcen der ROS Distribution

[29] http://wiki.ros.org/geometry
[30] http://wiki.ros.org/tf
[31] http://wiki.ros.org/urdf

eingetragen. Soll eine andere ROS Distribution installiert werden, so wird in den vorstehenden Listings der Name *melodic* durch die zu installierende Distribution ersetzt.

7.2. Sensorik Bibliotheken

7.2.1. Guidance Bibliotheken

DJI stellt für die Softwareentwicklung des Sensorsystems *Guidance* unterschiedliche Softwarebibliotheken zur Verfügung. Für die Entwicklung außerhalb einer ROS Umgebung wird das *Guidance-SDK*[32] auf dem GIT-Hub zur Verfügung gestellt. Es beinhaltet im wesentlichen Beispiel Codes sowie benötigte Bibliotheken für die Entwicklung in C. Mit Hilfe des SDK können Datenverbindungen über die vorhandene USB- oder RS232-Schnittstelle etabliert und der verfügbare Datenstrom empfangen und ausgewertet werden. Hierbei ist zu beachten, dass der Umfang der zur Verfügung gestellten Daten bezüglich Datenrate und Datenart auf beiden Schnittstellen unterschiedlich verfügbar ist. Mit Anbindung an die RS232 Schnittstelle werden weniger Daten in einer geringeren Sample-Rate angeboten. Zusätzlich ist eine Konfiguration der Schnittstellen im Tool *Guidance Assistant* erforderlich. An der USB Schnittstelle kann die Freigabe

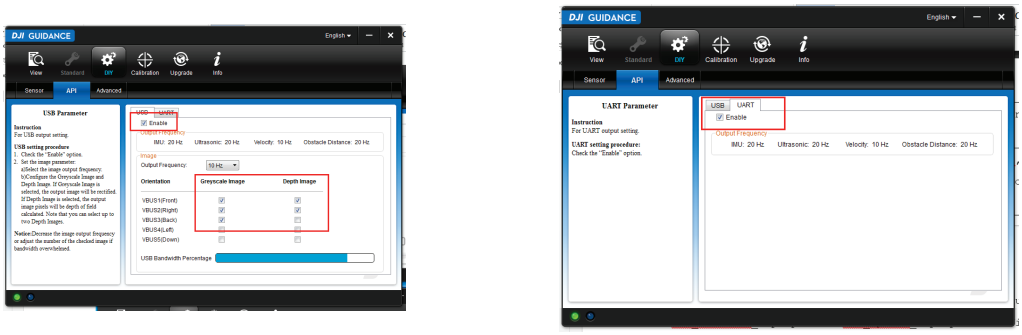

Abbildung 34: Konfiguration der API-Schnittstellen im Software Tool *Guidance Assistent*[33]

folgender Daten vorgenommen werden

- IMU Daten,
- Daten der Ultraschallsensoren,
- gerechnete Geschwindigkeitsdaten,
- gerechnete Positionsdaten,
- Abstandsdaten detektierter Hindernisse und
- Grau- und Tiefenbild-Daten der Stereokameras aller 5 Teilsysteme.

[32] `https://github.com/dji-sdk/Guidance-SDK`

[33] Quelle: `https://developer.dji.com/guidance-sdk/documentation/application-development-guides/index.html`

An der seriellen Schnittstelle ist die Freigabe der Daten für

- IMU Daten,
- Daten der Ultraschallsensoren,
- gerechnete Geschwindigkeitsdaten,
- gerechnete Positionsdaten sowie
- Abstandsdaten detektierter Hindernisse

möglich.
Das SDK stellt auf der Basis der verbauten IMU folgende drei Protokolle für Bewegungsdaten zur Verfügung,

1. *velocity data*,
2. *imu data* und
3. *motion data*.

Interessant für die aktuelle Anwendung sind hier die Datenstrukturen der *imu date*, Listing 4, sowie der *motion data*, Listing 5.

```
typedef struct _imu
{
    unsigned int frame_index; // correspondent frame index
    unsigned int time_stamp;  // time stamp of correspondent image
       captured in ms
    float        acc_x;       // acceleration of x in unit of m/s^2
    float        acc_y;       // acceleration of y in unit of m/s^2
    float        acc_z;       // acceleration of z in unit of m/s^2
    float        q[4];        // quaternion: [w,x,y,z]
}imu;
```

Listing 4: Struktur der *imu data*

```
typedef struct _motion
{
    unsigned int frame_index;
    unsigned int time_stamp;
    int      corresponding_imu_index;
    float    q0;
    float    q1;
    float    q2;
    float    q3;
    int      attitude_status;  // 0:invalid; 1:valid
    float    position_in_global_x;  // position in global frame: x
    float    position_in_global_y;  // position in global frame: y
```

```
13     float     position_in_global_z;  // position in global frame: z
14     int       position_status; // lower 3 bits are confidence. 0:
           invalid; 1:valid
15     float     velocity_in_global_x;  // velocity in global frame: x
16     float     velocity_in_global_y;  // velocity in global frame: y
17     float     velocity_in_global_z;  // velocity in global frame: z
18     int       velocity_status; // lower 3 bits are confidence. 0:
           invalid; 1:valid
19     float     reserve_float[8];
20     int       reserve_int[4];
21     float     uncertainty_location[3];// uncertainty of position
22     float     uncertainty_velocity[3];// uncertainty of velocity
23 } motion;
```

Listing 5: Struktur der *motion data*

Das Quaternion q [4], als Parameter der *imu data*, sowie q0 bis q3 der *motion_data* ist hier für eine Schätzung der Winkel α, β und γ als Lagewinkel um die drei Achsen interessant. Für die gerechnete Position stehen die Parameter *position_in_global_x*, *position_in_global_y* sowie *position_in_global_z* zur Verfügung.
Für die Implementierung des *Guidance* Sensors im ROS Umfeld steht ein zusätzliches *Guidance-ROS-SDK* zur Verfügung. Das *Guidance-ROS-SDK* ist für eine Implementierung in einem Linux Ubuntu 32- und 64-Bit Umfeld entwickelt. Das SDK implementiert ausschließlich die Anbindung mittels USB. Eine Anbindung über die vorhandene serielle Schnittstelle muss durch den Entwickler implementiert werden. Hierfür können jedoch die Beispiele aus dem *Guidance-SDK* verwendet werden. Das Publishen der Messages erfolgt wie im ROS Beispiel unter den gleichen Topics. Für die Nutzung des SDK müssen zusätzlich die Bibliotheken für *libusb* sowie bei Nutzung des Stereo-Beispielcodes für *openCV* installiert werden.

7.2.2. Bibliotheken der Hokuyo Laserscanner

Für den verwendeten Laserscanner *UST-20LX* stellt der Hersteller *Hokuyo* diverse SDK zur Verfügung. Die *URG Library* wird als C++ Bibliothek bereit gestellt[34]. Die Bibliothek wurde für

- Visual C++ 2005
- Visual C++ 2010
- gcc 4.6.0 (Linux)
- gcc 4.6.2 (MinGW)

[34] https://sourceforge.net/projects/urgnetwork/files/urg_library/

getestet. Zusätzlich werden Implementierungen für C-Sharp und Java angeboten. Mit Hilfe des SDK werden Source Codes für die Konfiguration des Laserscanner sowie das Etablieren von Datenkanäle zum Empfang der Scann- und Statusdaten zur Verfügung gestellt. Zusätzlich wird eine ROS Implementierung als *urg_node*[35] bereit gestellt. Mit Hilfe der SDK können Datenverbindungen sowohl über die serielle- wie auch IP-Schnittstelle des Scanners aufgebaut werden. Im Rahmen der ROS Integration werden die Daten des Laserscanners unter dem Topic *sensor_msgs::LaserScan* published. Eine Visualisierung der Scann-Daten in *rviz* ist in Abb. 35 dargestellt.

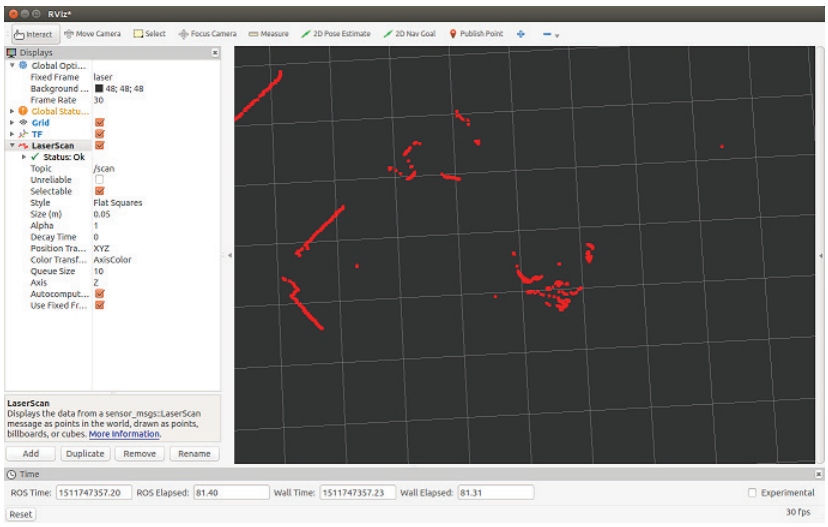

Abbildung 35: Visualisierung der Scandaten des *urg_node* in *rviz*[36]

7.2.3. Bibliotheken der Scanse Laserscanner

Der Hersteller *Scanse* stellt ein SDK[37] für die Softwareeinbindung des Scanners *sweep v1.0* zur Verfügung. Die Bibliothek stellt Codes für die Programmiersprachen

- C++,
- Java und
- Python

zur Verfügung. Die *libsweep*[38] Bibliothek stellt alle Low-Level Funktionen für

- Firmware Compatibility,
- Version and ABI Management,
- Error Handling,

[35] `http://wiki.ros.org/urg_node`
[36] Quelle: `https://sourceforge.net/p/urgnetwork/wiki/ROS_en/`
[37] `https://github.com/scanse/sweep-sdk`
[38] `https://github.com/scanse/sweep-sdk/blob/master/libsweep/README.md`

- Device Interaction,
- Full 360 Degree Scan und
- Additional Information

für die Softwareentwicklung bereit.
Neben dem vorgestellten SDK ist eine Implementierung für ROS unter *sweep-ros*[39] verfügbar. Die ROS-Implementierung benötigt eine Installation und Einbindung der *libsweep* Bibliothek. Durch den im ROS-SDK befindlichen Node werden die Scandaten unter dem Topic *sensor_msgs::PointCloud2* published.

7.3. Erstellen und Einrichten des Workspace

Für die Entwicklung mit Hilfe des *catkin* Build System wird ein eigener Workspace erstellt. In der Praxis hat sich der Name *catkin_ws* als Name für den Workspace durchgesetzt. Es kann jedoch jeder beliebige Name verwendet werden. Das Build System *catkin* ist bereits mit der Installation des ROS Systems installiert. Somit kann direkt mit der Erstellung eines eigenen Workspace begonnen werden.

```
1 $ mkdir -p ~/catkin_ws/src
2 $ cd ~/catkin_ws/
3 catkin_ws$ catkin_make
```

Listing 6: Erstellen eines eigenen Workspace für die ROS Entwicklung

Hierbei ist zu beachten, dass nach der Erstellung des Verzeichnisses *catkin_ws* sowie des Unterverzeichnisses *src* für das Build des Workspace in den Workspace Ordner *catkin_ws* gewechselt wird. *Catkin* erstellt jetzt alle für die Entwicklung benötigten Dateien und Verzeichnisse im Workspace. Nach dem erfolgreichen Build sollten sich im Workspace zusätzlich die Verzeichnisse *build* und *devel* befinden. Für die Übernahme der Änderungen wird der Workspace jetzt mittels

```
1 catkin_ws$ source devel/setup.bash
```

Listing 7: Sourcen des eigenen Workspace

gesourced. Dies erfolgt direkt aus dem Workspace Ordner heraus.
Alle für das aktuelle Projekt relevanten Source-Codes werden im Ordner *src* in eigenen Unterordnern als Packages gehalten. Werden bestehende ROS-Packages vom GIT-HUB im eigenen Workspace verwendet, so können diese direkt in den *src* Ordner des Workspace geklont werden. Soll ein eigenes Package erstellt werden, so muss auch dieses mit Hilfe des *catkin* Build Systems angelegt werden. Hierzu wird in das *src* Verzeichnis des Workspace gewechselt und mittels *catkin_create_pkg* das neue Package erstellt.

