

# Entwurf und Implementierung einer Daten-Schnittstelle zum Betrieb eines Laserscanner VLP-16 an einem Raspberry Pi

Bachelorthesis

vorgelegt von:  
Florian Timm

Mittwoch, den 13. Dezember 2017

**Verfasser**

Florian Timm

Matrikelnummer: 6028121

Gaiserstraße 2, 21073 Hamburg

E-Mail: florian.timm@hcu-hamburg.de

**Erstprüfer**

Prof. Dr. rer. nat. Thomas Schramm

HafenCity Universität Hamburg

Überseeallee 16, 20457 Hamburg

E-Mail: thomas.schramm@hcu-hamburg.de

**Zweitprüfer**

Dipl.-Ing. Carlos Acevedo Pardo

HafenCity Universität Hamburg

Überseeallee 16, 20457 Hamburg

E-Mail: carlos.acevedo@hcu-hamburg.de

## Kurzzusammenfassung

Die vorliegende Arbeit ist Teil eines Projektes, dass die Entwicklung eines Systems zum Ziel hat, welches den modular austauschbaren Betrieb verschiedenster Sensorsysteme an einem Multikopter erlauben soll. Im Speziellen soll hier die Datenschnittstelle von einem Kompakt-Laserscanner Velodyne Lidar Puck VLP-16 zu einem Einplatinencomputer Raspberry Pi entwickelt und implementiert werden. Der Scanner selbst liefert hierbei die Daten in einem proprietären, binären Format, welche in ein einfache lesbares Format, hier eine ASCII-Datei, umgewandelt und gespeichert werden sollen. Außerdem sollen die Daten mit einem eindeutigen Zeitstempel versehen werden, um diese später mit anderen Sensorsystemen verknüpfen zu können. Diese Datentransformation sollte möglichst simultan zur Aufnahme erfolgen.

Auch Teil der Arbeit ist die Schaffung einer Steuerung der Aufnahme des Laserscanners. Hierfür wurde ein Bedienmodul entwickelt, welches am Raspberry Pi direkt angeschlossen werden kann, sowie eine Steuerungsweboberfläche eingebunden, die die Steuerung während des Fluges ermöglichen soll.

## Abstract

The present work is part of a project aimed the development of a system that allows the modular interchangeable operation of various sensor systems on a multicopter. In particular, the data interface for compact laser scanner Velodyne Lidar Puck VLP-16 to a single-board computer Raspberry Pi will be developed and implemented. The scanner itself provides the data in a proprietary, binary format, which should be converted and stored into an easy-to-read ASCII file. In addition, the data should be provided with a unique timestamp in order to be able to link it later with other sensor systems. This data transformation should be carried out as simultaneously as possible while recording.

Also part of the work is the creation of a control of the recording of the laser scanner. For this purpose, an operating module was developed, which can be connected directly to the Raspberry Pi, as well as a web control surface integrated in the software, which should enable the control during the flight.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Problemstellung . . . . .	1
1.2 Zielsetzung . . . . .	1
1.3 Struktur . . . . .	2
<b>2 Grundlagen des Airborne Laserscanning</b>	<b>3</b>
2.1 Laserscanner . . . . .	3
2.1.1 Entfernungsmessung . . . . .	3
2.1.2 Ablenkeinheit . . . . .	5
2.1.3 Oberflächeneffekte . . . . .	6
2.2 Positionsbestimmung mittels globalen Navigationssatellitensystemen . .	6
2.3 Inertiale Messeinheit . . . . .	7
2.4 Kombination des Messsystems . . . . .	8
2.5 Bisherige Systeme zur dreidimensionalen Erfassung mittels Multikoptern	9
<b>3 Technische Realisierung</b>	<b>10</b>
3.1 Verwendete Gerätschaften . . . . .	10
3.1.1 Velodyne VLP-16 . . . . .	10
3.1.2 Inertiale Messeinheit und GNSS-Empfänger iMAR iNAT-M200-FLAT . . . . .	11
3.1.3 Raspberry Pi 3 Typ B . . . . .	12
3.1.4 Multikopter Copterproject CineStar 6HL . . . . .	13
3.1.5 Gimbal Freefly MöVI M5 . . . . .	14
3.2 Auswahl des Datenverarbeitungssystems . . . . .	15
3.3 Stromversorgung . . . . .	16
3.4 Anbindung des Raspberry Pi an den Laserscanner . . . . .	16
3.5 Verbindung des GNSS-Moduls zum Laserscanner . . . . .	17
3.6 Steuerung im Betrieb . . . . .	18
3.7 Platinenentwurf und -realisierung . . . . .	20
<b>4 Theoretische Datenverarbeitung</b>	<b>21</b>
4.1 Verwendung von Python . . . . .	21

4.2	Datenlieferung vom Laserscanner . . . . .	21
4.3	Geplantes Datenmodell . . . . .	22
4.4	Weiterverarbeitung der Daten zu Koordinaten . . . . .	23
4.5	Anforderungen an das Skript . . . . .	24
<b>5</b>	<b>Entwicklung des Skriptes</b>	<b>27</b>
5.1	Klassenentwurf . . . . .	27
5.2	Evaluation einzelner Methoden . . . . .	27
5.3	Multikern-Verarbeitung der Daten . . . . .	29
5.4	Klassen . . . . .	30
5.4.1	VdAutoStart . . . . .	30
5.4.2	VdInterface . . . . .	31
5.4.3	VdHardware . . . . .	32
5.5	Beispiel-Quelltext-Zitat . . . . .	32
<b>6</b>	<b>Konfiguration des Raspberry Pi</b>	<b>33</b>
6.1	Installation von Raspbian . . . . .	33
6.2	Befehle mit Root-Rechten . . . . .	34
6.3	IP-Adressen-Konfiguration . . . . .	34
6.4	Konfiguration als WLAN-Access-Point . . . . .	35
6.5	Autostart des Skriptes . . . . .	36
<b>7</b>	<b>Ausblick</b>	<b>37</b>
<b>Literaturverzeichnis</b>		<b>38</b>
<b>Abbildungsverzeichnis</b>		<b>40</b>
<b>Tabellenverzeichnis</b>		<b>41</b>
<b>Anhang</b>		<b>42</b>
<b>A</b>	<b>Python-Skripte</b>	<b>43</b>
A.1	vdAutoStart.py . . . . .	43
A.2	vdBuffer.py . . . . .	46
A.3	vdTransformer.py . . . . .	47
A.4	vdInterface.py . . . . .	49
A.5	vdHardware.py . . . . .	50
A.6	vdFile.py . . . . .	52
A.7	vdDataset.py . . . . .	53
A.8	vdPoint.py . . . . .	56
A.9	vdConfig.py . . . . .	56

A.10 convTxt2Obj.py . . . . .	57
<b>B Beispieldateien</b>	<b>59</b>
B.1 Rohdaten vom Scanner . . . . .	59
B.2 Dateiformat für Datenspeicherung als Text . . . . .	59
B.3 Dateiformat für Datenspeicherung als OBJ . . . . .	59

# 1 Einleitung

## 1.1 Problemstellung

Daten aus Airborne Laserscanning, dem Abtasten von Oberflächen mit einem Laserscanner aus der Luft, lassen sich für viele verschiedene Zwecke benutzen. Oft werden sie zur Erfassung von digitalen Geländemodellen verwendet, aber auch für die Erstellung von Stadtmodellen oder Vegetationsanalysen sind die Daten nutzbar. Aktuell werden als Trägersysteme des Laserscanner Helikopter oder Flugzeuge verwendet, die mit entsprechender Sensorik ausgerüstet sind. Diese Messmethode lohnt sich allerdings nicht für kleiner Gebiete und ist auch aufgrund der Größe und die Gefahren des Fluggerätes nicht für die Aufnahme feiner Strukturen wie Fassaden geeignet, bei denen zwischen Häuserschluchten geflogen werden müsste. Außerdem sind die Betriebs- und Anschaffungskosten sind sehr hoch, so dass sich eine solche Messung oft nur für sehr große Gebiete lohnt. Alternativ bietet sich die terrestrische Messungen mittels Tachymeter oder auch per Laserscanner an um kleinere Gebiete abzubilden – hier benötigt die Aufnahme jedoch viel Zeit und Personal. Hinzukommt, dass die Genauigkeit für viele Anwendungsfälle der 3D-Modelle zu hoch ist. Beide Möglichkeiten, die Messung aus der Luft oder vom Boden, sind sehr kostenintensiv. Ein Lösungsansatz hierfür wäre es, anstatt eines Helikopters als Trägersystem, einen Multikopter zu nutzen. Jedoch ist die Tragfähigkeit für die meisten Laserscanning-Systeme nicht ausreichend. Daher basieren 3D-Erfassungssysteme mittels Multikoptern heutzutage meist auf photogrammetrischen Prinzipien, welche Luftbilder zur nutzen. Hierzu muss jedoch ausreichend Beleuchtung vorhanden sein, welches wiederum die Einsetzbarkeit des Systemes in Städten beschränkt, in denen nur nachts für ausreichende Sicherheitszonen zum Betrieb von Multikoptern gesorgt werden kann.

## 1.2 Zielsetzung

Gesamtziel ist es ein Laserscanning-System zu entwickeln, dass von einem Multikopter getragen werden kann. Hierbei soll vor allem auf ein geringes Gewicht geachtet, aber auch die Kosten niedrig gehalten werden. Im Speziellen soll hier als erster Schritt die Datenverarbeitung des Laserscanners in einem solchen System realisiert werden. Hierfür soll ein Ein-Platinen-Computer Typ Raspberry Pi 3 die Speicherung und Aufbereitung

der von einem Laserscanner Velodyne Puck VLP-16 aufgezeichneten Laserpunktdata übernehmen. Hierfür müssen entsprechende Schnittstellen zum Verbinden der Geräte in Hard- und Software entwickelt werden.

## 1.3 Struktur

Im Kapitel 2 sollen die Grundlagen des luftgestützten Laserscanning erläutert werden. Außerdem wird die benötigte Hardware zur Durchführung eines solchen Laserscanning besprochen. Im folgenden wird näher auf die Realisierung des Projektes eingegangen: Welche Hardware wurde verwendet und wie wurde Sie angeschlossen (Kapitel 3), wie sollen die Daten verarbeitet werden (Kapitel 4) und wie wird die Verarbeitung schließlich durchgeführt (Kapitel 5) und das System konfiguriert (Kapitel 6). Zum Abschluss soll in Kapitel 7 noch ein Einblick in die Zukunft des Systemes geworfen werden.

## 2 Grundlagen des Airborne Laserscanning

Airborne Laserscanning bezeichnet das Verfahren, bei dem ein Laserscanner, welcher an einem Fluggerät befestigt ist, Oberflächen kontaktlos dreidimensional erfasst (Beraldin et al., 2010, S. 1). Der Laserscanner liefert hierbei Daten in Form von der Abstrahlrichtung des Strahles und der Entfernung, relativ zu seiner eigenen Ausrichtung und Position. Um diese lokalen Daten in ein globales System zu überführen, werden zusätzlich die Ausrichtung und die Position des Laserscanners zum Zeitpunkt der Messung benötigt (Beraldin et al., 2010, S. 22f). Diese Daten liefern im Normalfall eine Inertiale Messeinheit (siehe Abschnitt 2.3) und ein Satellitennavigationsempfänger (siehe Abschnitt 2.2). Auf diese Bestandteile wird im Folgenden eingangen. Anschließend werden einige bisherige Lösungsansätze zur dreidimensionalen Erfassung auf Basis von Multikopterplattformen vorgestellt.

### 2.1 Laserscanner

Ein Laserscanner besteht grundlegend aus einer Laser-Entfernungsmesseinheit und einer Ablenkeinheit. Für beide Teile gibt es verschiedenste Bauformen, auf die im folgenden eingegangen wird.

#### 2.1.1 Entfernungsmessung

Für Messung der Entfernung werden zwei verschiedene Prinzipien angewandt.