[39] https://github.com/scanse/sweep-ros

```
1 catkin_ws/src$ catkin_create_pkg oki_navigation_package std_msgs
2     rospy roscpp
```

Listing 8: Erstellen eines neuen Packages für die ROS Entwicklung

Als Parameter wird der neue Package-Name sowie die benötigten ROS Packages zu denen im Projekt Abhängigkeiten bestehen angegeben. Diese können jedoch auch später editiert werden. Mit Wechsel in den Workspace Ordner *catkin_ws* und Ausführen von *catkin_make* werden wiederum alle benötigten Unterordner und Dateien im Package-Verzeichnis erstellt.
Für das Erstellen des Projektes mittels *catkin_make* stellen die automatisch erstellten Dateien *CMakeLists.txt* und *package.xml* im Root des Package Ordners eine besondere Bedeutung dar. Die Datei *package.xml*[40] definiert alle ROS spezifischen Erstellungsregeln und Abhängigkeiten. In den jeweiligen ROS Tutorials stehen umfangreiche Anleitungen für das Hinzufügen von weiteren ROS Abhängigkeiten durch das Bearbeiten der Datei *package.xml* zur Verfügung. Die Datei *CMakeLists.txt*[41] listet alle Build Einstellungen für das aktuelle Package. Neben den Regelungen für das Einbinden eigener Benutzermessage oder -Action werden hier auch die Festlegungen der Namensgebung der eigenen Nodes sowie die Codeabhängigkeiten im Softwareprojekt geregelt. Wird der Workspace in eine IDE mit geeignetem ROS Plugin importiert, so werden diese Einträge auch durch die IDE geregelt.

7.4. Entwicklungsumgebung *Eclipse*

Für die Erstellung eigener Codedateien kann jeder Editor des Systems benutzt werden. Da das Erstellen der ausführbaren Dateien durch *catkin* erfolgt ist es nicht zwingend erforderlich eine integrierte Build Umgebung zu benutzen. Es ist jedoch empfehlenswert eine geeignete IDE für die präferierte Programmiersprache zu verwenden. Kriterium für die Auswahl der IDE sollte neben der verwendeten Programmiersprache die Verfügbarkeit eines ROS Plugin sein. Für die Entwicklung im Rahmen dieser Masterarbeit wurde *Eclipse* ausgewählt. Gründe hierfür waren

- Freie IDE ohne zusätzliche Kosten,
- Einbindung unterschiedlicher freier GNU-Compiler,
- Verfügbarkeit eines guten ROS Plugin,
- Unterstützung von Code Folding und Refactoring,
- Integration von ROS fähigem Compiler und Debugger.

[40] `http://wiki.ros.org/catkin/package.xml`
[41] `http://wiki.ros.org/catkin/CMakeLists.txt`

Diese Auswahl soll jedoch keine Wertung der Eignung gegenüber anderen IDE darstellen.

Für die Nutzung von *Eclipse* im *catkin* Umfeld ist zu beachten, das Projekte, basierend auf ROS Packages nicht mittels *Eclipse* Projekten erstellt werden können. Hierzu muss zuerst ein Package mittels *catkin* erstellt werden. Nach einem Build mittels *catkin_make* können jetzt die notwendigen *Eclipse* Projektdateien auf Konsolenebene erstellt werden. Hierzu wird auf der Konsolenebene in das Workspace Verzeichnis gewechselt und mittels folgender Kommandos die Projektdateien erzeugt.

```
1 catkin_ws$ catkin_make --force-cmake -G"Eclipse CDT4 - Unix Makefiles"
2 catkin_ws$ awk -f $(rospack find mk)/eclipse.awk build/.project >
3            build/.project_with_env && mv build/.proj ect_with_env
4            build/.project
```

Listing 9: Erstellen der *Eclipse* Projektdateien auf Konsolenebene

Nach dem Erstellen der Projektdateien wird im Workspace Unterverzeichnis *build* die Konfiguration des Projektes für den Debug Typ konfiguriert.

```
1 catkin_ws/build$ cmake ../src -DCMAKE_BUILD_TYPE=Debug
```

Listing 10: Konfiguration der *Eclipse* Projektdateien für das Debugging

Nach Abschluss dieser Arbeiten kann die IDE gestartet und der Workspace geladen werden. Hierzu wird über das Menü *File->Import* Der Import Dialog aufgerufen und der Eintrag *Existing Code as Makefile Project* ausgewählt. Im Anschluss steht der ak-

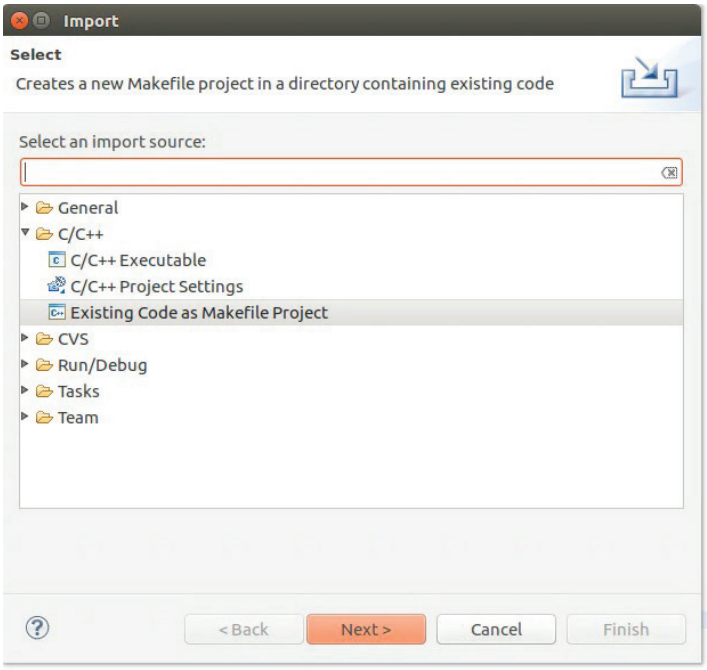

Abbildung 36: Import des ROS Workspace in Eclipse

tuelle Workspace in *Eclipse* unter dem Ordner **build** zur Verfügung. Im Projektbaum

stehen jetzt die im Workspace befindlichen Packages im Unterordner *[Source Direktory]* zur Verfügung. Neue Source Dateien können jetzt mittlels Popup Menü *New->Source File* in den jeweiligen *src* Unterverzeichnissen erstellt werden. Header Files werden im das Unterverzeichnis *include/Name-des-aktuellen-Packages* erstellt. Bei dieser Vorgehensweise aktualisiert *Eclipse* alle notwendigen Einträge in der Datei *CMakeLists.txt*. Mit dieser Vorgehensweise sind alle Voraussetungen für die Bearbeitung und Erstellung von Source- und Headerdateien eines ROS Packages mit *Eclipse* gegeben. Soll zusätzlich ein Build direkt in *Eclipse* möglich sein müssen noch weitere Einstellungen vorgenommen werden. Hierzu wird der Dialog der Projekteigenschaften über das Menü *Project->Properties* aufgerufen. In der Rubrik *C/C++ Make Poject* wird im Reiter *Make Builder* der Eintrag für *Build command:* auf den Pfad der *catkin* Distribution für *catkin_make* eingestellt. Als Ausgabeverzeichnis wird unter dem Eintrag *Build directory:* das *build* Verzeichnis des aktuellen Workspace eingetragen. Zusätzlich werden

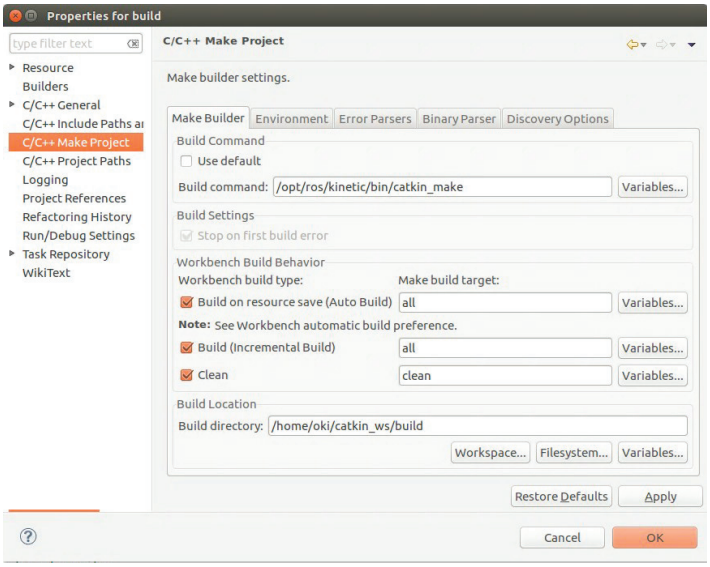

Abbildung 37: Konfiguration *Make Build* in *Eclipse*

im Reiter *Environment* die Umgebungsvariablen für

- ROS_ROOT,
- ROS_PACKAGE_PATH,
- PYTHONPATH und
- PATH

gesetzt. Damit sind alle Einstellungen für ein Build aus *Eclipse* abgeschlossen.
Ist eine Ausführung des erstellten Node oder dessen Debugging in *Eclipse* gewünscht, so wird die zugehörige Launch Konfiguration über den Menüpunkt *Run->Run configurations* vorgenommen. Um diesen Menüpunkt zu erreichen muss in die Debug Perspektive gewechselt werden. Im Dialog *Run Configurations* wird ein neuer Untereintrag

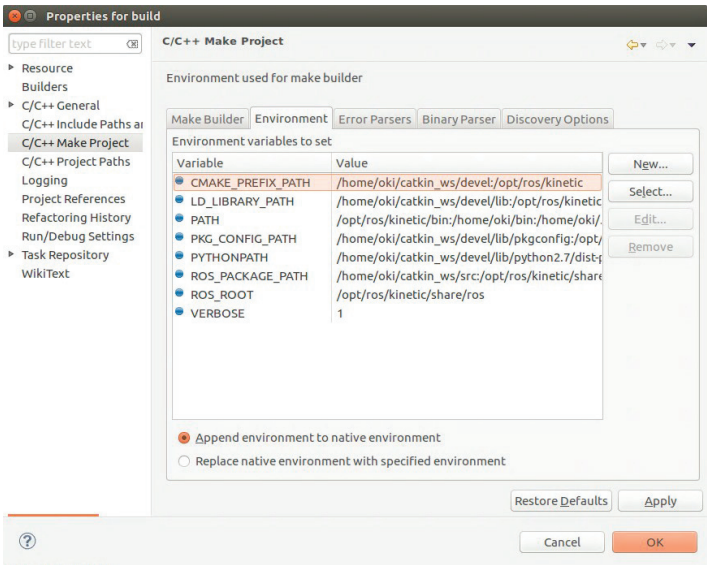

Abbildung 38: Setzen der ROS Umgebungsvariablen in *Eclipse*

für *C/C++ Applications* erstellt. Im Reiter *Main* wird der auszuführende oder zu debuggende Task ausgewählt. Die entsprechende Datei ist im Workspace unter dem Pfad */catkin_ws/devel/lib/* und Unterverzeichnis des Package Namens zu finden. Im Reiter

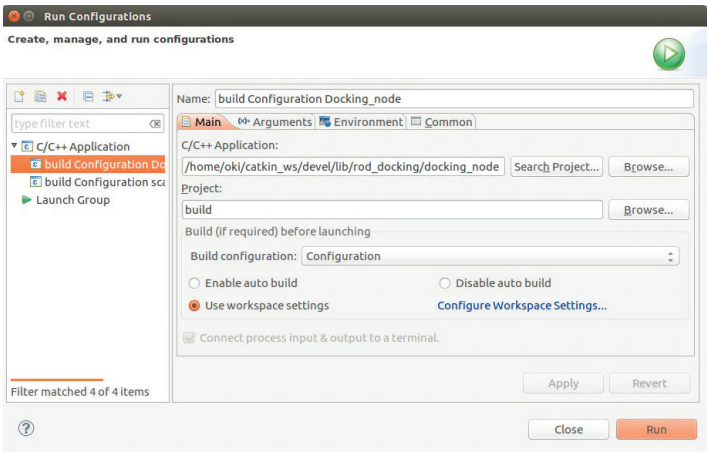

Abbildung 39: Launch Konfiguration Run/Debug

Environment werden die Umgebungsvariable mit den Werten

- ROS_MASTER_URI = http://localhost:11311 und
- ROS_ROOT = /opt/ros/kinetic/share/ros

eingetragen. Für den Eintrag ROS_ROOT wird bei einer anderen ROS Distribution als *kinetic* auf die jeweils installierte Distribution verwiesen.

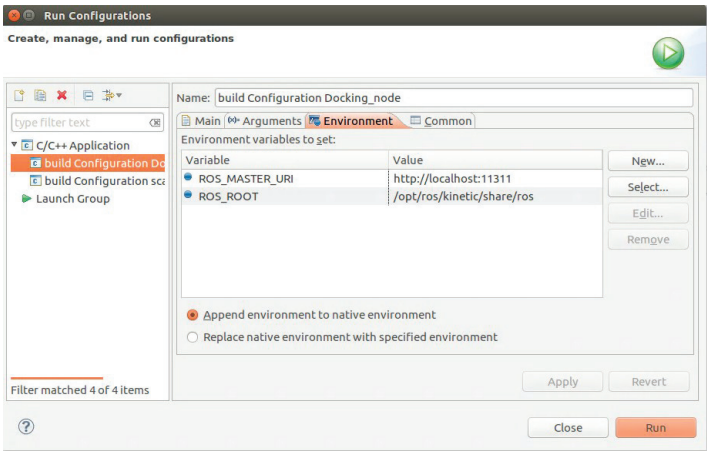

Abbildung 40: Konfiguration der Umgebungsvariablen in der Launch Konfiguration für Run/Debug

8. Positions- und Richtungsbestimmung im bekannten 3D Modell

9. Sensordatenverarbeitung

Die Daten der verwendeten Sensoren werden durch ROS Nodes an den jeweiligen Schnittstellen empfangen und in Form geeigneter Messages über Topics im ROS Netzwerk zur Verfügung gestellt. Alle Messages der Sensordaten verfügen über einen ROS Header. Mit Hilfe der im Header zur Verfügung stehenden Felder ist eine zeitliche Synchronisation der unterschiedlichen Daten möglich. Das nachfolgende Listing zeigt den in ROS definierten Header *std_msgs/Header.msg*[42].

```
1  # Standard metadata for higher-level stamped data types.
2  # This is generally used to communicate timestamped data
3  # in a particular coordinate frame.
4  #
5  # sequence ID: consecutively increasing ID
6  uint32 seq
7  #Two-integer timestamp that is expressed as:
8  # * stamp.sec: seconds (stamp_secs) since epoch (in Python the
       variable is called 'secs')
9  # * stamp.nsec: nanoseconds since stamp_secs (in Python the variable
       is called 'nsecs')
10 # time-handling sugar is provided by the client library
11 time stamp
12 #Frame this data is associated with
13 string frame_id
```

Listing 11: ROS Standard Message Header

[42] http://docs.ros.org/melodic/api/std_msgs/html/msg/Header.html

Der im Header definierte Zeitstempel *stamp* entspricht dem ROS Zeitformat *ros::Time*. Der eigene Node *oki_navigation_node* abboniert die entsprechenden Messages, syncronisiert diese unter Verwendung der veröffentlichten Zeitstempel und verarbeite die Daten für die Navigation.

9.1. Positionsdaten Guidance

Das *Guidance ROS SDK* stellt eine Implementierung der Sensordaten durch den Hersteller *DJI* im ROS System zur Verfügung. Die ROS Implementierung ist jedoch ausschließlich für eine Einbindung mittels *libusb* vorgesehen. Hiervon kann durch die vorkompilierte Bibliothek nicht abgewichen werden. Da bei der Implementierung auf dem Testsystem *AscTec Mastermind* mit der Installation der *libusb* Probleme auftraten, wurde eine eigene Umsetzung eines ROS Node mit Anbindung an die vorhandene RS232 Schnittstelle des Sensorsystems *Guidance* gewählt. Hierzu wurde die RS232 Implementierung des *Guidance SDK* des Herstellers für eine ROS Integration erweitert.
Die Programmierung erfolgte in der Programmiersprache C++ für das Betriebssystem Linux. Im Wesentlichen können alle Codedateien des Beispielprojektes *uart_example* für Linux übernommen werden. Hierzu zählen folgende Dateien:

- aus dem Ordner *Linux* die Dateien:
  - serial.cpp und
  - serial.h,
- aus dem Beispiel Ordner *uart_example* die Dateien:
  - crc16.cpp,
  - crc16.h,
  - crc32.cpp,
  - crc32.h,
  - protocal_uart_sdk.cpp,
  - protocal_uart_sdk.h
- sowie aus dem Ordner *include* des SDK-Verzeichnisses die Datei
  - DJI_guidance.h.

Die Datei *main.cpp* wird für die Einbindung in ROS neu geschrieben.
Im aktuellen Workspace wird ein neues Package für den *Guidance Node* mittels

```
1 catkin_ws/src$ catkin_create_pkg guidance_node std_msgs
2     rospy roscpp
```

Listing 12: Erstellen des Packages für den Guidance Node

angelegt. Mittels *catkin_make* werden alle notwendigen Package-Dateien und Unterverzeichnisse angelegt. Alle benötigten Code-Dateien werden in das Verzeichnis *src* sowie alle benötigten Header-Dateien in das Verzeichnis *include/guidance_node* kopiert. In der Entwicklungsumgebung *Eclipse* kann jetzt zur Kontrolle der Projektbaum mittels *F5* aktualisiert werden.
Die *guidance_oki_node.cpp* wird im Projekt durch das Erstellen einer neuen C++ Codedatei hinzugefügt. Aus der im Beispiel befindlichen *main.cpp* können die Codeteile für den Datenempfang und die Datendecodierung mittels der zur Verfügung gestellten Methode *push* und *pop* der *protocal_uart_sdk.c* übernommen werden. Im vorhandenen Beispiel ist jedoch nicht die Auswertung der *motion* Daten implementiert. Diese muss somit eigenständig erfolgen.
Für die Einbindung in das ROS System wird die Methode *main()* um die Initialisierung im ROS sowie die Bereitstellung der notwendigen *Publisher* erweitert.

```
1 int main(int argc, char** argv)
2 {
3     // Initialisierung ROS
4     ros::init(argc, argv, "GuidanceNode");
5     ros::NodeHandle gd_node("~");
6
7     // Anmeldung des Publisher fuer Motion-Data
8     //Setup Publisher
9     ros::Publisher motion_pub = gd_node.advertise<geometry_msgs::
          PoseStamped>("guidance_pos", 2);
```

Listing 13: ROS Initialisierung im Guidance Node

Für die Datenverarbeitung werden die gerechneten globalen Positionsdaten sowie die Orientierung aus der Datenstruktur *motion* verwendet. Hierzu wird die Decodierung der Daten für die Struktur der Main-Loop hinzugefügt.

```
1 if (e_motion == cmd_id)
2 {
3     motion mo;
4     memcpy(&mo, data + 2, sizeof(mo));
5     if (mo.position_status < 10)
6     {
7         PublishMotion(&motion_pub, &mo);
8     }
9 }
```

Listing 14: Decodieren der Motion Data im Guidance Node

Nach dem Füllen der Datenstruktur kann der Datensatz der aktuellen Motion-Daten veröffentlicht (publish) werden. Dies erfolgt mittels der eigenen Methode *PublishMotionData*.

```
1 void PublishMotion(ros::Publisher *pub, motion *mo)
```

```
{
    geometry_msgs::PoseStamped pos;
    pos.header.frame_id = "/guidance_position";
    pos.header.seq = mo->frame_index;
    pos.header.stamp = ros::Time::now();
    pos.pose.position.x = mo->position_in_global_x;
    pos.pose.position.y = mo->position_in_global_y;
    pos.pose.position.z = mo->position_in_global_z;

    tf::Quaternion quat;
    quat.setX(mo->q0);
    quat.setY(mo->q1);
    quat.setZ(mo->q2);
    quat.setW(mo->q3);
    quat.normalize();
    tf::quaternionTFToMsg(quat, pos.pose.orientation);
    pub->publish(pos);
}
```

Listing 15: Veröffentlichen der Motion Data im Guidance Node

9.2. LIDAR Daten

Für die Bereitstellung der LIDAR Daten wurde der Laserscanner *sweep V1.0* von *Scanse* verwendet. *Scanse* stellt in seinem ROS SDK Implementierungen für die ROS Integration bereit. Das Package kann direkt in das *src* Verzeichnis des Workspace geklont werden. Danach steht das Package Verzeichnis *sweep-ros* zur Verfügung. Zusätzlich muss wie bereits beschrieben das *sweep-sdk* installiert sein. Mittels *catkin_make* werden die ausführbaren Dateien erstellt. In der Quellcode Datei muss hier gegebenenfalls der Pfad für den Include der *sweep.hpp* angepasst werden.
Die Scan-Daten werden unter der Message *sensor_msgs::PointCloud2* veröffentlicht. Der *sweep* Namespace deklariert die Struktur *scan* mit dem Element *samples* als Vector der Struktur *sample*. Diese hält im weiteren die Daten eines Scanpunktes für Winkel, Entfernung und Intensität.

```
struct sample {
  std::int32_t angle;
  std::int32_t distance;
  std::int32_t signal_strength;
};

struct scan {
  std::vector<sample> samples;
};
```

Listing 16: Datenstuktur der Scandaten im Namespace sweep

Somit liegen die Rohdaten des jeweiligen Scan in Polarkoordinaten für Distanz und Intensität vor. Da die Daten im Message Format *PointCloud2* veröffentlicht werden, müssen diese im Vorfeld in kartesische Koordinaten umgerechnet werden. Die Speicherung der Daten erfolgt im konkreten Fall unter zu Hilfenahme der ROS Point Cloud Library und deren Konvertierung in das *PointCloud2* Message Format.

```
1  void publish_scan(ros::Publisher *pub,
2                    const sweep::scan *scan, std::string frame_id)
3  {
4      float angle;
5      int32_t range;
6      float x;
7      float y;
8      int i = 0;
9
10     pcl::PointCloud <pcl::PointXYZ> cloud;
11     sensor_msgs::PointCloud2 cloud_msg;
12     ros::Time ros_now = ros::Time::now();
13     cloud.height = 1;
14     cloud.width = scan->samples.size();
15     cloud.points.resize(cloud.width * cloud.height);
16     for (const sweep::sample& sample : scan->samples)
17     {
18         range = sample.distance;
19         angle = ((float)sample.angle / 1000); //millidegrees to
               degrees
20
21         //Polar to Cartesian Conversion
22         x = (range * cos(DEG2RAD(angle))) / 100;
23         y = (range * sin(DEG2RAD(angle))) / 100;
24
25         cloud.points[i].x = x;
26         cloud.points[i].y = y;
27         i++;
28     }
29     //Convert pcl PC to ROS PC2
30     pcl::toROSMsg(cloud, cloud_msg);
31     cloud_msg.header.frame_id = frame_id;
32     cloud_msg.header.stamp = ros_now;
33
34     ROS_DEBUG("Publishing a full scan");
35     pub->publish(cloud_msg);
36 }
```

Listing 17: Publishen der Scan Daten im sweep Node

Alternativ besteht die Möglichkeit die Scandaten im Message Format *sensor_msgs::LaserScan* zu veröffentlichen. Hier stellt sich die Überführung der Daten jedoch we-

sentlich komlizierter dar, da der Scanner nicht mit einem fixen Start- und Endewinkel arbeitet. Der aktuelle Winkel der Laserausrichtung wird für jeden Messpunkt bei jedem neuen Vollkreis individuell bestimmt. Die Parameter für *angle_min* und *angle_max* müssen somit für jeden Scan neu bestimmt werden und sind nicht durch den Scanner festgelegt.