**Impulsmessverfahren** Das bei Laserscannern am häufigsten eingesetzte Verfahren ist das Impulsmessverfahren, englisch time-of-flight genannt. Hierbei werden einzelne Laserimpulse ausgesandt. Mit dem Aussenden startet ein hochgenauer Timer seine Messung. Beim Eintreffen des an einer Oberfläche reflektierten Strahles beim Laserscanner wird der Timer gestoppt. Aus dieser gemessenen Laufzeiten lässt sich die zurückgelegte Strecke des Lichtstrahles und somit die doppelte Entfernung der Oberfläche bestimmen. Hierzu wird der Brechungsindex  $n$  des vom Laser durchlaufenden Mediums benötigt. Bei der Messung in der Luft lässt sich diese mittels Temperatur, Druck und Luftfeuchte-

messung ausreichend genau berechnen. Aus der bekannten Lichtgeschwindigkeit  $c_0$  und der benötigten Zeit  $t$  lässt sich dann die Entfernung  $s$  mit der Gleichung 2.1 berechnen.

$$s = \frac{c_0}{n} \cdot \frac{t}{2} \quad | \text{ Streckenberechnung} \quad (2.1)$$

**Phasenvergleichsverfahren** Eine andere, für Laserscanner selten verwendete Methode, ist das Phasenvergleichsverfahren. Hierbei wird nicht direkt die Zeit gemessen sondern die Phasenverschiebung eines kontinuierlichen Lichtstrahles, der mit einer Sinusschwingung amplitudenmoduliert wurde (Intensitäts- bzw. Helligkeitsschwankungen). Hierdurch können weniger frequente Wellen (Modulationswelle) verwendet werden, wodurch sich bei guten Ausbreitungseigenschaften der hochfrequenten Trägerwellen die leichtere Verarbeitbarkeit von längeren Wellen ausnutzen lässt. Durch Messung des Phasenunterschiedes des Messstrahles, kann auf die Reststrecke der nicht-vollständigen Phasen des Messstrahles geschlossen werden. Als Trägerwelle normalerweise Infrarotlicht verwendet, da dieses gute Ausbreitungseigenschaften hat. Die hierfür benötigte Modulationswelle wird durch ein Quarzoszillator erzeugt. Ein hier verbautes Quarzplättchen wird durch Anlegen einer Spannung in eine Schwingung versetzt, die Schwingung verstärkt und an den Infrarot-Laser geleitet, so dass dieser das modulierte Infrarotlicht aussendet. Die maximal eindeutig messbare Entfernung ist direkt von der längsten verwendeten Wellenlänge, dem Grobmaßstab, abhängig: Da nur die Phasenunterschiede und nicht die Anzahl der Schwingungen gemessen werden können, ist die maximale eindeutige Streckenmessung genau halb so groß wie die maximale Wellenlänge (Messung von Hin- und Rückweg). Wenn längere Strecken als die halbe Wellenlänge gemessen werden, ist nicht bekannt, wie viele ganze Wellen das Licht schon zurückgelegt hat. Die Messung wäre mehrdeutig. Zur Messung der Phase werden die ausgesendete und die eingehende Messwelle mit einer Überlagerungsfrequenz vermischt, die aus diesen beiden hochfrequenten Wellen eine niederfrequente Welle erzeugt. Da die Genauigkeit der Messung der Phasenverschiebung begrenzt ist, wird durch Nutzung verschiedener Wellenlängen eine Genauigkeitssteigerung durchgeführt werden. Nach der groben Messung mit einer langen Wellenlänge, wird die Genauigkeit durch die Verwendung immer kürzerer Modulationswellen gesteigert. Eine grobe Messung ist jedoch vorher notwendig, da ansonsten die Anzahl der ganzen Schwingungen des Messstrahles unbekannt ist. (Witte & Schmidt, 2006, S. 311ff)

**Zeitmessung** Bei beiden Verfahren ist ein Problem die genaue Zeitmessung. Eine Möglichkeit dieser Messung ist die Nutzung eines Frequenzgenerators, welcher Zählimpulse erzeugt. Diese werden dann zwischen zwei Flanken der zu messenden Ausgangs- und Eingangswellen mehrfach gezählt und gemittelt und ergeben so zum Beispiel die Phasenverschiebung. Dieses Verfahren wird als digitale Messung bezeichnet. Eine andere

Methode ist die analoge Messung. Hierbei öffnet die eine Flanke den Stromfluss zu einem Kondensator, die Flanke der anderen Welle schließt sie wieder. Aus der Ladung des Kondensators kann dann auf den Phasenwinkel und die Phasenverschiebung geschlossen werden. (Witte & Schmidt, 2006, S. 314f)

## 2.1.2 Ablenkeinheit

Bei den meisten Laserscannern ist nur eine Laserentfernungsmesseinheit verbaut. Um hiermit verschiedene Punkte messen zu können, muss der Laserstrahl durch geeignete Verfahren abgelenkt werden. Auch hierfür gibt es im Airborne Laserscanning verschiedenste Ansätze: (Pack et al., 2012, S. 23ff; Beraldin et al., 2010, S. 16ff)

**Schwenkspiegel** Der Laserstrahl wird auf einen schwingenden, flachen Spiegel gerichtet. Durch die Schwingung wird der Laserstrahl in einer Ebene nach Links und Rechts abgelenkt. Durch die Bewegung des Fluggerätes wird der Laser so in Richtung der Schwingachse entsteht so auf der Oberfläche eine Zick-Zack-Linie als Messmuster.

**Rotierender Polygon-Spiegel** Beim drehenden Polygon-Spiegel dreht sich ein Prisma mit einem gleichseitigen Polygon in der Achse der Flugbewegung. Seine rechteckigen Seiten sind verspiegelt und der Laser auf diese gerichtet. Von oben gesehen wird der Strahl somit immer nur in eine Richtung abgelenkt und springt dann wieder zurück auf die andere Seite. Es entsteht ein Streifenmuster.

**Palmer Scanner** Beim Palmerscanner rotiert ein Flachspiegel um eine Achse, die fast senkrecht zur Spiegeloberfläche steht. Da der Spiegel nicht genau senkrecht auf dieser Achse montiert ist, beschreibt der auf den Spiegel gerichtete Laser einen Kreis. Durch einen im 45 Grad Winkel zur Achse stehenden Spiegel und einem sich in der Drehachse befindenden Scanner können die Strahlen auch rechtwinklig abgelenkt werden und somit eine Ebene scannen. Dies wird häufig bei terrestrischen Laserscannern im Zusammenhang mit einer zweiten Drehachse angewandt.

**Glasfaser-Scanner** Der Glasfaserscanner nutzt zur Ablenkung zusätzlich Glasfasern, welche fest verklebt sind. Hierdurch sind die Winkel zur Seite festgegeben. Zum Beispiel ein Polygonspiegel wie er zuvor beschrieben wurde, reflektiert den Messstrahl in die jeweiligen Faserbündel. Die Ablenkungswinkel sind fest vom Hersteller vorgeben.

**Zusätzliche Achsen** Zusätzlich zu den Spiegelmechanismen verfügen die terrestrische Laserscanner über eine weitere Achse. Während beim normalen Airborne Laserscanning die weitere Bewegung des Lasers durch die Fortbewegung des Fluggerätes durchgeführt wird, muss hier dies ein Motor übernehmen. Panorama-Laserscanner haben hierfür

einen Drehmechanismus um ihre Hochachse. Bei Kamerascannern, sie messen nur eine quadratische Fläche wie eine Kamera, kann die zusätzliche Bewegung auch einfach durch einen zweiten Ablenkungsspiegel erfolgen. (Beraldin et al., 2010, S. 37)

Zusätzlich haben natürlich alle Ablenk- und Drehsysteme eine Meßeinheit, die den Stand des Spiegels misst. Hierdurch lässt sich dann die Abstrahlrichtung des Lasers berechnen, beziehungsweise beim Faserlaser bestimmen, welches Faserbündel genutzt wurde. Die Richtung wird dann wiederum zur Berechnung von Koordinaten benötigt.

Bild  
malen

### 2.1.3 Oberflächeneffekte

Der vom Laser ausgesendete Impuls wird nur bei rechtwinklig zur Strahlenachse verlaufenden, ebenen Oberflächen als identischer, abgeschwächter Impuls zurückgestrahlt. Der Laser trifft bei der Messung nicht, wie idealisiert angenommenen, punktförmig auf die Oberfläche, sondern stellt einen Kreis beziehungsweise bei schrägem Auftreffen eine Ellipse mit einer bestimmten Größe, dem sogenannten Footprint dar. Hierdurch ergeben sich je nach Oberfläche verschiedene Reflexionen: Bei zum Laserstrahl schrägen Oberflächen wie einem Dach wird das Signal geweitet, die Impulsdauer des reflektierten Strahles (Echo) wird verlängert, da er auf der Oberfläche zeitversetzt auftrifft. Ein anderes Phänomen sind mehrfache Echos. Dies tritt auf, wenn zwei unterschiedlich weite entfernte Oberflächen von einem Strahl getroffen werden – zum Beispiel bei der Messung von Gebäudekanten oder Bäumen. zeigt die Echos in grafischer Form. (Beraldin et al., 2010, S. 28)

Abbildung

Laserscannern mit Impulsmessverfahren ermöglichen typischerweise bis zu vier einzelne Echos aufzuzeichnen. Alternativ gibt es Scanner, die das komplette Signal mit einer Abtastrate von bis zu 0,5 Nanosekunden digitalisieren. Hier ist es dann möglich, spezielle Auswertung aufgrund der Wellenform im Postprocessing durchzuführen. (Beraldin et al., 2010, S. 29)

Abbildung

## 2.2 Positionsbestimmung mittels globalen Navigationssatellitensystemen

Zur Bestimmung der Position des Fluggerätes wird ein Empfänger für globale Satellitennavigationssysteme (global navigation satellite system, GNSS) verwendet. Ein solcher Empfänger kann durch die Laufzeitbestimmung des Signales von verschiedenen Satelliten zum Beispiel des US-amerikanischen Navstar GPS seine aktuelle Position bestimmen. Hierzu ist eine freie Sicht zum Himmel notwendig. Je nach Auswertung und Weiterverarbeitung des Signales sind Genauigkeiten zwischen 10 Metern ohne

zusätzliche Daten und wenigen Millimetern bei statischen Dauermessungen und dem Einsatz von Daten von Referenzstationen im Postprocessing möglich. Es befinden sich pro System etwa 30 Satelliten in einer bekannten Umlaufbahn. Durch an Bord befindlich Atomuhren können die Satelliten hochgenaue Zeitstempel und sich wiederholende Codemuster aussenden. Im Fall von Navstar GPS erfolgt die Aussendung aktuell auf drei verschiedenen Frequenzen L1, L2 und L5. Für die öffentliche Nutzung ist nur L1 freigegeben. L2 und L5 sind der militärischen Nutzung vorbehalten. Durch reine Auswertung des ausgesendeten L1-Codes können Genauigkeiten bis 5 Meter erreicht werden. Für geodätische Anwendungsfälle wird zusätzlich die Phasenmessung benutzt. Hierbei wird nicht nur das dem Funksignal aufmodelierte Codemuster ausgewertet, sondern auch die Phase des Signals. Hierdurch ist es auch möglich, dass verschlüsselte L2-Signal mitzunutzen. Durch die Nutzung von Referenzstationsnetzen wie SPOS Genauigkeiten von 1-2cm in Echtzeit und von unter Zentimetergenauigkeit im Postprocessing erreicht werden (Witte & Schmidt, 2006, S. 375).