9.3. Vororientierung der LIDAR Daten

Für die Lösung der Navigationsaufgabe wurde der *oki_navigation_node* im eigenen Package *oki_navigation* erstellt. Zur Erstellung des Packages wird im *src* Verzeichnis des Workspace wieder ein neues Package mittels *catkin_create_pkg* und *catkin_make* erstellt. In der Entwicklungsumgebung werden die zugehörigen Source- und Header-Dateien angelegt.
Der Node empfängt die Positionsdaten des *guidance_oki_node* auf dem Topic */guidance_position* sowie die Scan Daten auf dem Topic */laser_frame*.

```
1  int main(int argc, char** argv)
2  {
3      ros::init(argc, argv, "OkiNavigationNode");
4      ros::NodeHandle my_node;
5      ros::NodeHandle gd_node("~");
6
7      // Lesen der initialen Position
8      ReadInitialPosition();
9
10     // Subscribber
11     ros::Subscriber guidance_Pose_sub = gd_node.subscribe(
12         "/guidance_oki/guidance_pos", 2, GuidanceCallback);
13     ros::Subscriber scan_PC_sub = gd_node.subscribe(
14         "/pc2", 2, LaserCallback);
15     ros::Subscriber fitt_Pose_sub = gd_node.subscribe(
16         "/laser_frame", 2, FittingPoseCallback);
17
18     // Publisher
19     voLaserScan_pub = gd_node.advertise<sensor_msgs::PointCloud2>(
20         "volaser_frame", 1000);
21
22     ros::spin();
23 }
```

Listing 18: ROS-/Subscriber Initialisierung im oki_navigation_node

Für das nachfolgende Fitting der Scan Daten in das Objektmodell wird der Scan Frame mit Hilfe der gerechneten Positionsdaten des *guidance_node* vororientiert. Da die gerechneten Daten im lokalen Koordinatensystem der Guidance starten, müssen diese zu diesem Zeitpunkt mit der initialen Position und Ausrichtung des Systems korrigiert

werden. Die initiale Position ist im ROS Parameter *Nav_Initial_Pose* vom Typ *tf::Pose* veröffentlicht. Nach der initialen Korrektur wird die aktuelle Position mit den Änderungen aus den fortlaufenden Positionsdaten fortgeschrieben. Der entstehende Trendfehler wird hier nicht berücksichtigt. Eine Korrektur des Trendfehlers erfolgt erst nach einer erfolgreichen Transformation der Scan Daten in das Objektmodell (siehe Kapitel 10.4). Aus diesem Grund wird die aktuelle Position und Ausrichtung als ROS Parameter im System unter den Namen *Last_Nav_Pose* vom Typ *tf::Stamped<Pos>* veröffentlicht. Mit Start des Node wird die initiale Position geladen und als aktuelle Position im Parameterspeicher abgelegt.

```
1 void ReadInitialPosition()
2 {
3     tf::Vector3 initialPosition = ReadInitialNavPosition();
4     tf::Quaternion initialOrientation = ReadInitialNavOrientation();
5 
6     lastPointDiv = lastGuidancePosition + initialPosition;
7     lastOrientationDiv = lastGuidanceOrientation + initialOrientation;
8 }
```

Listing 19: Initialisierung der aktuellen Position

Nach der Initialilierung steht eine gültige Position mit Orientierung für die Vororientierung zur Verfügung. Auf diese kann bei Aufruf eines neuen Laserscan für die Vororientierung zurück gegriffen werden.

```
1 void GuidanceCallback(const geometry_msgs::PoseStamped::ConstPtr&
    guidance_msg)
2 {
3     tf::Stamped<tf::Pose> guidancePos;
4     tf::poseStampedMsgToTF(*guidance_msg, guidancePos);
5 
6     tf::Vector3 position;
7     tf::pointMsgToTF(guidance_msg->pose.position, position);
8     tf::Quaternion orientation;
9     tf::quaternionMsgToTF(guidance_msg->pose.orientation, orientation)
        ;
10    // Differnz der Position fortfuehren
11    tf::Vector3 posDiv = position - lastGuidancePosition;
12    lastGuidancePosition = position;
13    lastPointDiv += posDiv;
14 
15    // Differenz der Orientierung fortfuehren
16    tf::Quaternion qDiv = orientation * lastOrientationDiv.inverse();
17 }
```

Listing 20: Fortführen der Differenzwerte der Guidance Positionsnachricht

Für jede neue Positionsnachricht des Guidance Node wird jetzt die Differenz zur letzten Positionsnachricht ermittelt und die Differenzwerte fort geführt.

Die im Subscriber-Callback *scan_PC_sub* empfangene Punktwolke wird mit den aktuellen Positions- und Orientierungsdaten vororientiert. Dabei stellt die aktuelle Position *lastGuidancePose* den Translationsvektor für die Transformation dar. Durch Rotation der Punktkoordinaten mit dem Quaternion *lastGuidanceOrientation* wird die Punktwolke orientiert. Die so vororientierte Punktwolke wird durch den Publisher

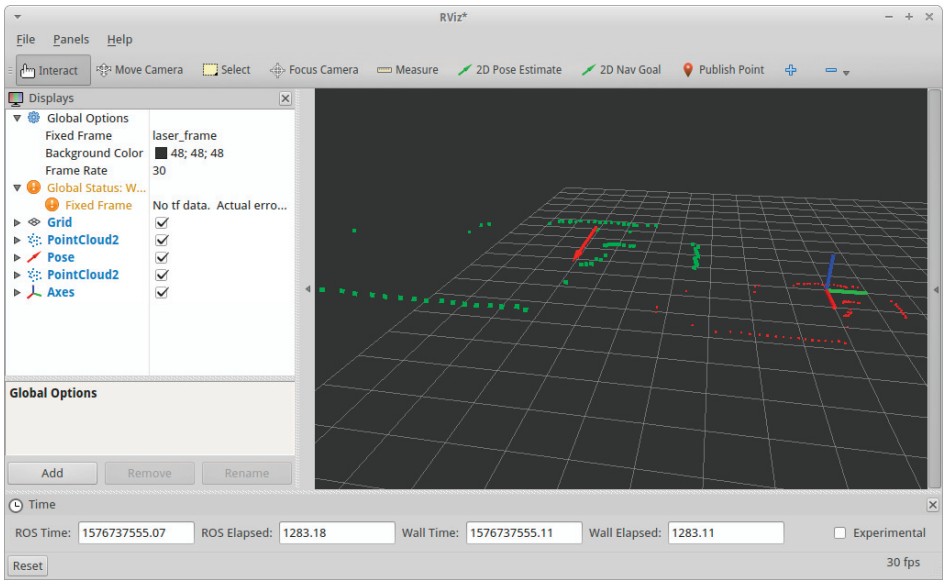

Abbildung 41: Punktwolke vor (rot) und nach (grün) der Vororientierung

fitt_Pos_sub veröffentlicht und steht nun dem *fitting_node* zur Verfügung.

```
1  void LaserCallback(const sensor_msgs::PointCloud2::ConstPtr& msg)
2  {
3      // letzte Position und Orientierung lesen
4      tf::Vector3 lastPosition = ReadLastNavPosition();
5      tf::Quaternion lastOrientation = ReadLastNavOrientation();
6
7      // Differenzen beruecksichtigen
8      tf::Vector3 position = lastPosition + lastPointDiv;
9      tf::Quaternion orientation = lastOrientation * lastOrientationDiv;
10
11     pcl::PCLPointCloud2 pcl_pc2;
12     pcl_conversions::toPCL(*msg,pcl_pc2);
13     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl
           ::PointXYZ>);
14     pcl::fromPCLPointCloud2(pcl_pc2,*cloud);
15
16     // Vororientieren
17     for (int i = 0; i < cloud->width; i++)
18     {
19         tf::Vector3 vp(cloud->points[i].x, cloud->points[i].y,
20               cloud->points[i].z);
21         vp = vp + position;
22         vp = vp.rotate(orientation.getAxis(), orientation.getAngle());
```

```
23          cloud->points[i].x = vp.x();
24          cloud->points[i].y = vp.y();
25          cloud->points[i].z = vp.z();
26      }
27
28      sensor_msgs::PointCloud2 cloud_msg;
29      ros::Time ros_now = ros::Time::now();
30      pcl::toROSMsg(*cloud, cloud_msg);
31      cloud_msg.header.frame_id = "laser_frame";
32      cloud_msg.header.stamp = ros_now;
33      // Cloud publishen
34      voLaserScan_pub.publish(cloud_msg);
35  }
```

Listing 21: Vororientierung der Laser-Scan-Punktwolke

Die durch den *fitting_node* korrigierte Position und Orientierung wird durch den Subscriber *fitt_pose_sub* empfangen. Auf der Basis der durch das Fitting verbesserten Position wird die aktuelle Guidance Position korrigiert.

10. Fitting

Das Fitting realisiert die Transformation der geschätzten Position in eine Position im gegebenen 3D Modell mit bester Einpassung der vororientierten Scan-Punktwolke des Laserscanners. Für diese Aufgabe wurde der *oki_fitting_node* erstellt. Er implementiert folgende Funktionalitäten:

- Laden der OBJ-Modelldaten,
- Minimierung des 3D Datenmodelles,
- Transformation der vororientierten Punktwolke des *oki_navigation_node* in das 3D Modell,
- Transformation und Veröffentlichung der transformierten Position und Ausrichtung des Systems im 3D Modell.

Der Node führt die Registrierung im ROS-Master durch und initiiert den Subscriber für den vororientieten Laser-Scann auf dem Topic */volaser_frame* sowie den Publisher für die transformierte Position und Ausrichtung des Systems auf dem Topic *fitting_pose*. Zusätzlich wird das 3D Objektmodell in den Speicher geladen.

```
1  ros::Publisher voLaserScan_pub;
2
3  int main(int argc, char** argv)
4  {
5      ros::init(argc, argv, "OkiFittingNode");
```

```
6       ros::NodeHandle my_node;
7       ros::NodeHandle fi_node("~");
8
9       std::string objFileName;
10      fi_node.getParam("objFileName", objFileName);
11      objl::Loader objLoader;
12      if (!objLoader.LoadFile(objFileName))
13      {
14          ROS_ERROR("OBJ-Datei %s konnte nicht geladen werden!",
15              objFileName.c_str());
16          return 0;
17      }
18      else
19      {
20          ROS_INFO("OBJ-Datei %s geladen!", objFileName.c_str());
21          ROS_INFO("%d Vertices geladen", objData.GetLoadedVertices());
22          ROS_INFO("%d Indizes geladen", objData.GetLoadedIndizes());
23          ROS_INFO("%d Meshes geladen", objData.GetLoadedMeshes());
24      }
25
26      // Subscribber
27      ros::Subscriber voScan_sub = fi_node.subscribe(
28          "/volaser_frame", 2, VOScanCallback);
29
30      // Publisher
31      voLaserScan_pub = fi_node.advertise<geometry_msgs::PoseStamped>(
32          "/fitting_pose", 1000);
33
34      ros::spin();
35  }
```

Listing 22: ROS Initialisierung im oki_fitting_node

Alle Mechanismen des Ladens der Modelldaten, deren Minimierung sowie des ICP-Algorithmus wurden in der Klasse *OBJClass* implementiert. Die einzelnen Funktionalitäten und deren Umsetzung werden in den folgenden Kapiteln beschrieben.

10.1. 3D Modelldaten

Als Austauschformat für das 3D Modell der Umgebung wurde das OBJ Format gewählt. Wie in Kap. 4.6 erläutert, wird auf das gleiche Datenformat ohne Konvertierung der Modelldaten zwischen externer Visualisierung und UAV Rechnersystem zugegriffen. Für zukünftige Erweiterungen ist die Bereitstellung von Metadaten gewünscht. Hierfür ist die Verwendung der Materialdateien des OBJ Formates geeignet.
Das digitale Objektmodell der Umgebung wird im OBJ Wave Front Format über einen

integrierten Loader importiert. Hierfür wurde auf die C++ Bibliothek *OBJ_Loader*[43] des Autors Robert Smith zurück gegriffen. Die Bibliothek stellt eine Headerdatei mit allen zugehörigen Typen, Methoden und Klassen für das Laden sowie die Datenhaltung von 3D Modellen und zugehörigen Materialdateien im Wave Front Format zur Verfügung. Die Klasse *OBJClass* kapselt die in der Bibliothek zur Verfügung gestellten Single Header als Wrapper. Mit Hilfe der veröffentlichten Methode *LoadOBJ* wird eine OBJ-Datei in den Speicher geladen werden. Nach dem Laden stehen folgende Datenelemente vom Typ *vector<>* zur Verfügung:

- LoadedMeshes,
- LoadedVertices,
- LoadedIndices,
- LoadedMaterials.

Die Implementierung der Punkt-Koordinaten wurde mittels selbst definierter Struct-Objekte im Header File *OBJ_Loader.h* mit einem minimalen Funktionsumfang durchgeführt. Somit ist es sinnvoll und notwendig die vorhandenen Vertex Objekte in der Datenverarbeitung in *tf:Vector3* Objekte zu überführen. Das Listing 23 zeigt das Interface der Klasse *OBJClass*.

```
1  class OBJClass
2  {
3  protected:
4
5  public:
6      OBJClass();
7
8      bool LoadOBJFile(std::string fileName);
9      int GetLoadedMeshes();
10     int GetLoadedVertices();
11     int GetLoadedIndizes();
12
13     // Minimieren der Modelldaten
14     int Culling(tf::Vector3 position, double minRange, double maxRange
           );
15     // Bestimmung der Zielpunktwolke
16     pcl::PCLPointCloud2 GetTargetPointCloud(pcl::PCLPointCloud2 scan);
17     // Fitting mittels ICP
18     tf::Pose Fitt(pcl::PCLPointCloud2 scan);
19     // Transformation der Punktwolke des Ausgangssystems
20     pcl::PCLPointCloud2 Transform(pcl::PCLPointCloud2 & source, tf::
           Pose pose);
21     // Varianz
```

[43] https://github.com/Bly7/OBJ-Loader

```
22      double GetVariance(pcl::PCLPointCloud2 & source, pcl::
            PCLPointCloud2 & dest);
23  };
```

Listing 23: Klasse OBJClass

10.2. Minimierung der 3D Modelldaten

Auf die Notwendigkeit der Minimierung der Modelldaten wurde bereits im Kapitel 6 eingegangen. Unabhängig vom gewählten Verfahren besteht hierbei die Möglichkeit der Reduzierung der Modelldatenmenge für die nachfolgenden mathematischen Operationen. Die Klasse *OBJClass* implementiert die Methode *SetCulling*. Der Methode wird die aktuelle geschätzte Position im Modellkoordinatensystem sowie ein minimaler und maximaler Range übergeben. Die Parameter *minRange* und *maxRange* bestimmen die minimale und maximale Entfernung der zu berücksichtigenden Geometrieobjekte und setzen somit die untere und obere Entfernungsgrenze des Sichtbarkeitsalgorithmus.
Die Methode liefert alle Indizes der Mesh-Objekte aller Geometrien des geladenen Model, die im 360 Grad Umkreis um die angegebene Position im Rahmen der Sichtbarkeitsanalyse selektiert werden. Zur Vermeidung von Datenredundanz werden die Indizes der selektierten Mesh Objekte im Vektor *selectedMeshes* gespeichert.
Im ersten Durchlauf des Algorithmus wird der gesamte Vektor *LoadedMeshes* nach den Kriterien *minRange* und *maxRange* durchsucht und eine erste Untermenge aller in Betracht kommenden Meshes gebildet. Dies hat den Vorteil, dass die zu bearbeitende Datenmenge mit einer geringen Laufzeit von $\mathcal{O}(n)$ erheblich minimiert wird. Im zweiten Durchlauf werden die Meshes mit Hilfe des Occlusion Culling (Verdeckung) durchsucht. Das Occlusion Culling ist eine in der Computer Grafik weit verbreitete Methode zur Sichtbarkeitsanalyse. In der Sichtbarkeitsprüfung werden nacheinander folgende Verfahren zur weiteren Minimierung von Meshes angewendet:

- Back Face Culling, entfernen von Flächen, deren Flächen Normale mit einem Winkel $> 90°$ zum Beobachtungspunkt ausgerichtet sind,
- Portal Rendering,
- Culling für Teilverdeckungen.

Für das Back Face Culling ist wiederum eine Laufzeit von $\mathcal{O}(n)$ gegeben. Der mathematische Aufwand ist jedoch höher als bei der Tiefenprüfung. Das Portalrendering sowie das Culling für Teilverdeckungen stellt den höchsten Aufwand und somit mit die größte Laufzeit dar. Besteht kein hoher Anspruch an die Kantengenauigkeit, so kann der Aufwand hier jedoch minimiert werden. Als Ergebnis der Sichtbarkeitsprüfung steht eine Liste von Meshes für die Bestimmung der Zielpunktwolke zur Verfügung.

10.3. Bestimmung der Zielpunktwolke

Für die Koordinatentransformation der gegebenen Punktwolke des Laserscanners (Ausgangssystem) in das 3D-Model (Zielsystem) wird ein ICP-Algorithmus verwendet. Die notwendigen Transformationsparameter werden mittels Helmert Transformation bestimmt. Hierbei werden die Koordinatenpunkte eines Ausgangssystems an korrespondierende Koordinatenpunkte des Zielsystems bestmöglich eingepasst. Zur Überführung des Ausgangssystems in das Zielsystem werden korrespondierende Punkte unter der Bedingung des *nearest neightbor* benötigt. Das Zielsystem stellt eine Menge von Geometrieobjekten zur Verfügung. Die im Modell befindlichen Vertices (Koordinatenpunkte) dienen der Beschreibung der Geometrieobjekte und stellen keine Koordinatenpunkte eines Zielsystems für eine Transformation dar. Aus diesem Grund müssen die korrespondierenden Koordinatenpunkte ermittelt werden.
Durch die Tatsache, dass die Punktwolke im Ausgangssystem bereits durch die Lagewinkel um die X- und Y-Achse sowie die Höhe vororientiert ist, kann die Transformationsgleichung auf eine 3 Parameter Transformation vereinfacht werden. In Folge ist es somit möglich die korrespondierenden Koordinatenpunkte im 3D-Modell als Schnittpunkte eines horizontal orientierten Vektors der Geometrien zu den Koordinatenpunkte des Ausgangssystems zu bestimmen (Abb. 42). Hiebei muss beachtet werden, dass die

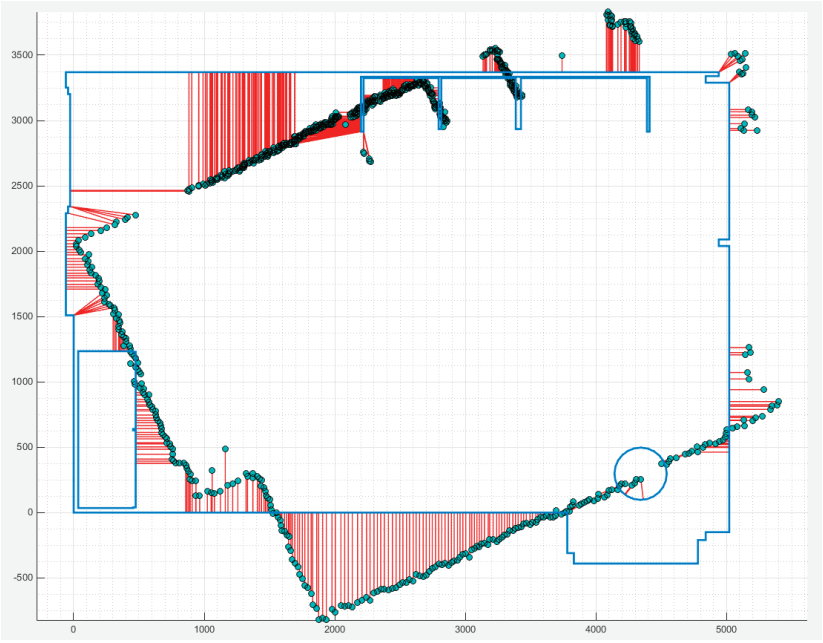

Abbildung 42: Bestimmung korrespondierender Koordinatenpunkte im Zielsystem (Z-Ansicht)

Schnittvektoren zur Punktbestimmung in der X/Y Ebene des Zielsystems liegen. Auf Grund der Minimierung der Transformationgleichung erfolgt die Transformation in der X/Y Ebene. Abb. 43 stellt das Prinzip in der vertikalen Ansicht dar. Das Datenmodell stellt drei geometrische Objekte aus den Meshes für eine Schnittpunktberechnung zur Verfügung:

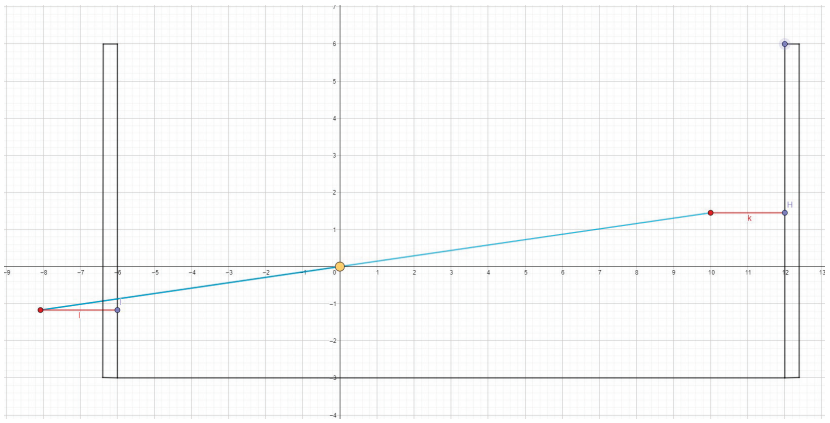

Abbildung 43: Bestimmung korrespondierender Koordinatenpunke im Zielsystem (vertikale Ansicht)

- Flächenobjekt Mesh,
- Kanten des Mesh als Linienobjekt,
- Eckpunkte des Mesh als Koordinatenpunkte.