## 2.3 Inertiale Messeinheit

Bei der Inertialen Messeinheit (inertial measurement unit, IMU) handelt es sich um einen Sensor, der die Neigung sowie Drehbewegungen der Sensoreinheit misst. Sie wird benötigt, um beim Airborne Laserscanning die genaue Ausrichtung des Laserscanners zu bestimmen. Daher muss diese auch unverwindbar mit dem Laserscanner verbunden sein. In Kombination mit den Positionsdaten des GNSS-Modules ermöglicht sie die Rekonstruktion der Flugbewegungen (Trajektorie). Ein weiterer Vorteil der inertialen Messeinheit ist ihre Messfrequenz: Im Gegensatz zum GNSS, dass im Normalfall nur eine Messung pro Sekunde durchgeführt, kann die IMU bis zu 500 Messungen pro Sekunde ausführen. Sie stützt daher nicht nur die GNSS-Messung sondern hilft auch, die Trajektorie zu interpolieren und somit auch für den Bereich zwischen den GNSS-Messungen genaue Positionen zu bestimmen. (Beraldin et al., 2010, S. 23ff)

Die Messung erfolgt mit mehreren Einzelsensoren: Für die Messung der Beschleunigung in drei Dimensionen sind drei jeweils rechtwinklig zu einander stehende Beschleunigungsmesser verbaut. In der klassischen Bauform ist hierfür jeweils eine Probemasse zwischen zwei Federn gelagert. Durch eine auf die Probemasse wirkende Beschleunigung wird diese zwischen den Federn ausgelenkt. Die Messung der Drehrate erfolgt mittels drei einzelnen Kreiselinstrumenten (Gyroskop) in drei Achsen. Sie basieren auf Kreiseln, welche drehbar gelagert sind. Sie streben dazu, die Ausrichtung ihrer Drehachsen im Raum beizubehalten. Durch Messung der Kräfte kann die Drehrate berechnet werden. Einige inertiale Messeinheiten enthalten auch ein dreidimensionales Magnetometer, mit dem sich die magnetische Nordrichtung dreidimensional feststellen lässt.

Die Genauigkeit von inertialen Messeinheiten wird in ihre Messgenauigkeit und in

ihrer zeitliche Abweichung unterteilt. Die zeitliche Stabilität ist vor allem bei der Inertialnavigation, die ausschließlich auf deren Messungen basiert, wichtig.

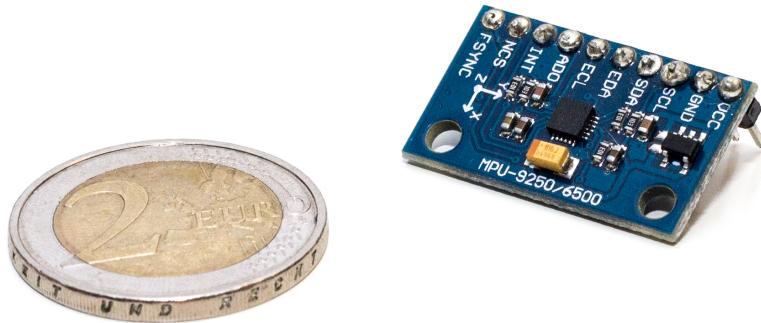


Abbildung 2.1: MPU-9250 - Low-Cost-MEMS-IMU-Modul wie es in vielen Consumer-Geräten und Multikoptern verwendet wird (schwarzes Bauteil mittig auf der Platine, eigene Aufnahme)

Hauptsächlich unterschieden werden die inertialen Messeinheiten in klassische, mechanische Systeme, wie sie bereits hier beschrieben wurden, und mikroelektromechanische Systeme, sogenannte MEMS (microelectromechanical systems). Bei den zweitgenannten handelt es sich um stark miniaturisierte Bauteile (beispielsweise Abbildung 2.1), welche zum Beispiel in aktuellen Smartphones eingesetzt werden. Sie werden für Zwecke eingesetzt, in denen keine hohen Genauigkeitsanforderungen gestellt werden und der Preis gering gehalten werden soll. Auch zur Stabilisierung der Fluglage von Multikoptern oder Gimbalen werden diese Sensoren eingesetzt. Für geodätische Anwendungsfälle werden jedoch meist noch mechanische Systeme genutzt, da deren Genauigkeit und ihre zeitliche Abweichung geringer sind.

Quelle

## 2.4 Kombination des Messsystems

Um die drei eigenständigen Messsysteme kombiniert nutzen zu können, muss die relative Position der Systeme genau bekannt sein und darf sich während des Fluges nicht verändern. Beim klassischen Airborne Laserscanning vom Helikopter werden hierfür zum Beispiel eigenständige Module entwickelt, die alle benötigten Systeme verdrehsicher enthalten und an den Kuffen des Helikopters montiert werden können (Beraldin et al., 2010, S. 23f)

Außerdem müssen alle Systeme synchronisiert werden, damit die Daten später miteinander verarbeitet werden können. Hierfür wird üblicherweise das Sekundensignal des Satellitennavigationssystems (pulse per second, PPS) genutzt. Das GNSS-System sendet dazu zu jeder vollen Sekunde der GNSS-Zeit ein Impuls aus, mit welchen sich

die anderen Sensorsysteme synchronisieren können.

## 2.5 Bisherige Systeme zur dreidimensionalen Erfassung mittels Multikoptern

Die meisten aktuellen Verfahren zur Erzeugung von 3D-Modellen unter Nutzung von kompakten Multikoptern mit einer Tragkraft von bis zu 5kg, basieren auf photogrammetrischen Verfahren. Sie erzeugen Bilder, meist unter direkter Georeferenzierung, welche im Postprocessing zu Bildverbänden verknüpft werden. Mittels Bilderkennung werden hieraus 3D-Punktwolken berechnet. Nachteil dieses Verfahrens ist es, dass ausreichende Beleuchtung vorhanden sein muss. Es kann somit nur tagsüber geflogen werden, aber auch starke Schatten können das Ergebnis verschlechtern. Für die automatische Erstellung von Punktwolken muss außerdem das Gelände ausreichende Strukturen aufweisen, damit automatische Verknüpfungen der Pixel erfolgen können.

Laserscanning als aktiver Sensor hat hier den Vorteil, dass keine zusätzliche Beleuchtung benötigt wird – der Sensor bringt sein Licht selber mit. Problematisch ist hierbei jedoch die Größe der Systeme. Aus diesem Grund wurden bisher hauptsächlich Systeme mit großen UAVs erprobt und verwendet (Ehring et al., 2016, S. 19). Durch die immer weiter fortschreitende Miniaturisierung und die Weiterentwicklung von Laserscannern zum Beispiel für die Entwicklung von autonomen Fahrzeugen werden die Scanner auch inzwischen kleiner und leistungsfähiger.

---

Quelle,  
füllen

# 3 Technische Realisierung

Im Folgenden wird zunächst auf die verwendeten Geräte und ihre technischen Eigenarten eingegangen, bevor danach auf die technischen Verbindungen eingegangen wird.

## 3.1 Verwendete Gerätschaften

### 3.1.1 Velodyne VLP-16

Um Gewicht zu sparen, wird für die Messung ein miniaturisierter Laserscanner eingesetzt. Einer dieser Kompakt-Laserscanner ist der Velodyne Puck VLP-16 (siehe Abbildung 3.1). Er hat einen Durchmesser von etwa 10 cm und eine Höhe von 7 cm bei einem Gewicht von etwa 830 g ohne Kabel und Schnittstellenbox. Es handelt sich beim VLP-16 sehr wahrscheinlich um einen Faserscanner (siehe Abschnitt 2.1.2) mit 16 Messstrahlen, der sich zusätzlich um seine Hochachse dreht. Seine Messgenauigkeit beträgt laut Datenblatt  $3 \text{ cm}$ . Gemessen wird im Impulsmessverfahren (siehe Abschnitt 2.1.1) mit einem Infrarotlaser mit einer Wellenlänge von 903nm. (Velodyne Lidar, 2017b)

Der Scanner sendet die Messstrahlen mit einem Zeitabstand von  $2,3\mu\text{s}$  hintereinander aus, gefolgt von einer Nachladezeit von  $18,4\mu\text{s}$ , so dass jeder Messstrahl alle  $55,3 \mu\text{s}$  ausgesendet werden kann (Velodyne Lidar, 2016, S. 16). Es ergibt sich somit eine durchschnittliche Messfrequenz von  $289.357 \text{ Hz}$  (siehe Gleichung 3.1). Während der Messungen dreht sich der Laserscanner je nach Einstellung über das Webinterface des Scanners mit 5 bis 20 Umdrehungen pro Sekunde (Velodyne Lidar, 2017b). Pro ausgesendeten Strahl können jeweils die erste und die stärkste Reflexion zurück gegeben werden, so dass über eine halbe Million Punkte pro Sekunde gemessen werden können (siehe Gleichung 3.1). Die Daten werden anschließend über den Netzwerkanschluss gestreamt (siehe auch Abschnitt 4.2). Außerdem verfügt der Scanner über einen Anschluss für ein GNSS-Modul des Typs Garmin GPS 18x LVC. Auch andere GNSS-Module sind nutzbar, so dass im Weiteren der Versuch unternommen wurde, hier das GNSS-Modul der Inertialen Messeinheit (siehe Unterabschnitt 3.1.2) oder eines uBlox-GNSS-Modules zu nutzen (siehe Abschnitt 3.5). Durch die Nutzung eines GNSS-Moduls am Scanner ist es möglich, die Daten mit einem hochgenauen Zeitstempel zu versehen und die Messungen des Scanners so in der Nachbearbeitung mit den

Daten aus der Inertialen Messeinheit zu verknüpfen.

$$f = \frac{1s}{55,295\mu s} \cdot 16 \frac{\text{Messstrahlen}}{\text{Messung}} = 289.357 \frac{\text{Messung}}{\text{Sekunde}} \quad (3.1)$$

$$n = 289.357 \frac{\text{Messung}}{\text{Sekunde}} \cdot 2 \frac{\text{Messwerte}}{\text{Messtrahl}} = 578.714 \frac{\text{Messwerte}}{\text{Sekunde}}$$



Abbildung 3.1: Laserscanner Velodyne VLP-16 (eigene Aufnahme)

### 3.1.2 Inertiale Messeinheit und GNSS-Empfänger iMAR iNAT-M200-FLAT

Als inertiale Messeinheit wird das auf Abbildung 3.2 zu sehende Sensorsystem des Typs iMAR iNAT-M200-FLAT verwendet. Hierbei handelt es sich um ein hochgenaues MEMS-System. Durch die Verwendung von mikroelektromechanischen Bauteilen wiegt der Sensor inklusive Gehäuse nur 550 Gramm. Er kann bis zu 500 Messungen pro Sekunde durchführen. Die Abweichung der Richtungsmessungen pro Stunde liegt unter 0,5 Grad. (iMAR Navigation GmbH, 2015)

Außerdem verfügt die verwendete Einheit über zwei differentielle Satellitennavigationsempfänger (GNSS-Module, siehe Abbildung 3.3). Durch Nutzung einer zusätzlichen GNSS-Basisstation oder auch einem entsprechenden Korrekturdienst können diese eine Positionsgenauigkeit von etwa 2 Zentimeter in Echtzeit erreichen (iMAR Navigation GmbH, 2015). Durch Postprocessing lässt sich diese sogar noch steigern. Durch die Verwendung von zwei Empfängern, die an jeweils einem Ausleger befestigt sind (siehe Bild ), kann auch die Orientierung des Scanners bestimmt werden. Hierdurch wird die ungenaue Messung des magnetischen Nordpols überflüssig. Außerdem kann die Positionssicherheit durch Mittlung der beiden Positionen erhöht werden.

mehr  
Daten

alternativ:  
Matt-  
hias  
Wil-  
kens

Bild  
machen



Abbildung 3.2: iMAR iNAT-M200-Flat im Prototypen des modularen Gehäuses, Leitungen führen zu den GNSS-Antennen (eigene Aufnahme)

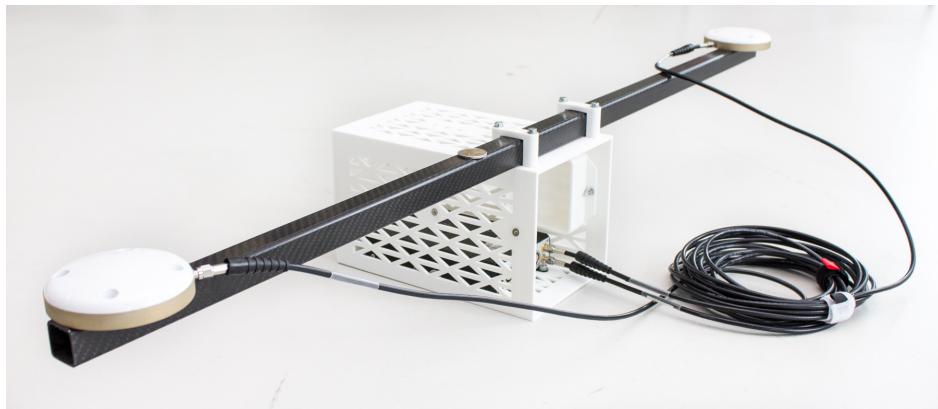


Abbildung 3.3: GNSS-Antennen des (links und rechts) iMAR iNAT-M200-Flat an Prototypen des modularen Gehäuses (eigene Aufnahme)

Im Postprocessing kann aus den Daten der inertialen Messeinheit zusammen mit denen der GNSS-Module und GNSS-Korrekturdaten eine genaue Flugbahn des Multikopters berechnet werden. Die Daten der inertialen Messeinheit werden hierbei regelmäßig durch die Daten der GNSS-Module gestützt.