Für die Schnittpunktberechnung auf die Mesh-Flächen wird die Lotrechte $\overrightarrow{v_{Lot}}$ der Fläche in der X/Y Ebene bestimmt. Schneidet die Gerade der Lotrechten aus dem zu betrachtenden Scann-Punkt s die Fläche im Punkt s_F, wird die Länge des resultierenden Vektors $|\overrightarrow{s_{scan}s_F}|$ berechnet. Wird kein Schnittpunkt ermittelt, entfällt die betreffende Fläche. Für einen Scann-Punkt wird somit eine Menge von Schnittpunkten auf Flächen der Modelldaten mit den zugehörigen Längen der Vektoren $\overrightarrow{s_{scan}s_F}$ ermittelt. Der korrespondierende Schnittpunkt des Zielsystems ist der Schnittpunkt mit der kleinsten Länge des Vektors $\overrightarrow{s_{scan}s_F}$ (nearest nightbor).
Die Schnittpunktberechnung erfolgt analog für alle Kanten der Meshes. Die Kanten werden als Linienobjekte aus den korrespondierenden Vertices eines Mesh gebildet. Da Meshes an Kanten verbunden sein können ist es aus Gründen der Laufzeitoptimierung sinnvoll bestehende Doppelungen von Kanten an zwei Meshes im Vorfeld zu erkennen und zu entfernen. Dies gilt analog für alle Eckpunkte.
Als Ergebnis der Schnittpunktberechnung für die Flächen, Kanten und Eckpunkte liegen für jedes dieser Geometrieobjekt ein nächster Nachbar zu einem Scannpunkt des Ausgangssystems vor. Der für die korrespondierende Punktwolke des Zielsystems zu findende Punkt ist der nächsten Nachbarn aus diesen Schnittpunkten. Dieser Vorgang wird für alle Scann-Punkte des Ausgangssystems durchgeführt.

10.4. Fitting mittels Iterative Closed Point Algorithmus

Für die Transformation der Scann-Punktwolke in das 3D-Model wird der ICP-Algorithmus verwendet. Der Algorithmus arbeitet iterativ. In jedem Iterationsschritt werden folgende Programmschritte abgearbeitet:

1. Bestimmung korrespondierender Punktpaare,
2. Fehlerbestimmung,
3. Bestimmung der Transformationsparameter,
4. Transformation der Scann Punktwolke,
5. Bestimmung korrespondierender Punktpaare,
6. Fehlerbestimmung und
7. Prüfung der Fehlerminimierung und Setzen der Abbruchbedingung.

Der Algorithmus wird bei Empfang einer vororientierten Punktwolke im Callback *VOScanCallback* des Subsriber auf den Topic */volaser_frame* aufgerufen. Hierfür steht in der Klasse *OBJClass* die Methode *Fitt* zur Verfügung. Der Methode werden die Daten des vororientierten Laser-Scann vom Typ *pcl::PCLPointCloud2* übergeben. Der Rückgabewert stellt die korrigierte Position dar.

```
1  void VOScanCallback(const sensor_msgs::PointCloud2::ConstPtr& msg)
2  {
3      pcl::PCLPointCloud2 pcl_pc2;
4      pcl_conversions::toPCL(*msg, pcl_pc2);
5      pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl
           ::PointXYZ>);
6
7      // Fitting der Scann-Punktwolke in das 3D Model
8      tf::Pose fittPose = objData.Fitt(pcl_pcl2);
9      // Publishen der korrigierten Position
10     if (objData.SolvedFitting)
11         PublishPose(fittPose);
12 }
```

Listing 24: Initialisierung der aktuellen Position

Die Iteration wird beendet, wenn der Fehler der Passung nicht weiter minimiert, die minimale Fehlergrenze unterschritten wird oder die maximale Anzahl erlaubter Iterationsschritte erreicht ist. Eine Begrenzung der Iterationstiefe ist notwendig, um die Laufzeit des Algorithmus zu begrenzen wenn die Abbruchbedingung mit einer minimalen Fehlergrenze nicht erreicht werden kann. Dies hat jedoch keine oder eine Lösung mit schlechter Genauigkeit zur Folge.
Nach Bestimmen der korrespondierenden Punktwolke im Zielsystem wird das Fehlerquadrat über alle Punktpaare berechnet. Der Fehler ε für ein Punktpaar mit s_{Scan} als Punkt der Scann-Punktwolke und s_A als korrespondierender Punkt des Zielsystem,

entspricht der Länge des Vektors den das Tupel bildet.

$$\varepsilon = |\overrightarrow{s_{scan}s_A}| \tag{39}$$

$$\overline{x} = \frac{\sum_{n=1}^{N} \varepsilon}{N} \tag{40}$$

$$\sigma^2 = \frac{\sum_{n=1}^{N} (\varepsilon - \overline{x})^2}{N} \tag{41}$$

Das Fehlerquadrat σ^2 wird für die Auswertung der Abbruchbedingung gespeichert. Nach der Bestimmung des Fehlerquadrates werden die Transformationsparameter für die Ausgleichung ermittelt. Auf Grund des Vorliegens zweier dreidimensionaler Koordinatendatensätzen in jeweils rechtwinkligen Koordinatensystemen kann auf eine Ausgleichung mittels 3D-Helmert-Transformation zurückgegriffen werden [Nie01]. Die Transformationsfunktion kann wie folgt aufgestellt werden:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{T} + m \cdot \mathbf{R}\,(\Phi, \Theta, \Psi) \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{42}$$

In Formel 42 stellt die Matrix $[X, Y, Z]$ den Datensatz im Zielkoordinatensystem und die Matrix $[x, y, z]$ den Datensatz im Ausgangskoordinatensystem dar. Der Parameter m stellt den Maßstabsparameter, $\mathbf{R}$ die Rotationsmatrix und $\mathbf{T}$ die Translationsmatrix dar. In der aktuellen Betrachtung wird der Maßstabsparameter auf 1 gesetzt. Somit kann die Funktion aus 42 vereinfacht werden.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{T} + \mathbf{R}\,(\Phi, \Theta, \Psi) \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{43}$$

Die Vororientierung der Scann-Punktwolke erfolgt durch die Positionsdaten des *Guidance* Sensors. Die Lagewinkel Φ und Θ der X- und Y-Achse basieren hierbei auf den Messwerten der Beschleunigungssensoren der integrierten IMU. Da die Winkellagen aus den gemessenen Werten der Erdbeschleunigung errechnet werden unterliegen diese keinem Trendfehler. Zusätzlich wird die aktuelle Höhe aus den Daten des vertikal messenden Sensors auf Basis der Stereokamera und des Ultraschallsensors gerechnet. Somit können weiter Vereinfachungen auf Grund der Vororientierung der Punktwolke für die Parameter

- Translation entlang der Z-Achse (Höhe) T_z
- Rotation um die X- und Y-Achse Φ und Θ

vorgenommen werden. Es entfallen die zugehörigen Transformationsparameter aus Formel 43. Das Gleichungssystem kann zu

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \mathbf{T} + \mathbf{R}(\Psi) \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{44}$$

vereinfacht werden. In Folge ist eine Vereinfachung des Transformationsmodelles auf eine Transformation im $\mathbb{R}2$ Raum und somit auf einen linearen Gleichungsansatz möglich. Die somit verbleibenden Transformationsparameter bilden

- die Translation entlang der X- und Y-Achse T_x und T_y, sowie
- die Rotation um die Hochachse Ψ.

Nach Lösen des Gleichungssystems stehen die Transformationsparameter für die Transformation der Punktwolke aus dem Ausgangssystem in das Zielsystem zur Verfügung. Mit Hilfe der Methode *Tranform* werden die Transformationsparameter bestimmt und die Punktwolke sowie der Koordinatenursprung transformiert.
Nach der Transformation wird erneut die Punktwolke im Zielsystem bestimmt und das kleinste Fehlerquadrat berechnet. Die Abbruchbedingung für die Iteration kann jetzt geprüft werden. Listing 25 zeigt den Algorithmus als C++ Quellcode.

```
1  tf::Pose Fitt(pcl::PCLPointCloud2 scan)
2  {
3      pcl::PCLPointCloud2 sourceCloud = scan;
4      // Punktwolke im Zielsystem ermitteln
5      pcl::PCLPointCloud2 targetCloud GetTargetPointCloud(sourceCloud);;
6      // Varianz bestimmen
7      double epsylon = GetVariance(sourceCloud, targetCloud);
8      double lastEpsylon;
9      int maxIteration = 100;
10     int itCount = 0;
11     // Ursprung des des Ausgangssystems
12     tf::Pose transformedPose = lastPosition;
13     do
14     {
15         double lastEpsylon = epsylon;
16         // Transformation
17         tf::Pose newPose = transformedPose;
18         pcl::PCLPointCloud2 transformedCloud = Transform(sourceCloud,
               targetCloud, &newPose);
19         // Bestimmen der naechsten Zielpunktwolke
20         targetCloud = GetTargetPointCloud(transformedCloud);
21         // Bestimmen der Varianz
22         epsylon = GetVariance(transformedCloud, targetCloud);
23         itCount++;
24         // Fehler groesser? -> letztes Fitting nehmen
```

```
25          if (epsylon < lastEpsylon)
26          {
27              // wenn nicht, transformierte Punktwolke und Pose
                    uebernehmen
28              sourceCloud = transformedCloud;
29              transformedPose = newPose;
30          }
31      } while (epsylon <= lastEpsylon && epsylon >= minEpsylon &&
            itCount <= maxIterations);
32
33      // transformierte Position zurueckgeben
34      return transformedPose;
35  }
```

Listing 25: C++ Implementierung des ICP Algorithmus

Ist das Minima der Verbesserung erreicht liefert die Methode die transformierte Position als Ergebnis zurück. Der Node kann jetzt die korrigierte Position als *fittPose* wie in Listing 24 veröffentlichen.

10.5. Übergabe der Position und Orientierung

Das vom Hersteller *DJI* zur Verfügung gestellte *Onboard SDK* bietet keine Möglichkeit eine Position und Ausrichtung alternativ zum GNSS-Kompass Modul zu übergeben. Aus diesem Grund müssen alternative Wege gesucht werden.
Eine Möglichkeit stellt die Adaption des verbauten GNSS-Kompass Moduls dar. Hiebei wird das GNSS- und Kompass Signal auf dem CAN Bus durch einen eigenen CAN-Node ersetzt. Auf dem UAV ist das *DJI GPS-Compass Pro Plus* Modul, Abb. 44, verbaut und mittels CAN-Bus in das System angeschlossen. Das Modul wird vom

Abbildung 44: DJI GPS-Compass Pro Plus Modul[44]

System getrennt und für die CAN-Bus Anbindung ein Software Node etabliert. Da das

[44] Quelle: https://www.dji.com/newsroom/news/introducing-the-new-a2-gps-pro-plus-module

verwendete Rechnersystem *AscTec Mastermind* über kein CAN-Bus Interface verfügt muss hier ein geeigneter USB-CAN Adapter verwendet werden. Die Abb. 45 zeigt einen USB-CAN Adapter des Herstellers *Zubax*[45] sowie des Herstellers *Lawicel*[46]. Mit Hilfe

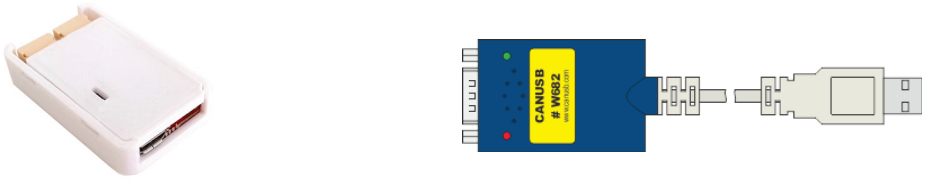

Abbildung 45: CAN-USB Adapter *babel* der Firma *Zubax* und *CANUSB W682* der Firma Lawicel

des CAN-USB Adapters ist es möglich einen freien USB-Port des *AscTex Mastermind* an den GPS-CAN Ansschluss des UAV zu schalten. Entsprechende Treiber werden von den Herstellern zur Verfügung gestellt.
Für die Übertragung der Positions- und Orientierungsdaten ist somit die Softwareimplementierung eines CAN Node erforderlich. Durch den Hersteller *DJI* werden keine Informationen über das unterstützte Protokoll sowie den Kommunikationsablauf auf dem CAN-Bus zur Verfügung gestellt. Auf der Basis unterschiedlicher Threads[47] in diversen Foren wird davon ausgegangen, dass der *DJI N1 Flight Controller* die GNSS Nachrichten auf der Message ID 0x7F8 sowie die Kompass-Daten auf der Message ID 0x118 im Litle Endian Format auf dem CAN Bus empfängt. Das Message-Format der jeweiligen Datenpakete ist für die GNSS Daten[48] wie folgt beschrieben.