### 3.1.3 Raspberry Pi 3 Typ B

Es wurde sich entschieden, die Datenverarbeitung mit einem Raspberry Pi 3 (siehe Abbildung 3.4) durchzuführen. Es handelt sich hierbei um einen von der Raspberry Pi Foundation entwickelten Einplatinencomputer. Die Stiftung gründete sich 2006, um einen erschwinglichen Computer zu entwickeln, an den Schüler direkt Hardware- und Elektronikprojekte entwickeln können. Die erste Version des Raspberry Pi kam im Februar 2012 auf den Markt. Er verfügte über 256 MB Arbeitsspeicher und einen 700 MHz Ein-Kernprozessor. Das verwendete dritte Modell verfügt über einen Vier-

Kern-Prozessor mit 1,2 Ghz und 1 GB Arbeitsspeicher. Bisher wurden alle Versionen zusammen über 11 Millionen mal verkauft. (Möcker, 2017)

Alle Modelle der Raspberry Pi Serie basieren auf Ein-Chip-System von des Halbleiterherstellers Broadcom. In diesem Chip sind die wichtigsten Bauteile des Systemes integriert wie ein ARM-Prozessor, eine Grafikeinheit sowie verschiedene andere Komponenten. Die so gering gehaltene Anzahl an einzelnen Bauelementen beim Raspberry Pi ermöglichen den geringen Preis - ein Ziel der Raspberry Pi Foundation.

Der Vorteil des Raspberry Pi zur Datenverarbeitung sind vor allem seine verschiedenen Schnittstellen zur Daten Ein- und Ausgabe (RS Components Limited, 2015):

- 4 USB 2.0 Host-Anschlüsse
- Netzwerkschnittstelle (RJ45)
- Bluetooth- und WLAN
- 27 GPIO-Ports, nutzbar als
  - Digitale Pins
  - Serielle Schnittstelle
  - I2C-Schnittstelle
  - SPI-Schnittstelle
- Stromversorgung 3,3V und 5V
- MicroUSB-Anschluss zur eigenen Stromversorgung (5V)
- MicroSD-Steckplatz
- verschiedene Video- und Audioausgänge

Außerdem vorteilhaft ist für die Nutzung am Multikopter ist seine geringe Größe und sein relativ geringer Stromverbrauch von maximal 12,5 Watt (RS Components Limited, 2015).

### **3.1.4 Multikopter Copterproject CineStar 6HL**

Bei einem Multikopter handelt es sich um ein Fluggerät mit drei oder mehr Rotoren. Es gibt entsprechend der Rotoranzahl verschiedene Modelle wie zum Beispiel den weit verbreiteten Quadrokopter oder den Hexakopter, welcher in dieser Arbeit betrachtet wird. Multikopter wurden ursprünglich für Militär- und Polizeizwecke eingesetzt, inzwischen sind sie aber auch vermehrt in kleineren Ausführungen im Privatbesitz für Videoaufnahmen zu finden (Heise Online, 2017). Angetrieben werden die handelsüblichen Modelle, welche eine Flugdauer von bis zu 30 Minuten und eine Tragkraft von bis zu



Abbildung 3.4: Raspberry Pi 3 (eigene Aufnahme)

fünf Kilogramm versprechen, mit Lithium-Polymer-Akkumulatoren (LiPo-Akkus). Die Anzahl und die maximale Umdrehung der Rotoren bestimmt die Schubkraft und somit auch die Tragkraft des Multikopters. Im Normalfall ist die Anzahl der Rotoren durch zwei teilbar, damit sich das auf das Traggestell wirkende Drehmoment aufhebt. Dies ist der große Vorteil gegenüber einem Hubschrauber, bei welchem mit einem Heckrotor dem Drehmoment um die Hochachse entgegengewirkt werden muss. Die einzelnen Motoren und Propeller werden kreuzweise angeordnet, so dass eine Drehzahländerung eines Propellerpaars zur Steuerung ausreicht. Vorteil eines Multikopters im Gegensatz zu einem Modellflugzeug ist es außerdem, dass er senkrecht starten kann und auch zum Beispiel für die Aufnahme von Bildern auf der Stelle stehen bleiben kann. Nachteil ist der höhere Energieverbrauch, so dass Flugzeuge bei gleicher Akkukapazität deutlich länger in der Luft bleiben können. (Bachfeld, 2013)

In dieser Arbeit soll der Multikopter den Laserscanner, die IMU, das Gimbal, die Stromversorgung, Datenverarbeitung und -speicherung im Betrieb tragen können. Bei der Systementwicklung des Multikopters muss daher darauf geachtet werden, dass das Gewicht möglichst gering bleibt und dennoch müssen die angehängten Messeinrichtungen auch für härtere Landungen ausgelegt sein. Bei dem verwendeten Hexakopter (siehe Abbildung 3.5) handelt es sich um einen mit einer Tragkraft von etwa 5 Kilogramm und einer Flugdauer von bis zu 20 Minuten (Schulz, 2016).

### 3.1.5 Gimbal Freefly MöVI M5

Um die Messgeräte während des Fluges des Multikopter zu stabilisieren und zu verhindern, dass sich jede Neigung der Flugsteuerung an den Laserscanner überträgt, wird ein sogenanntes Gimbal verwendet. Durch einen Regelkreis aus Motoren und einer inertialen Messeinheit (siehe auch Abschnitt 2.3), werden Neigungen und Drehungen in



Abbildung 3.5: Multikopter Copterproject CineStar 6HL mit Gimbal Freefly MöVI M5  
(eigene Aufnahme)

Echtzeit ausgeglichen. Außerdem ist es durch viele Gimbals möglich, die Messtechnik unabhängig vom Multikopter auszurichten - dies ist zum Beispiel bei der Luftbildaufnahme wichtig.

Für das Projekt wird ein Gimbal des Herstellers Freefly verwendet.

[mehr...](#)

## 3.2 Auswahl des Datenverarbeitungssystems

Ein Teil der Datenverarbeitung und die Speicherung soll direkt auf dem Sensorsystem durchgeführt werden. Da bei dem Betrieb des Multikopters jede weitere Masse die Laufzeit verkürzt, muss hierbei auf das Gewicht geachtet werden. Somit kommen für die Verarbeitung nur Ein-Chip-Computersysteme wie der Raspberry-Pi oder Mikrokontroller-Boards wie die der Arduino-Serie in Frage.

Vorteile eines Arduinos wären vor allem der geringere Stromverbrauch und die Echtzeitfähigkeit. Jedoch ist die Steuerung der Datenaufnahme über die Netzwerkschnittstelle und die Speicherung deutlich komplizierter und die Hardware nicht so leistungsfähig. Bei der Alternative, dem Raspberry-Pi übernimmt das Betriebssystem die grundlegenden Steuerungen, so dass nur noch die Daten selbst verarbeitet werden müssen. Außerdem bietet er mit der festverbauten Netzwerkschnittstelle und dem MicroSD-Karten- und der USB-Schnittstelle auch die komplette benötigte Hardware, die so nicht einzeln zusammengestellt und -gebaut werden muss.

Gerät	Laserscanner	IMU	Raspberry Pi
Spannung	9 - 18 V	10 - 36 V	5,0 V
max. Strom	0,9 A	0,75 A	2,5 A
typ. Leistung	8 W	7,5 W	12,5 W

Tabelle 3.1: Spannungs- und Strombedarf der einzelnen Module (Velodyne Lidar, 2017b; iMAR Navigation GmbH, 2015; RS Components Limited, 2015)

### 3.3 Stromversorgung

Die Stromversorgung des Raspberry-Pi an der Drohne soll mittels Lithium-Ionen-Zellen erfolgen. Der Raspberry-Pi erfordert hierbei eine stabilisierte Spannungs- und Stromversorgung. Eine fehlerhafte Stromversorgung kann hierbei zu Systeminstabilitäten führen und so im schlimmsten Fall die Datenaufzeichnung komplett verhindern. Auf den genauen Aufbau einer solchen Versorgung wird hierbei verzichtet, sondern nur die Anforderungen an die Energiequelle erläutert.

Tabelle 3.1 listet die verschiedenen Module und den die jeweils benötigte Energieversorgung auf. Der Multikopter mit der Gimbal verfügt über eine eigene Versorgung und muss daher nicht weiter beachtet werden. Außerdem hat hier eine eigene Akkukapazität auch Vorteile - auch bei einem zu hohen Verbrauch der Sensortechnik bleibt der Multikopter durch seine eigenständige Akku-Überwachung immer noch flugfähig um sicher landen zu können.

Für eine geplante Flugdauer von 30 Minuten wird bei einem angenommenen Wirkungsgrad von 90% eine Akkukapazität von mindestens 16 Wh (siehe Gleichung 3.2) benötigt. Außerdem muss ein Teil in 12 Volt und ein Teil mit 5V stabilisierter Spannung abgeben werden können. Gegebenenfalls sind hierfür auch zwei komplett unabhängige Spannungsquellen zu nutzen.

$$E = \frac{P \cdot t}{\eta} = \frac{(8 \text{ W} + 12,5 \text{ W} + 7,5 \text{ W}) \cdot 0,5 \text{ h}}{90 \%} \approx 15,6 \text{ Wh} \quad (3.2)$$

+IMU

### 3.4 Anbindung des Raspberry Pi an den Laserscanner

Durch seine vielseitigen Anschlussmöglichkeiten bildet der Raspberry Pi den Sternpunkt der Schnittstellen. Der Laserscanner wird mit einem RJ45-Kabel an der Netzwerkschnittstelle angeschlossen. Die Inertiale Messeinheit zeichnet die Daten selbstständig auf, kann aber auch mittels der als serieller Schnittstelle nutzbaren GPIO-Pins an den Raspberry Pi angeschlossen werden. Außerdem kann an diesem Port auch ein GNSS-Modul angeschlossen werden. Dieses GNSS-Modul kann im folgenden dem Raspberry Pi zu einer genauen Uhrzeit verhelfen, die für die Verarbeitung der Daten benötigt wird.

Alternativ kann auch ein an den Laserscanner angeschlossenes GNSS-Modul sein Zeitstempel per Netzwerk an den Raspberry Pi liefern. Diese Methode soll hier verwendet werden.

### 3.5 Verbindung des GNSS-Modules zum Laserscanner

Für die Versorgung des Laserscanners mit einem GNSS-Signal zur Synchronisierung wurde ein zusätzliches GNSS-Modul vom Typ uBlox NEO6M mit PPS-Signal ausgewählt (ähnlich dem auf Abbildung 3.6), da dieses kleiner und leichter ist, als die entsprechenden Adapterkabel der inertialen Messeinheit um dieses Signal zu nutzen.

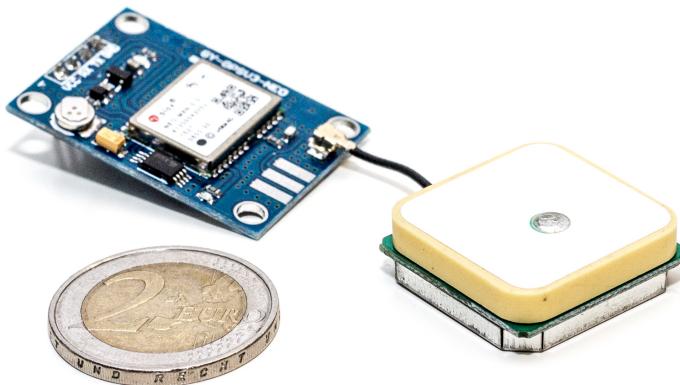


Abbildung 3.6: uBlox NEO-M8N, das Vorgängermodell NEO-6M mit PPS-Ausgang wurde verwendet (eigene Aufnahme)

Die Übertragung der Daten des GNSS-Modules zum Laserscanner erfolgt per serieller Schnittstelle über einen acht poligen Platinensteckverbinder. Bei dem vom Laserscanner benötigten Übertragungsprotokoll handelt es sich um das standardisierte NMEA-Protokoll, welches mit einer Datenrate von  $9600 \frac{bit}{s}$  und einer Signalspannung zwischen 3 und 15 Volt. Der direkte Anschluss eines uBlox GNSS-Modules vom Typ NEO-6M brachte zunächst keinen Erfolg. Messungen mit einem Ardunio (siehe Abbildung 3.7) zeigten, dass das Signal des verwendeten GNSS-Moduls nicht dem im Datenblatt von Velodyne Lidar (2017a, S. 3) entsprach. Es zeigte sich, dass das Signal gedreht werden musste, da die Definition der Signalspannung verschieden war: Der Laserscanner benötigte ein Signal, bei dem Logisch 1 mit einer Spannung von über 3 Volt (Velodyne Lidar, 2017a, S. 3) codiert ist (HIGH), beim GNSS-Modul entspricht die höhere Spannung Logisch 0.

Um das Signal zu drehen wurde ein Integrierter Schaltkreis 74HC04 verwendet. Hierbei handelt es sich um ein Logikkonverter, der die HIGH- und LOW-Signale (Signal gegen Masse) tauscht. Der Laserscanner versorgt das GNSS-Modul nur mit 5 Volt



Abbildung 3.7: Messung des Signals am uBlox NEO-6M (grün: Ausgangssignal; rot: Signal nach Nutzung eines Pegelwandler; 1000 Punkte entsprechen 5 Volt)

Spannung, der GNSS-Chip benötigt jedoch eine Spannung von 3,3 Volt. Hierfür wurde ein Spannungsregler verwendet, der die Spannung auf 3,3 Volt stabilisiert. Zur weiteren Stabilisierung wurden Kondensatoren eingesetzt. In Kombination mit dem Logikkonverter dient dieser auch als Pegelwandler. Die genaue Schaltung ist Abbildung 3.8 zu entnehmen.

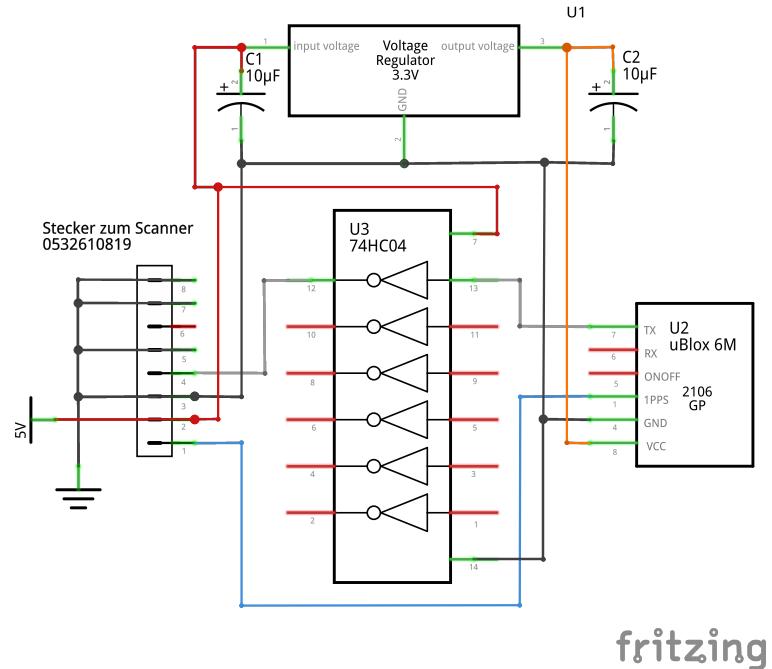


Abbildung 3.8: Schaltplan zum Anschluss des GNSS-Modules an den Laserscanner, gezeichnet in Fritzing

## 3.6 Steuerung im Betrieb

Der Betrieb des Raspberry Pi erfolgt im Betrieb ohne Tastatur und Bildschirm. Daher ist es notwendig, eine alternative Benutzerschnittstelle zu implementieren. Ein großer

Steuerbedarf ist nicht gegeben, so dass wenige Tasten zum Stoppen der Datenaufzeichnung und zum Herunterfahren des Raspberry Pi ausreichend sind. Um auch eine Steuermöglichkeit zu implementieren, die im Flug genutzt werden kann, soll ein WLAN-Access-Point und ein simpler Webserver auf dem Raspberry Pi implementiert werden, der den Zugriff zum Beispiel über ein Smartphone oder Laptop ermöglicht.

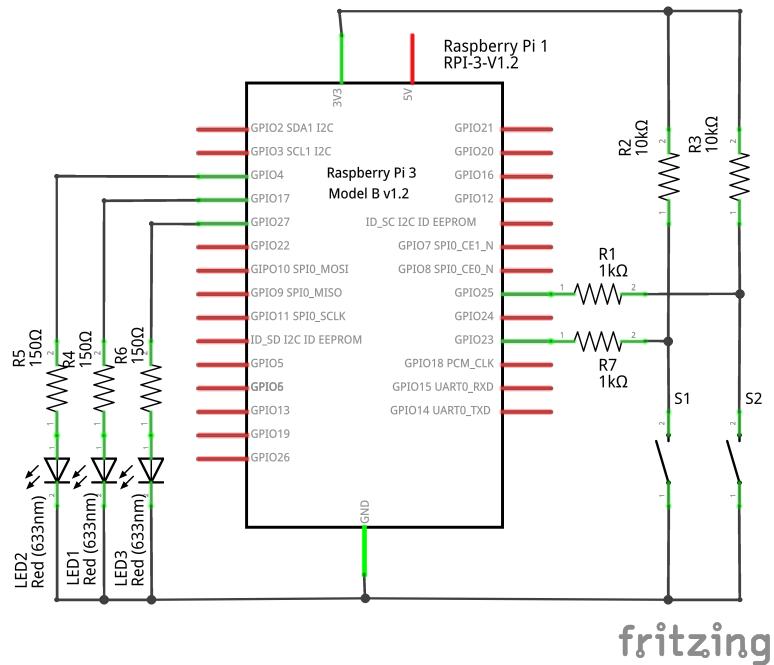


Abbildung 3.9: Schaltplan für Steuerung des Raspberry, gezeichnet in Fritzing

Abbildung 3.9 zeigt den Schaltplan des entwickelten Steuermodules. Dieses bietet mit drei Leuchtdioden und 2 Tastern die Möglichkeit, im Skript später einfache Anzeigen und Eingaben zu realisieren. Hierfür wurde eine Erweiterung auf Basis des GPIO-Portes des Raspberry Pi aufgebaut. Die zwei Taster sind über die beiden GPIO-Pins erreichbar. Ohne Betätigung werden die Eingänge über die Pull-Down-Widerstände () auf ein Low-Level gezogen. Durch Drücken des Tasters wird ein High-Signal erzeugt, welches durch das Python Skript zur Laufzeit ausgelesen werden kann. Der Widerstand dient zur Strombegrenzung. Die drei Leuchtdioden wurden mit jeweils einem 150Ohm Vorwiderstand direkt zwischen ein GPIO-Pin und Ground eingebaut. Durch Ansteuerung der GPIO-Pins lassen sich diese An- und Abschalten. Außer zum Schutz und Betrieb der LEDs verhindern die Vorwiderstände auch eine zu hohe Stromaufnahme aus den GPIO-Pins. Die genaue Belastbarkeit der Pins ist nicht dokumentiert, jedoch wird meist von einem Wert um 10mA bei 3,3 Volt gesprochen (zum Beispiel Schnabel PinNummer R? R?)

(2017)).

$$U_R = U_{GPIO} - U_{LED} = 3,3V - 2,0V = 1,3V \quad | \text{ Benötigter Spannungsabfall}$$
$$R = \frac{U_R}{I_{LED}} = \frac{1,3V}{0,01A} = 130\Omega \quad | \text{ min. Vorwiderstand} \quad (3.3)$$

## 3.7 Platinenentwurf und -realisierung

Nach dem Entwurf und Test der beiden Schaltungen aus Abbildung 3.8 und Abbildung 3.9 auf einem lötfreien Steckbrett, soll diese Schaltungen zum späteren Einsatz an Bord des Multikopters als Platine mit verlötzten Bauteilen erstellt werden. Vorteile der gelötzten Schaltung sind in diesem Projekt ihre höhere Widerstandsfähigkeit gegen Vibrationen und Korrosion. Durch die Vibrationen im Flug könnten sich so Bauteile lösen und im schlimmsten Fall zum Kurzschluss und somit zur Zerstörung führen. Auch können die Kontakte zwischen den Federklemmen und den Bauteilen durch den Betrieb außerhalb von Gebäuden durch Luftfeuchtigkeit korrodieren und somit der Kontaktwiderstand höher werden, was zu Störungen führen kann.

Für den Prototyp soll die Schaltung von Hand aufgebaut und verlötet werden. Erst in der zukünftigen Entwicklung, wenn die Schaltung ausreichend erprobt wurde, könnte es sinnvoll sein, eine Platine ätzen zu lassen. Als Platine kommen daher vorerst nur vorgefertigte Layouts in Frage:

- Lochrasterplatten (Platine mit einzelnen Lötpunkten)
- Streifenrasterplatine (Lötpunkte sind in Streifen verbunden)
- Punktstreifenrasterplatine (Streifenrasterplatine, bei denen die Streifen regelmäßig, zum Beispiel alle 4 Lötpunkte, unterbrochen sind)
- spezielle Aufsteckplatten für den Raspberry Pi

Platine  
be-  
schrei-  
ben,  
Layout

# 4 Theoretische Datenverarbeitung

## 4.1 Verwendung von Python

Zur Realisierung der Programmierung wurde die Skriptsprache Python ausgewählt. Python bietet den Vorteil vergleichsweise kurzen und gut lesbaren Programmierstil zu fördern. Hierfür werden unter anderem nicht Klammern zur Bildung von Blöcken genutzt sondern Texteinrückungen verpflichtend hierfür eingesetzt (Theis, 2011, S. 13f). Die Struktur des Programmes ist so schnell erfassbar. Außerdem ist es nicht notwendig, den Quellcode zu kompilieren. Er wird vom Interpreter direkt ausgeführt. So sind kurze Entwicklungszyklen ohne (zeit-)aufwändiges Kompilieren möglich. Änderungen und Anpassungen können schnell durchgeführt werden.

Python wurde in seiner ersten Version 1991 von Guido van Rossum freigegeben. Sein Ziel war es, eine einfach zu erlernende Programmiersprache zu entwickeln, die der Nachfolger der Sprache ABC werden sollte. Außerdem sollte die Sprache leicht erweiterbar sein und schon von Haus aus eine umfangreiche Standardbibliothek bieten. Python bietet mehrere Programmierparadigmen an, so dass je nach zu lösenden Problem objektorientiert oder strukturiert programmiert werden kann (Theis, 2011, S. 14).

Die aktuelle Version von Python (Oktober 2017) ist die Version 3.6. Das Skript wurde unter Verwendung dieser Version entwickelt. Es wurde aber auch auf eine Kompatibilität mit Python 2.7, der neusten Version von Python 2, die noch sehr häufig im Einsatz ist, geachtet. Um Teile des Quellcodes als Python-Module auch in andere Skripte einfach einbinden zu können, aber auch den Quelltext übersichtlich zu halten, wurde der objektorientierte Programmierstil gewählt.

## 4.2 Datenlieferung vom Laserscanner

Der Laserscanner Velodyne VLP-16 liefert seine Daten als UDP-Netzwerkpakete in einem proprietären binären Datenformat. Diese Daten sind nicht direkt lesbar sondern müssen vor einer weiteren Nutzung aufbereitet und umgeformt werden. Dies soll mittels des in dieser Arbeit entwickelten Skriptes durchgeführt werden.

Ein Datenpaket (siehe Tabelle 4.1) besteht jeweils aus einem Header von 42 Bytes, gefolgt von 12 Datenblöcken mit jeweils 32 Messungen, abgeschlossen von 4 Bytes, die

den Zeitstempel angeben und 2 Bytes, die den eingestellten Scan-Modus zurückliefern. Jeder Datenblock enthält die aktuelle horizontale Ausrichtung des rotierenden Lasers und darauf folgend die Messwerte von zwei Messungen der 16 Laserstrahlen. Die genaue Horizontalrichtung zum Zeitpunkt der Messung muss aus den Horizontalrichtungen aus zwei auf einander folgenden Messungen interpoliert werden.

Header			Netzwerk-Header	42 Bytes	
Block 1	0-1		Flag	2 Bytes	
	2-3		Horizontalrichtung	2 Bytes	
	Messung 1	4-5	Entfernung	2 Bytes	
		6	Reflektivität	1 Byte	
	Messung 2	7-8	Entfernung	2 Bytes	
		9	Reflektivität	1 Byte	
Messungen 3 - 32					
Block 2 - 12					
Time		1200-1204	Zeitstempel	4 Bytes	
Factory		1205-1206	Return-Modus	2 Bytes	

Tabelle 4.1: Aufbau der Daten des Netzwerkpaketes, nach Velodyne Lidar (2016)

Der Laserscanner sendet bei der Einstellung Dual Return, also der Rückgabe vom stärksten und letzten Echo pro Messung bis zu 1508 Pakete dieser Form pro Sekunde (Velodyne Lidar, 2016, S. 49). Die Ausgangsdaten werden, bei einer Paketgröße von 1248 Bytes mit einer Datenrate von 1,8 MB/s empfangen (siehe Gleichung 4.2). Hierbei werden fast 600.000 Messwerte pro Sekunde übertragen (siehe Gleichung 4.1).

$$1508 \frac{\text{Pakete}}{\text{Sekunde}} \cdot 12 \frac{\text{Datenblöcke}}{\text{Paket}} \cdot 32 \frac{\text{Messungen}}{\text{Datenblock}} = 579.072 \frac{\text{Datensätze}}{\text{Sekunde}} \quad (4.1)$$

$$1508 \frac{\text{Pakete}}{\text{Sekunde}} \cdot 1248 \frac{\text{Bytes}}{\text{Paket}} = 1,79 \text{ MB/s} \quad (4.2)$$

### 4.3 Geplantes Datenmodell

Die Daten des Laserscanners sollen in einer einfach lesbaren Textdatei abgelegt werden. In der Nachbereitung sollen die Daten aus dieser Textdatei mit den Daten der Inertialen Messeinheit und des GNSS-Empfängers verknüpft werden, um so die Daten georeferenziieren zu können. Als Verknüpfung bietet sich hier der Zeitstempel an. Die Inertialen

Messeinheit und der Laserscanner können hierbei die Zeitdaten aus dem GNSS-Signal verwenden. Hierdurch sind hochgenaue Zeitstempel möglich. Die Zeitinformation bildet also einen wichtigen Schlüssel in den Daten. Als einfaches Textformat wurden durch Tabulator getrennte Daten, jeweils eine Zeile je Messung, gewählt. Folgende Daten sind in dieser Reihenfolge enthalten:

- Zeitstempel in Mikrosekunden
- Richtung der Messung in der Rotationsebene in Grad
- Höhenwinkel zur Rotationsebene in Grad
- Gemessene Entfernung in Metern
- Reflektivität auf einer Skala von 0 bis 255

Problematisch ist bei diesem Datenmodell jedoch die benötigte Datenrate. Eine Datenzeile erfordert 29 Bytes und somit wird bei über einer halben Million Messungen pro Sekunde (siehe Gleichung 4.1) eine Datenschreibrate von mindestens 16 MB/s benötigt (siehe Gleichung 4.3). Da das Schreiben nicht dauerhaft erfolgt, sollte die Datenrate bevorzugt deutlich höher sein.

$$579.072 \frac{\text{Datensätze}}{\text{Sekunde}} \cdot 29 \frac{\text{Bytes}}{\text{Datenzeile}} = 16,02 \text{ MB/s} \quad (4.3)$$

Erste Test ergaben, dass diese Verarbeitungsgeschwindigkeit nicht mit dem Raspberry Pi erreicht werden konnte. Außerdem benötigen die Daten sehr viel Speicher. Daher wurde sich später für eine Hybridlösung entschieden (siehe Kapitel 5).

## 4.4 Weiterverarbeitung der Daten zu Koordinaten

Die als Text gespeicherten Rohdaten sollen dann im Rahmen einer weiterführenden Arbeit zu Koordinaten umgewandelt werden. Zu dieser Umwandlung werden die Positionen des Laserscanners mittels dem GNSS-Empfänger in der IMU und die Neigungsdaten aus der IMU verwendet. Die Neigungen werden dazu direkt mit den Winkeldaten verrechnet.

Bei der Berechnung ist jedoch zu beachten, dass der Ursprungsort der Entfernungsmeßung zwar in der Drehachse des Laserscanners liegt, jedoch der Ursprungsort der ausgesendeten Strahlen etwa 40mm in Strahlrichtung verschoben ist (siehe Abbildung 4.1). Bei der Streckenberechnung ist diese Strecke mit enthalten, jedoch kann zur Berechnung der Z-Komponente der lokalen Koordinaten nicht einfach der Höhenwinkel und die gemessene Strecke verwendet werden. Die lokalen Koordinaten berechnen sich somit nach der Gleichung 4.4.

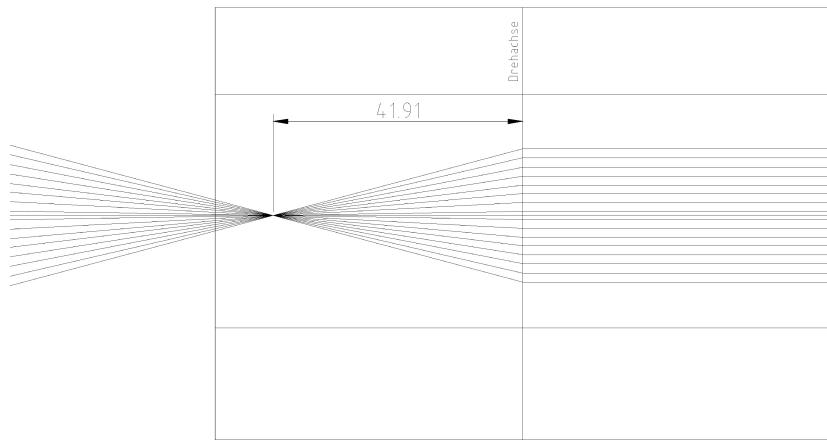


Abbildung 4.1: Strahlengang im Laserscanner VLP-16, Werte in Millimetern, nach Velodyne Lidar (2014)

$h$  : Höhenwinkel ( $-15^\circ - 15^\circ$ )

$r$  : Horizontalrichtung ( $0^\circ - 360^\circ$ )

$s$  : Gemessene Strecke

$$\begin{aligned} s_S &= s - 41,91 \text{ mm} && | \text{ Schrägstrecke nach dem Fokuspunkt} \\ s_H &= s_S * \cos(h) + 41,91 \text{ mm} && | \text{ Horizontalstrecke von der Drehachse} \end{aligned} \quad (4.4)$$

$$X = s_H \cdot \sin(r) \quad | \text{ Y-Achse in Nullrichtung}$$

$$Y = s_H \cdot \cos(r)$$

$$z = s_S \cdot \sin(h)$$

## 4.5 Anforderungen an das Skript

Aus den technischen Vorgaben ergeben sich dann folgende Funktionen, die das Skript aufweisen muss:

- Rohdaten vom Scanner abrufen
- Zeit vom GNSS-Modul abrufen
- Steuerungsmöglichkeit mittels Hard- und Software
- Umwandlung in eigenes Datenmodell

Der Ablauf der einzelnen Schritte ist oft abhängig vom Fortschritt anderer Schritte und Gegebenheiten. Daher wurden die benötigten, einzelnen Schritte vorerst als grober Ablaufplan skizziert. So hat der Raspberry Pi keinen eigenen Zeitgeber. Um die Dateien aber mit dem korrekten Zeitstempel zu versehen, ist daher eine aktuelle Uhrzeit notwendig - diese liefert das GNSS-Modul, welches am Laserscanner angeschlossen ist, sofern ein GNSS-Fix besteht. Es muss also vor dem Erzeugen der Dateien auf ein gültiges GNSS-Signal gewartet werden. Der endgültige, vereinfachte Ablaufplan ist der Abbildung 4.2 zu entnehmen.

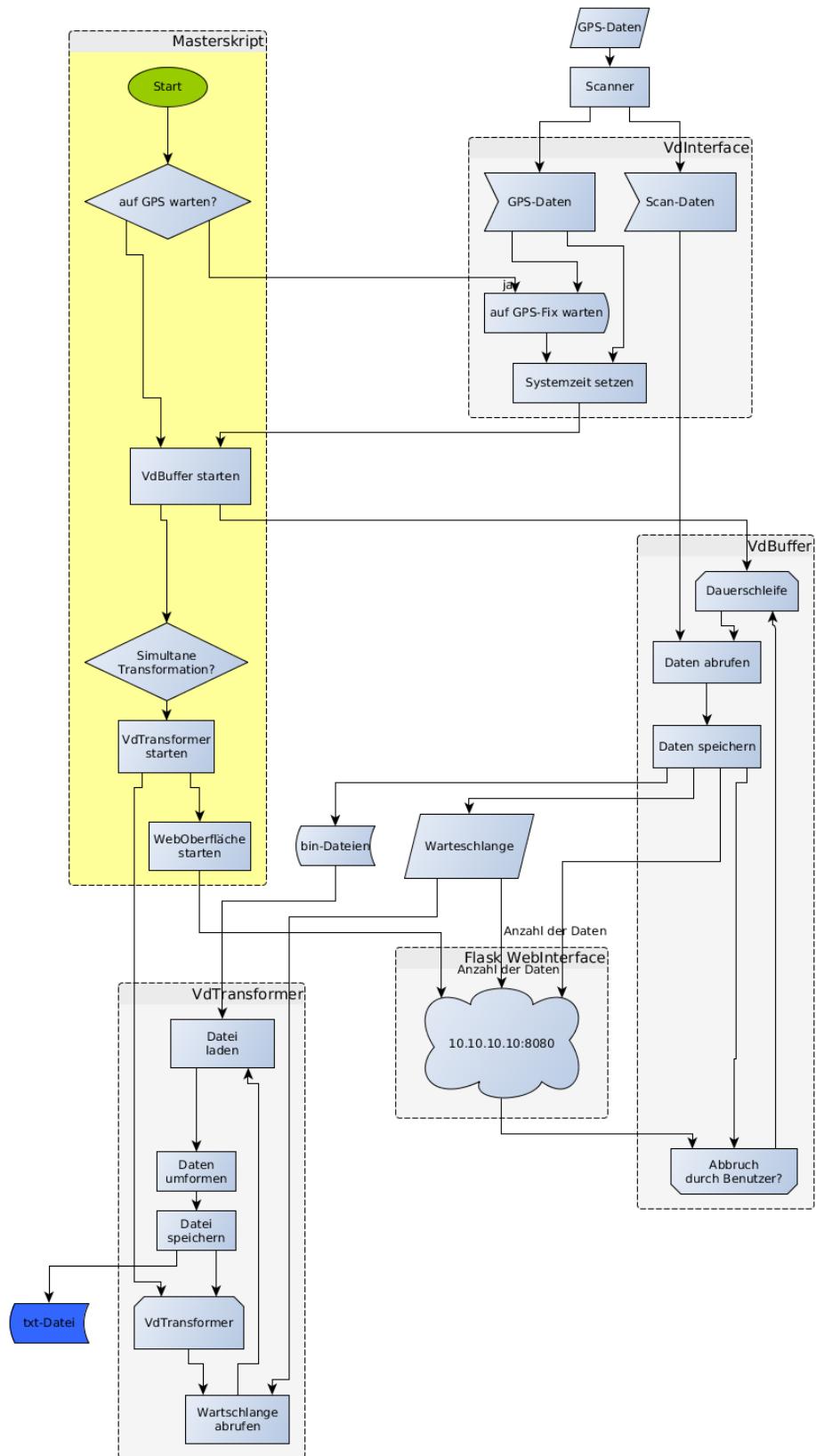


Abbildung 4.2: Vereinfachter Ablaufplan des Skriptes

# 5 Entwicklung des Skriptes

## 5.1 Klassenentwurf

Da das Skript objektorientiert programmiert werden soll, wurde zunächst mit Hilfe des Ablaufplanes aus Abbildung 4.2 die benötigten Klassen entworfen. Die endgültigen Klassen sind der Abbildung 5.1 zu entnehmen. Auf die genauen Funktionen der einzelnen Klassen wird im Abschnitt [eingegangen](#).

[Verlinken](#)

## 5.2 Evaluation einzelner Methoden

Um eine einfache Fehlersuche zu ermöglichen, wurde die grundlegenden Funktionen in einzelnen Skripte entwickelt und geprüft. Diese kleineren Skripte haben den Vorteil, Fehler schneller eingrenzen zu können und auch schon früh konzeptionelle Fehler zu entdecken. In diesem Schritt wurde bemerkt, dass das ein großes Problem die Geschwindigkeit der Datenverarbeitung ist.

**Datenempfang** Die Verbindung zum Laserscanner mittels Python-Socket funktionierte ohne weitere Probleme. Die binären Daten konnten zeitgleich abgespeichert werden.

**Datentransformation** Zunächst war es geplant, die Daten direkt in das in Abschnitt 4.3 vorgestellte Datenmodell umzuformen. Hierzu sollte der Empfang der Daten direkt eine Umformmethode starten. Die Versuche erfolgten zunächst mit dem im vorherigen Test aufgezeichneten Daten. Schon hier zeigte sich, dass die Umwandlung der aufgezeichneten Daten etwa die fünffache Zeit als für deren Messung und Aufzeichnung benötigt wurde beanspruchte. Wie erwartet, brachte auch das direkte Einlesen der Daten vom Scanner keinen Erfolg. Es folgte ein Überlauf des Netzwerk-Buffer und somit der Verlust von Messdaten. Grund hierfür war hauptsächlich die benötigte Prozessorzeit. Die Nutzung einer schnelleren Datenspeicherung auf einer Solid-State-Disk mit einer Schreibrate von bis zu 300 MB/Sekunde änderte nichts an der Geschwindigkeit des Skriptes. Auch das Erzeugen eines neuen Threads für jeden empfangenen Datensatz war nicht erfolgsversprechend, da bis zu 1500 Threads pro Sekunde hierdurch gestartet wurden und das gesamte System überlastet wurde. Die Umformung musste

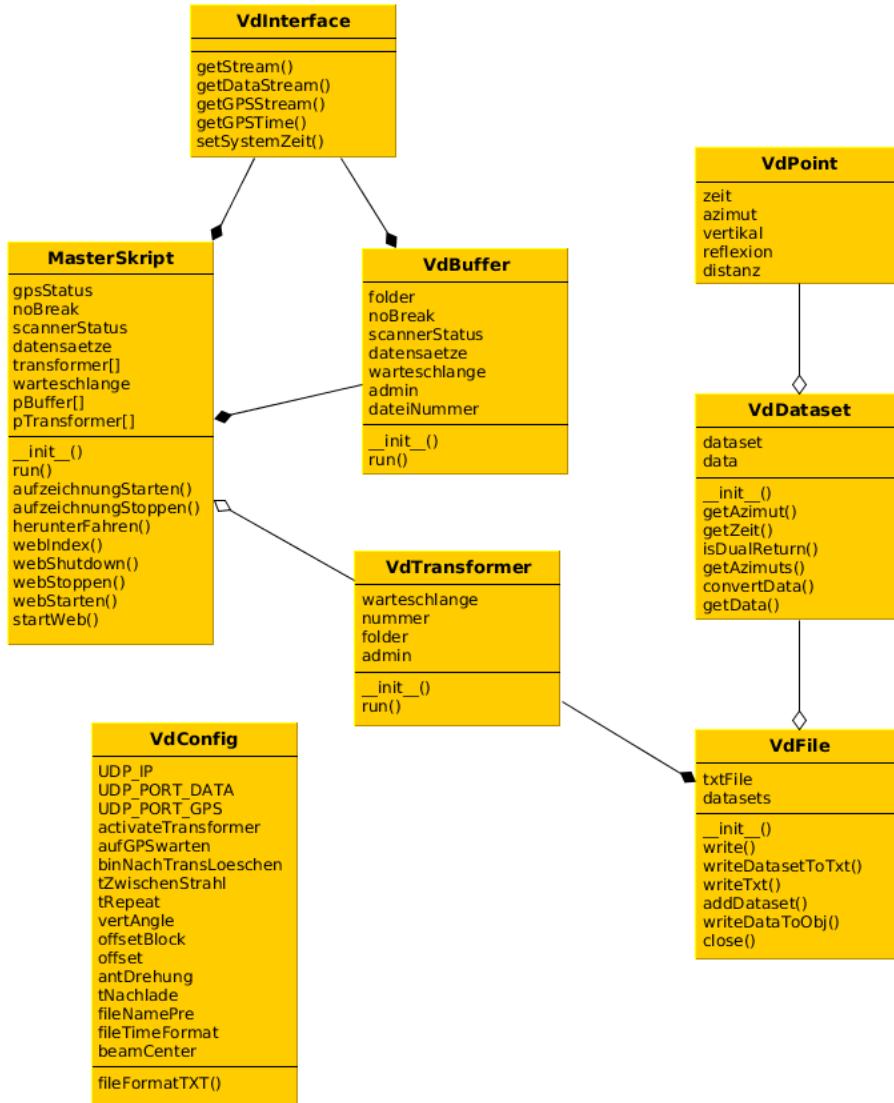


Abbildung 5.1: UML-Klassendiagramm

daher von dem Datenempfang entkoppelt werden und das Skript für die Nutzung von Mehrkernprozessoren optimiert werden. Threads in Python laufen dennoch in einem Prozess und somit nur auf einem Prozessor. Es wurde das in Abschnitt 5.3 vorgestellte Multikern-Konzept erarbeitet.

**Hardware-Steuerung** Ein Tastendruck auf dem Steuerungsmodul (siehe Abschnitt 3.6) sollte den Raspberry Pi zum Beispiel herunterfahren. Auch dieses Skript wurde getestet. Ein Problem hierbei war es, dass das Skript Administratorrechte (`root`) benötigte, um den Rechner herunterfahren zu können. Hierfür wurde jedoch eine Lösung gefunden, indem dem Nutzer `pi` die entsprechenden Rechte zum Herunterfahren gegeben wurden (siehe Abschnitt 6.2). Eher zufällig zeigte sich aber noch ein anderes Problem: Sofern das Skript im automatisch mit dem Start des Raspberry Pi gestartet wurde und

das Steuerobermodul nicht angeschlossen war, fuhr der Raspberry Pi automatisch nach wenigen Sekunden Betrieb herunter. Da mit dem fehlenden Modul auch die Pull-Down-Widerstände fehlten, war der GPIO-Pin auf einem nicht definierten Zustand. Es kam dazu, dass er zufällig auf einem HIGH-Niveau war, welches als Drücken des Tasters interpretiert wurde. Nach Überschreiten der konfigurierten Haltezeit des Ausschalters von zwei Sekunden, wurde der Herunterfahrprozess gestartet. Um dieses Problem zu unterdrücken, wurde dem Skript eine vorherige Abfrage hinzugefügt, die beim Start überprüft, ob die beiden Taster auf einem Low-Niveau sich befinden, dass durch die beiden angeschlossenen Pull-Down-Widerstände erreicht wird. Falls dieses nicht der Fall ist, beendet sich die Hardwaresteuerung selbstständig.

## 5.3 Multikern-Verarbeitung der Daten

Da bei der Evaluation der einzelnen Methode herausgefunden wurde, dass die Verarbeitungsgeschwindigkeit des Raspberry Pi für eine sofortige Transformation der Daten nach deren Eingang zu langsam ist, wurde ein Konzept erarbeitet, den hierdurch auftretenden Messdatenverlust zu unterdrücken.

Der Verarbeitung musste ein weiterer Buffer vorgeschaltet werden. Da aber das Abschalten des Raspberry Pi, zum Beispiel durch einen Verlust der Energieversorgung, nicht zu Datenverlusten führen sollte, konnten nicht die in Python integrierten Funktionen zur Datenzwischenspeicherung verwendet werden – diese setzen zur Zwischenspeicherung auf den Arbeitsspeicher, der durch Stromverlust gelöscht wird. Das dauerhafte Schreiben auf die Festplatte – im Fall des Raspberry Pi einer MicroSD-Speicherkarte – führt aber zur weiteren Verlangsamerung. Es wurde daher eine Hybridlösung erarbeitet.

Die Arbeit wird nun auf mehrere Prozesse verteilt:

- Start des Skriptes und Gesamtsteuerung in Prozess mit mittlerer Priorität (Klasse `VdAutoStart`, mit Threads für Weboberfläche (Methode `startWeb()` in Klasse `VdAutoStart` und Hardwaresteuerung (Klasse `VdHardware`)
- Sammeln der Daten mit höchster Priorität (Klasse `VdBuffer`)
- Umformen der Daten durch mehrere Prozesse je nach Prozessorkernanzahl mit erhöhter Priorität (Klasse `VdTransformer`)

Die Daten werden nun zuerst für wenige Sekunden im Arbeitsspeicher gesammelt. Sofern 7.500 Datensätze zwischengespeichert wurden – je nach Einstellung des Laserscanners in etwa fünf oder zehn Sekunden – werden diese im Dateisystem als binäre Datei abgelegt und der Dateiname in einer Warteschlange aus dem `multiprocessing`-Modul von Python (Klasse `Queue`) abgelegt. Die Prozesse zum Umformen der Daten fragen diese Warteschlange nun ab, verarbeiten jeweils eine binäre Datei und hängen

die Ergebnisse an eine Ergebnis-Textdatei an. Die Dateinamen der binäre Dateien, dessen Bearbeitung begonnen wurde, werden aus der Warteschlange entfernt. Nach dem Schreiben der umgeformten Daten werden die binären Dateien aus dem Dateisystem entfernt. Damit die Umformer-Prozesse beim Schreiben nicht auf einander warten müssen, schreibt jeder Prozess in eine andere Ergebnisdatei. Diese können nach der Messung einfach zusammengefügt werden. Falls nun zum Beispiel die Stromversorgung unterbrochen wird, sind nur die Daten der maximal letzten 10 Sekunden verloren. Daten, die älter sind, sind entweder als binäre Daten oder als Textdatei gespeichert. Durch neues Starten des Umformerprozesses können die restlichen, noch nicht gewandelten Daten umgeformt werden.

Durch diese Prinzip stört eine stockende Datenumformung nicht das Aufzeichnen der Daten vom Scanner. Sofern der Raspberry Pi nicht die Geschwindigkeit der Umformung halten kann, werden einfach mehr binäre Dateien zwischengespeichert, die gegebenenfalls im Postprocessing umgewandelt werden können.

## 5.4 Klassen

Es folgt die Beschreibung der einzelnen Klassen. Auf die Konstruktor-Methoden `__init__()` wird nicht eingegangen, da hier meist nur Variablen deklariert werden.

### 5.4.1 VdAutoStart

Die Klasse mit dem Namen `VdAutoStart` (siehe Anhang A.1) steuert den automatischen Start des Skriptes beim Hochfahren des Raspberry Pi. Sie ist verantwortlich für den korrekten Start der einzelnen Skriptteile in der richtigen Reihenfolge. Außerdem sind in der zugehörigen Datei auch alle Programmteile abgelegt, die nicht zu einer Klasse gehören, wie zum Beispiel der Startaufruf.

Zu Beginn wird geprüft, auf welcher Umgebung das Skript läuft. Sofern es auf einem Raspberry Pi läuft, werden später zusätzliche Klassen aufgerufen.

**run()** Das Skript startet mit dieser Methode. Als erstes wird die Hardwaresteuerung in einem neuen Thread aktiviert, sofern es sich bei der Umgebung um einen Raspberry Pi handelt und die entsprechenden Module zur Steuerung der GPIO-Pins installiert sind. Sofern auf ein GNSS-Fix zur Einstellung der Uhrzeit gewartet werden soll, wird zunächst die Methode `getGNSSTime()` der Klasse `VdInterface` aufgerufen. Außerdem wird der für die zu speichernden Daten genutzte Ordner angelegt.

Folgend wird ein Prozess der Klasse Buffer gestartet. Um eine Kommunikation zu diesem neuen Prozess zu ermöglichen, werden einige Pipes und eine Warteschlange (Queue) mit an den neuen Prozess übergeben.

Sofern die Daten simultan transformiert werden sollen, wird abgefragt, wie viele Prozessoren dem System zur Verfügung stehen und eine entsprechende Anzahl an Transformer-Prozessen gestartet ( $n - 1$ , mindestens 1). Auch hier werden zur Kommunikation Pipes und die Queue verwendet.

**aufzeichnungStarten()** Die Methode startet den Buffer-Vorgang des VdBuffer-Prozesses. Sie wird durch die Steuersysteme aufgerufen.

**aufzeichnungStoppen()** Diese Methode stoppt die Aufzeichnung durch den VdBuffer-Prozess und wird auch durch die Steuersysteme genutzt.

**herunterfahren()** Ermöglicht den Steuersystemen, dass die VdBuffer- und VdTransformer-Prozesse zu unterbrechen und das System herunterzufahren.

**\_\_main\_\_( )** Die Methode `__main__()` gehört nicht zu der Klasse sondern ist nur mit in dieser Datei abgelegt. Sie erzeugt ein Objekt der Klasse `VdAutoStart` und startet die `run()`-Methode. Außerdem wird die Weboberfläche hieraus gestartet.

**Flask-Webinterface app** Die Weboberfläche zur Steuerung wird mit dem Modul `Flask` erzeugt. Die Weboberfläche wird durch die `main`-Methode in einem zusätzlichen Thread gestartet.

## 5.4.2 VdInterface

Die Klasse `VdInterface` (siehe Anhang A.4) übernimmt die Kommunikation mit dem Laserscanner.

**getDataStream()** Die Methode öffnet den Netzwerk-Stream, der die Messdaten des Laserscanners überträgt. Das Auslesen der Daten aus dem Stream erfolgt dann in der Klasse `VdBuffer`.

**getGNSSStream()** Durch die Methode wird der Netzwerkstream geöffnet, der die aktuellen Datensätze des an den Laserscanner angeschlossenen GNSS-Moduls überträgt. Aus den Daten kann zum Beispiel die Uhrzeit gewonnen werden.

**getStream()** Da die benötigten Schritte zum Öffnen der beiden vorher vorgestellten Streams identisch sind, wurden diese Funktionalitäten in diese Methode ausgelagert, um den Code möglichst redundanzfrei zu halten.

**getGNSSTime()** Diese Methode fragt die Daten, die über den GNSS-Netzwerkstream geliefert werden, solange ab, bis der NMEA-Datensatz eine GPRMC-Nachricht mit einem GNSS-Fix, also einer gültigen Position, enthält. Diese Nachricht enthält außer der aktuellen Position und dem GNSS-Fix-Status auch den aktuellen Datums- und Zeitstempel. Die erkannte Uhrzeit wird als Python-timestamp an die Methode `setSystemZeit()` weitergegeben.

**setSystemZeit()** Die Methode setzt die aktuelle Systemzeit auf Basis eines ihr übergebenen Zeitstempels. Hierzu werden Befehle des Linuxbetriebssystems angesprochen, dessen Verwendung vorher freigegeben werden muss (siehe Abschnitt 6.2). Zuerst wird die Netzwerk-Zeitsynchronisierung abgeschaltet, dann die Uhrzeit gesetzt und die Synchronisierung wieder aktiviert. Die Deaktivierung ist notwendig, da ansonsten Linux keine Änderung an der Uhrzeit erlaubt. Eine komplette Deaktivierung der Netzwerk-Zeitsynchronisierung ist nicht zu empfehlen, da so die Uhr immer manuell gestellt werden muss.

#### 5.4.3 VdHardware

### 5.5 Beispiel-Quelltext-Zitat

Die Daten werden eingelesen (siehe Zeile 18, Listing 5.1)

Listing 5.1: Quelltext-Test

```
16     def __init__(self, fileType = "txt", fileName = ""):  
17         # Dateiname erzeugen, sofern kein Dateiname mitgeliefert  
18         if (fileName == ""):  
19             # Jahr-Monat-TagTStunde:Minute:Sekunde an Dateinamen anhaengen  
20             fileName = VdConfig.fileNamePre
```

# 6 Konfiguration des Raspberry Pi

Als Grundlage wurde auf die MicroSD-Karte, die dem Raspberry Pi als Festplatte dient, das Betriebssystem Raspbian aufgespielt. Hierbei handelt es sich um ein Derivat von Debian GNU/Linux, das speziell auf die Hardware des Raspberry Pi angepasst wurde. Die aktuelle Version (Stand 27.10.2017) nennt sich Raspian Stretch. Für die Verwendung als Verarbeitungsgerät ohne angeschlossenen Display reicht die Variante ohne grafische Benutzeroberfläche aus (Raspbian Stretch Lite). Die Konfiguration des Raspberry Pi erfolgt vollständig über Konfigurationsdateien. In dieser Arbeit erfolgte die Konfiguration per Fernzugriff über SSH, einem Standard für das Fernsteuern der Konsole über das Netzwerk. Eine Konfiguration hätte aber auch mittels einem angeschlossenen Display und einer USB-Tastatur erfolgen können.

Die Änderungen der Konfigurationsdateien erfolgte mit vorinstallierten Editor `nano` unter Nutzung der Administratorrechte. Ein solcher Aufruf erfolgt zum Beispiel mit dem Befehl `sudo nano /pfad/zur/konfiguration.txt`. Nachfolgend müssen die betroffenen Programme oder sogar das komplette Betriebssystem neugestartet werden. Der Neustart eines Services erfolgt zum Beispiel mit dem Aufruf `sudo service programmname restart`, der Neustart des Betriebssystems mit `sudo shutdown -r now`. Es empfiehlt sich, von allen zu ändernden Konfigurationsdateien Sicherungskopien anzulegen. Dies erfolgt zum Beispiel mit `sudo cp original.txt original.old.txt` (Kopieren) oder `sudo mv original.txt original.old.txt` (Verschieben, zum Beispiel zum Anlegen einer komplett neuen Datei). Auf diese Linux-Grundlagen wird im folgenden nicht mehr eingegangen.

## 6.1 Installation von Raspbian

Die Installation von Raspbian erfolgt durch das Entpacken des Installationspaketes von der Website der Raspberry Pi Foundation auf einer leeren MicroSD-Karte mit dem Tool `Etcher`. Auf der nach dem Entpacken erzeugten boot-Partition wird eine leere Datei mit dem Namen `ssh` angelegt. Hierdurch wird sofort nach dem Start der SSH-Zugang über das Netzwerk zum Raspberry Pi ermöglicht, die IP-Adresse wird per DHCP, zum Beispiel von einem im Netzwerk vorhandenen Router, bezogen. Nach dem Einloggen zum Beispiel unter Linux mit dem Befehl `ssh pi@raspberrypi` und dem Passwort `raspberry`, kann mittels `passwd` das Passwort verändert werden.

	Schnittstelle	IP-Adresse bzw. Bereich	
Laserscanner	Ethernet	192.168.1.111	statisch
Raspberry Pi	Ethernet	192.168.2.110	statisch
	WiFi	10.10.10.10	statisch
Client	WiFi	10.10.10.100	- 10.10.10.254

Tabelle 6.1: IP-Adressen-Verteilung

## 6.2 Befehle mit Root-Rechten

Linux erlaubt das Ändern der Zeit und das Herunterfahren über die Kommandozeile nur dem Administrator (`root`). Da es jedoch nicht empfohlen ist, Skripte als `root` auszuführen, muss hier eine andere Lösung gefunden werden, um den Skripten die Möglichkeit zu geben, den Raspberry Pi auf Tastendruck oder per Web-Steuerung herunterzufahren. Hierfür wurde dem normalen Nutzer (`pi`) die Rechte gegeben, einzelne Befehle als Admin ohne Passwortabfrage auszuführen. Diese Rechte können dem Nutzer durch Eintragung in die Konfigurationsdatei `/etc/sudoers` gegeben werden. Da eine fehlerhafte Änderung der Datei den kompletten Administratorzugang zum System versperren kann, wird die Datei mit dem Befehl `visudo` überarbeitet, der nach dem Editieren die Datei auf Fehler prüft. Die zusätzlichen Einträge in der Konfiguration sind dem Listing 6.1 zu entnehmen.(ubuntuusers.de, 2017)

Listing 6.1: Änderung der `/etc/sudoers`

```

1 # Cmnd alias specification
2 Cmnd_Alias VLP = /sbin/shutdown, /sbin/timedatectl

4 # User privilege specification
5 pi  ALL=(ALL) NOPASSWD: VLP

```

## 6.3 IP-Adressen-Konfiguration

Per Ethernet soll der Raspberry auf die IP-Adresse 192.168.1.111 konfiguriert werden, da diese IP-Adresse im Laserscanner als Host eingestellt war und an diesen die Daten vom Scanner übertragen werden. Die IP-Adresse des Raspberry Pi im WLAN wurde fest auf die gut zu merkende Adresse 10.10.10.10 geändert, hierüber erfolgt später der Zugriff auf die Weboberfläche (siehe auch Tabelle 6.1).

Die Konfiguration der IP-Adressen für den Raspberry erfolgt in der Konfigurationsdatei `/etc/network/interfaces` (siehe Listing 6.2. Raspberry Pi Foundation (2017)

Original  
hinzufügen

Listing 6.2: Konfiguration der /etc/network/interfaces

```
1 # localhost
2 auto lo
3 iface lo inet loopback

5 # Ethernet
6 auto eth0
7 iface eth0 inet static
8   address 192.168.1.110
9   netmask 255.255.255.0
10  gateway 192.168.1.110

12 # WLAN
13 allow-hotplug wlan0
14 iface wlan0 inet static
15   address 10.10.10.10
16   netmask 255.255.255.0
17   network 10.10.10.0
```

Zur Konfiguration der dynamischen IP-Adressen der Clients im WLAN wird ein DHCP-Server eingerichtet. Ein solcher Server weißt neuen Geräten – beziehungsweise welchen, die länger nicht im Netzwerk waren – automatisch eine neue, unverwendete IP-Adresse zu. Hierdurch benötigen die Clients keine spezielle Konfiguration und ihre IP-Einstellungen können auf dem üblichen Standardeinstellungen verbleiben (automatische IP-Adresse beziehen). Als DHCP-Server wird hier das Paket `dnsmasq` verwendet. Außer dem DHCP-Server bietet dieses Paket auch einen DNS-Server, der es erlaubt, den Geräten auch einen Hostname zuzuweisen. So wäre der Zugriff zum Beispiel über den Hostname `raspberry.ip` anstatt durch Eingabe der IP-Adresse möglich.

Die Konfiguration des DHCP-Servers ist vergleichsweise einfach und benötigt nur das verwendete Netzwerk-Interface, hier `wlan0`, den zu nutzenden IP-Bereich, die Netzmarske und die Zeit, nach der eine IP-Adresse an ein anderes Gerät vergeben werden darf, die sogenannte Lease-Time (siehe Listing 6.3). Raspberry Pi Foundation (2017)

Listing 6.3: Konfiguration der /etc/dnsmasq.conf

```
1 interface=wlan0
2 dhcp-range=10.10.10.100,10.10.10.254,255.255.255.0,24h
```

## 6.4 Konfiguration als WLAN-Access-Point

Um einen Zugriff auf die Python-Weboberfläche des Skriptes und die Konfiguration des Laserscanners zu ermöglichen, soll der Raspberry Pi selbst als WLAN-Access-Point fungieren. Hierzu das Paket `hostapd` verwendet. Zur Konfiguration werden die Einstellungen in die Datei `/etc/hostapd/hostapd.conf` geschrieben. Raspberry Pi Foundation (2017)

Listing 6.4: Konfiguration der /etc/hostapd/hostapd.conf

```
1 # WLAN-Router-Betrieb
```

```

3 # Schnittstelle und Treiber
4 interface=wlan0
5 #driver=nl80211

7 # WLAN-Konfiguration
8 ssid=VLPinterface
9 channel=1
10 hw_mode=g
11 ieee80211n=1
12 