[45] Quelle: `https://zubax.com/products/babel`
[46] Quelle: `http://www.can232.com/?page_id=16`
[47] Quelle: `https://www.rcgroups.com/forums/showthread.php?2611607-DJI-M100-N1-CAN-Bus-Communication-Protocol`
[48] Quelle: `https://www.rcgroups.com/forums/showpost.php?p=26210591&postcount=15`

```
1  -----------
2  BYTE 1-2: message header - always 55 AA
3  BYTE 3: message id (0x10 for GPS message)
4  BYTE 4: lenght of the payload (0x3A or 58 decimal for 0x10 message)
5
6  PAYLOAD
7  --------------
8  BYTE 5-8 (DT): date and time, see details below
9  BYTE 9-12 (LO): longitude (x10^7, degree decimal)
10 BYTE 13-16 (LA): latitude (x10^7, degree decimal)
11 BYTE 17-20 (AL): altitude (in milimeters)
12 BYTE 21-24 (HA): horizontal accuracy estimate (see uBlox NAV-POSLLH
       message for details)
13 BYTE 25-28 (VA): vertical accuracy estimate (see uBlox NAV-POSLLH
       message for details)
14 BYTE 29-32: ??? (seems to be always 0)
15 BYTE 33-36 (NV): NED north velocity (see uBlox NAV-VELNED message for
       details)
16 BYTE 37-40 (EV): NED east velocity (see uBlox NAV-VELNED message for
       details)
17 BYTE 41-44 (DV): NED down velocity (see uBlox NAV-VELNED message for
       details)
18 BYTE 45-46 (PD): position DOP (see uBlox NAV-DOP message for details)
19 BYTE 47-48 (VD): vertical DOP (see uBlox NAV-DOP message for details)
20 BYTE 49-50 (ND): northing DOP (see uBlox NAV-DOP message for details)
21 BYTE 51-52 (ED): easting DOP (see uBlox NAV-DOP message for details)
22 BYTE 53 (NS): number of satellites (not XORed)
23 BYTE 54: ??? (not XORed, seems to be always 0)
24 BYTE 55 (FT): fix type (0 - no lock, 2 - 2D lock, 3 - 3D lock, not
       sure if other values can be expected - see uBlox NAV-SOL message
       for details)
25 BYTE 56: ??? (seems to be always 0)
26 BYTE 57 (SF): fix status flags (see uBlox NAV-SOL message for details)
27 BYTE 58-59: ??? (seems to be always 0)
28 BYTE 60 (XM): not sure yet, but I use it as the XOR mask
29 BYTE 61-62 (SN): sequence number (not XORed), once there is a lock -
       increases with every message. When the lock is lost later LSB and
       MSB are swapped with every message.
30
31 CHECKSUM
32 -----------------
33 BYTE 63-64 (CS): checksum, calculated the same way as for uBlox binary
        messages
```

Listing 26: CAN-Message-Format der GNSS Daten

Das Message-Format der jeweiligen Datenpakete ist für die Kompass Daten[49] wie folgt beschrieben.

```
HEADER
-------------
BYTE 1-2: message header - always 55 AA
BYTE 3: message id (0x20 for compass message)
BYTE 4: length of the payload (0x06 or 6 decimal for 0x20 message)

PAYLOAD
--------------
BYTE 5-6 (CX): compass X axis data (signed) - see comments below
BYTE 7-8 (CY): compass Y axis data (signed) - see comments below
BYTE 9-10 (CZ): compass Z axis data (signed) - see comments below

CHECKSUM
-----------------
BYTE 11-12 (CS): checksum, calculated the same way as for uBlox binary
    messages
```

Listing 27: Installation der ROS Distribution

Jedoch weichen bei beiden Formatbeschreibungen die Header von den tatsächlichen Headerdaten in Länge und Codierung ab.
Für die Implementierung des eigenen CAN-Node steht die Nutzung der durch die Hersteller der USB-CAN Adapter zur Verfügung gestellten Softwarebibliotheken zur Verfügung. Hierbei kann jedoch nur auf die Grundfunktionalitäten der CAN-Protokolle zurück gegriffen werden. Eine korrekte Codierung der Datenstrukturen muss in einer eigenen Programmierung implementiert werden.
Für die Softwareimplementierung von CAN-Nodes im UAV-Bereich hat das Projekt *UAVCAN*[50] weite Verbreitung gefunden. Mehrere freie Flight Controller im UAV Bereich unterstützen oder basieren auf *UAVCAN*. *UAVCAN* spezifiziert alle notwendigen Layer für den Datentransport, Low Level Anwendungsfunktionen sowie grundlegende Datenstrukturen für die Kommunikation von Flug- und Roboteranwendungen auf dem CAN-Bus. Als SDK werden die Bibliotheken:

- *Libuavcan* in C++ für Embedded Systems und Linux,
- *Pyuavcan* als CAN protocol stack und DSDL parser Implementierung in Python sowie
- *Libcanard* als minimale C Implementierung für Embedded Systems und Ressourcen sparende Anwendungen

[49] Quelle: `https://www.rcgroups.com/forums/showpost.php?p=26248426&postcount=62`
[50] https://uavcan.org/

zur Verfügung gestellt. Eine Anbindung auf der Basis dieser Bibliothek ist somit ebenfalls möglich.

Teil III.
Auswertung

11. Zusammenfassung

Ziel der vorliegenden Arbeit war es, einen Hard- und Software Lösungsansatz zu entwickeln, der es einem UAV System ermöglicht innerhalb geschlossener Räume zu navigieren. Für die Navigationslösung soll ein 3D Modell der räumlichen Umgebung verwendet werden. Alle notwendigen Komponenten der Sensorik sowie der verwendeten Rechensysteme sollten auf dem UAV System verbaut sein. Die Einbindung externer Ressourcen sollte ausgeschlossen werden. In der Einleitung zu dieser Arbeit wurde die Zielstellung detailliert beschrieben, die vorhandenen Ausgangsbedingungen in Bezug auf die vorhandene Hardware dargelegt sowie die Zielstellung gegenüber ähnlich gelagerten Lösungsansätzen abgegrenzt.
In den Grundlagen wurde auf die zum derzeitigen Stand gebräuchlichen Typen von UAV-Systemen eingegangen sowie deren Eignung für die vorliegende Aufgabenstellung bewertet. Zusätzlich wurde ein Überblick über in Luftfahrtsystemen verwendeter Sensorik, deren Eignung für UAV Systeme sowie der Entwicklung speziell für UAV geeignete Sensorik gegeben. Zusätzlich wurde ein Überblick über geeignete 3D-Datenformate für die Abbildung der für die Navigation relevanten Umgebung gegeben.
Im Abschnitt Umsetzung wurde der Hardwareaufbau des Testsystems für das UAV, die Sensorik sowie die verwendete Rechnereinheit und deren technische Daten beschrieben. Die Sensorik und Rechnereinheit wurde wie gefordert auf dem UAV verbaut. Auf der Basis des zu verwendenden Hardwareaufbaus wurde ein Lösungsansatz für die Realisierung der Aufgabenstellung erarbeitet. Zusätzlich zur theoretischen Lösung wurde ein Lösungsansatz für die Softwareimplementierung gesucht. Sowohl die Softwareeinbindung der verwendeten Sensorik wie auch die Realisierung der Navigationslösung wurde in diesem Abschnitt beschrieben. Für die Datenkommunikation wurde auf das *Robot Operation System*, eine in der Robotik weit verbreitete Implementierung zurück gegriffen.
Für die zur Verfügung stehenden Laserscanner liegen Implementierungen als ROS-Node von Seiten der Hersteller vor. Diese konnten eingebunden und getestet werden. Beide Scanner sind somit für die Implementierung geeignet. Auf Grund der größeren Robustheit sowie der kürzer Datenerfassungszeit für einen vollen Scann liegt die Empfehlung beim Scanner *UST-20LX*. Die Tests erfolgten jedoch mit dem Scanner *Sweep V1.0*. Somit konnte gezeigt werden, dass eine Lösung auch mit Scannern mit geringerer Datenmenge sowie geringeren Frame Rate erreicht werden kann.
Für den Sensor *Guidance* liegt ebenfalls eine ROS-Node von Seiten des Herstellers vor.

Die Implementierung erfolgt über die Anbindung an die vorhandene USB-Schnittstelle des Sensors unter Verwendung der *libusb* Softwarebibliothek in der vom Hersteller *DJI* zur Verfügung gestellten Library. Ein Verbindungsaufbau über diese Implementierung war auf dem verwendeten Rechnersystem *AscTec Mastermind* jedoch nicht erfolgreich. Die Gründe hierfür konnten nicht abschließend ermittelt werden. Aus diesem Grund wurde ein eigener ROS Node für die Anbindung des Sensors über die serielle Schnittstelle entwickelt. Mit Hilfe des Node konnte erfolgreich auf die Sensordaten zugegriffen werden. Da im Rahmen der vorliegenden Arbeit kein Zugriff auf die Daten der Kamerasensoren des Sensors *Guidance* geplant waren, ist die gewählte Anbindung über die vorhandene serielle Schnittstelle ausreichend.
Das verwendete Rechnersystem *AscTec Mastermind* wurde vom Hersteller speziell für den Einsatz auf UAV entwickelt. Da bereits ein Linux sowie eine ROS Distribution vorinstalliert ist konnte ohne größeren Aufwand mit der Softwareimplementierung begonnen werden. Es hat sich jedoch herausgestellt, dass die installierte Linux Ubuntu Version nicht über die Standard Update Funktionalität aktualisiert werden kann. Gleiches trifft für das installierte ROS *Indigo* zu. Auf Grund der spezifischen Hardwarekonstellation des Rechnersystems wurde auf ein manuelles Update für das Betriebssystem sowie die ROS Distribution verzichtet. Der dafür erforderliche Zeitaufwand ist als nicht überschaubar sowie die Einbindung aller notwendigen Hardwarekomponenten als nicht sicher eingeschätzt worden. Mit dem eingesetzten *Intel I7* steht ein leistungsfähiger Prozessortyp zur Verfügung. Die Laufzeit kritischen Softwareimplementierungen sind vorwiegend im Bereich der Koordinaten bezogenen Datenverarbeitung zu finden. In Auswertung der aktuellen Arbeit wird hier ein Wechsel auf ein *NVIDIA* System vom Typ *TX2* oder nachfolgend empfohlen. Vorteile sind hier auf Grund der höheren Anzahl an Prozessorkernen für die Parallelisierung der Datenverarbeitung im Bereich der Koordinatentransformation sowie des Occlusion Culling zu erwarten. Durch die Verwendung der Programmiersprache *Cuda* sind hier hohe Steigerung in der Laufzeitoptimierung zu erwarten.
Durch die Vororientierung der Scann Punktwolke auf der Basis der Positions- und Orientierungdaten des Sensors *Guidance* konnte die Transformationsgleichung erheblich minimiert werden. Zusätzlich konnte durch diese Maßnahme auf eine Schätzung der Transformationsparameter vor der Ausgleichung verzichtet werden. Auf Basis der kontinuierlichen Verbesserung der gerechneten Positions- und Ausrichtungsdaten mit Hilfe des ICP-Algorithmus wurden diese fortlaufend mit einem minimalen Fehler gehalten. Dies führt in Folge dazu, dass der Fehler der Vororientierung der Scann-Daten minimal bleibt und somit zu einer geringe Iteratationstiefe des Algorithmus führt. Die Wahl des 3D Datenformates als Wave Front Format hat sich als richtig erwiesen. Das Format ist leicht implementierbar. Die geometrischen Objektdaten stehen in einem für den Algorithmus gut verwertbaren Format zur Verfügung. Die enthaltenen Materialdateien stellen eine gute Möglichkeit für eine zukünftigen Implementierung von Metadaten dar.

Hierauf wurde im Rahmen dieser Arbeit jedoch verzichtet.
In den praktischen Tests hat sich gezeigt, dass die durch den Sensor *Guidance* gelieferten Höhe einem Trendfehler unterliegt. Hierzu sollten weitere Untersuchungen angestellt werden. Im Zweifel kann der translative Parameter der Z-Achse nicht aus der Transformation entfernt werden oder es müssen andere Wege zur Sicherstellung eines korrekten Höhenwertes gefunden werden.
Der Lösungsansatz sah vor, dass die Navigation mit einer gültigen Startposition und Orientierung beginnt. Dieser Ansatz hat sich als richtig erwiesen. Durch die Vorgabe gültiger Startwerte kann eine sichere Position und Ausrichtung garantiert werden. Unsichere Navigationslösungen durch lokale Minima in der Ausgleichung außerhalb des globalen Minima werden vermieden. Dieses Konzept ist auch in anderen Lösungsansätzen in ROS-Navigationslösungen zu finden.
Die ermittelten Positions- und Orientierungsdaten mussten in geeigneter Form an den Autopiloten des UAV übergeben werden. Die gewählte Lösung der Übergabe auf dem CAN-Bus alternativ zum GPS-Kompass Sensor bringt den Vorteil mit sich, dass keine Eingriffe in die Software des Autopiloten notwendig werden. Durch die geschlossene Programmierung des Flug-Controllers *DJI N1 Flight Controller* ist eine Modifikation der Firmware ausgeschlossen. Die Möglichkeit der Übergabe der Daten durch das Onboard SDK ist durch den Hersteller nicht vorgesehen. Somit stellt diese Lösung in der aktuell verwendeten Hardwarekonstellation die einzige Möglichkeit der Adaption dar.
Die Anbindung an den CAN-Bus ist durch geeignete Hardware Adapter problemlos möglich. Schwieriger gestaltet sich jedoch die Softwareumsetzung der Aufgabe. Da keine konkrete Protokollbeschreibung der Datenkommunikation auf dem CAN-Bus zur Verfügung steht müssen im Vorfeld umfangreiche Traces auf dem CAN Bus zur Bestimmung der benötigten Header und Datencodierung der erforderlichen Telegramme vorgenommen werden.
Mit der vorliegenden Arbeit konnte gezeigt werden, dass eine 3D-Navigation ohne GNSS- und Kompassdaten auf der Basis vorhandener 3D-Modelle möglich ist. Durch intelligente Nutzung von Sensordaten kann die Komplexität des Ausgleichungsproblemes erheblich minimiert werden. Dies führt zu einem erheblichen Performance Gewinn sowie einer sicheren Navigationslösung. Durch die Nutzung bestehender Implementierungen für Robot Systeme wie dem ROS ist eine schnelle und sicher Implementierung möglich. Dies verkürzt die Entwicklungszeit für solche Anwendungen erheblich.

12. zukünftige Arbeiten

Für die Realisierung einer funktionsfähigen technischen Lösung eines autonom navigierenden Systems sind weitere Arbeiten notwendig. Dies umfasst die Implementierung notwendiger Funktionalitäten die in dieser Arbeit nicht behandelt wurden. Dazu zählen die

- Verbesserung der Höhenmessung,
- Alternative Erfassung von räumlichen Daten in der Z-Ebene durch Implementierung mehrkanaliger Laser oder Tiefenbilder der Kamerasysteme,
- endgültige Implementierung des CAN Node für die Übertragung der Position und Ausrichtung an den Autopiloten,
- Implementierung einer Datenverbindung für das Monitoring,
- Erstellen einer Simulation für notwendige Test Szenarien mittels *Gazebo* im ROS System.

Auf Grund der nicht zuverlässigen Höheninformation wird eine Überarbeitung empfohlen. Alternativ besteht die Möglichkeit den Parameter der Translation in Z-Richtung wieder in die Transformation aufzunehmen. Hierfür sollte jedoch der räumliche Scann auf die Z-Achse erweitert werden. Dies kann durch die Verwendung von mehrkanaligen Laserscannern oder durch geeignete Kamerasysteme realisiert werden. Der *Guidance* Sensor liefert bereits Tiefenbilder seiner Stereokameras auf dem USB-Port. Diese können zur Bestimmung weiterer räumlicher Koordinatenwerte mit einer Ausdehnung in Z-Richtung verwendet werden. Es wird jedoch empfohlen, im Vorfeld auch hier Berechnungen und Tests zur Ermittlung der Messgenauigkeit vorzunehmen. Der Hersteller *DJI* liefert auch hierzu keine Angaben wobei der geringe Kameraabstand einer hohen Messgenauigkeit entgegen spricht.
Für die Umsetzung eines autonomen Fluges mittels Navigation ist die Implementierung des CAN Node für die Übertragung der Position und Orientierung an den Autopiloten zwingend erforderlich. Hierzu müssen umfangreiche Messungen zur Bestimmung der Message-Protokolle mit dem aktuell vorhandenen *DJI GPS-Compass Pro Plus* Modul vorgenommen werden. Nach Implementierung des CAN Node kann dieser das vorhandene Modul ersetzen. Die vorhandene Routenplanung kann dann mit den Koordinaten des verwendeten 3D-Modell erfolgen.
Für eine Planung und Visualisierung des Fluges inklusive der Datenübertragung von Messdaten ist die Umsetzung einer Datenkommunikation als eigenständiges Softwaremodul erforderlich. Dies eröffnet die Möglichkeit einer Planungs- und Visualisierungslösung außerhalb der Herstellerlösungen.
Das ROS System stellt diverse Werkzeuge für die Simulation und Visualisierung wie die Software-Tools *Gazebo*, *rviz* oder *rqt* zur Verfügung. Für die weiteren Arbeiten zur Verbesserung der Algorithmen sowie der Verbesserung der Parametrierung ist eine Systemsimulation mittels *Gazebo* zu erstellen. Das *DJI ROS SDK* liefert bereits eine *Gazebo* Implementierung für die Simulation einer *Matrice 100*. Diese muss um ein eigenes Modell des Sensors *Guidance* erweitert werden.

Literaturverzeichnis

[AMS09] AMSYS: *Wissenswertes zur Absolutdruckmessung mit piezoresistiven Messzellen.* AMSYS GmbH & Co. KG, 2009

[AMS17] AMSYS: *Datasheet MS5611-01BA03 Barometric Pressure Sensor, with stainless steel cap.* AMSYS GmbH & Co. KG, 2017

[Ans18] Anschütz, Raytheon: *Produkt Datenblatt Standard 22 Compact Gyro Compass Retrofit Solution.* Raytheon Anschütz GmbH, 2018

[BAL11] Brockhaus, Rudolf ; Alles, Wolfgang ; Luckner, Robert: *Flugregelung.* 3., neu bearbeitete Auflage. Springer Verlag, 2011

[BB17] *Kapitel* Luftverkehrs-Ordnung (LuftVO) 21b Verbotener Betrieb von unbemannten Luftfahrtsystemen und Flugmodellen. In: BMVI ; BMU: Bundesanzeiger Verlag, 2017

[Bos14] Bosch: *BMI055 Small, versatile 6DoF sensor module.* Bosch, 2014

[Cas19] Castro, Felipe: *LibDWG free access to DWG.* `https://libdwg.sourceforge.io/en/index.html`. Version: 2019. – (letzter Zugriff: 27.12.2019)

[DJI16] DJI: *DJI Matrice 100 User Manual.* DJI, 2016

[DN90] DIN-Normenausschuss: *Luft- und Raumfahrt; Begriffe, GrösSen und Formelzeichen der Flugmechanik; Bewegung des Luftfahrzeuges gegenüber der Luft; ISO 1151-1:1988 modifiziert.* Beuth Verlag, 1990

[Ene18a] Energy, Intelligent: *Datasheet AC64 Lightweight Stack.* Intelligent Energie, 2018

[Ene18b] Energy, Intelligent: *UAV Fuel Cell Power Module.* Intelligent Energie, 2018

[Flü12] Flühr, Holger: *Avionik und Flugsicherungstechnik.* 2. Auflage. Springer Verlag, 2012

[HLN12] Hertzberg, Joachim ; Lingemann, Kai ; Nüchter, Andreas: *Mobile Roboter.* Springer Verlag, 2012

[HWLW03] Hofmann-Wellendorf, Bernhard ; Legat, Klaus ; Wieser, Manfred: *Navigation.* Springer-Verlag, 2003

[ICA93] ICAO: *Manual of the ICAO Standard Atmosphere.* International Civil Aviation Organisation, 1993

[ICA11] ICAO: *Cir 328 AN/190 Unmanned Aircraft Systems (UAS), S. 3, Absatz 2.4 MODEL AIRCRAFT.* International Civil Aviation Organization, 2011. – ISBN 978–92–9231–751–5

[Ise15] Isentek: *IST8319 3D Magnetometer Brief Datasheet.* 1.2. Isentec Technologies, 2015

[Jan06] Janisch, Josef: Was Sie schon immer über Hallsensoren wissen wollten - Kleiner Effekt - Große Wirkung. In: *elektronik industrie* 7 (2006)

[Kni18] Knickmeyer, Elfriede: *Einführung in die Navigation*. Hochschule Neubrandenburg, 2018

[Law93] Lawrence, Anthony: *Modern Inertial Technology*. Springer-Verlag, 1993

[LLC17] LLC, Scanse: *Userś Manual and Technical Specification, sweep v1.0 scanning laser range finder*. Scanse LLC, 2017

[Mak15] Maksymiw, Alexander: *AscTec Mastermind*. `http://wiki.asctec.de/display/AR/AscTec+Mastermind`. Version: 2015. – (letzter Zugriff: 23.08.2018)

[Mer96] Merhav, Samuel: *Inertial Sensor Systems and Applications*. Springer-Verlag, 1996

[NFHS11] Nischwitz, Alfred ; Fischer, Max ; Haberäcker, Peter ; Socher, Gudrun: *Computergrafik und Bildverarbeitung*. Bd. Band 1 Computergrafik. 3. Auflage. Vieweg+Teubner Verlag, 2011

[Nie01] Niemeier, Wolfgang: *Ausgleichsrechnung*. Walter de Gruyter GmbH & Co. KG, 2001

[ODA18] Open Design Alliance, Inc.: *Open Design Specification for .dwg files*. Version 5.3. Open Design Alliance, Inc., 2018

[TDK18] TDK: *ICM-20689 High Performance 6-Axis MEMS Motion Tracking Device in 4x4 mm Package*. TDK Corporation, 2018

[Wen11] Wendel, Jan: *Integrierte Navigationssysteme*. Oldenburg Verlag München, 2011

[Win17] Winter: *Einbau und Wartungsanweisung für Höhenmesser 4FG10 und 4HM6*. Gebr. Winter GmbH & Co. KG, 2017

[Zwe97] Zweck, Axel: *Technologieanalyse Magnetismus Band 2 - XMR-Technologien*. VDI Technologiezentrum GmbH, 1997

Selbstständigkeitserklärung

Ich versichere, die von mir vorgelegte Arbeit „Entwicklung einer Sensorfusion basierten, UAV gestützten Navigationslösung in geschlossenen baulichen Strukturen unter Verwendung von 3D-Raummodellen“ selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift$: $Ort, Datum$: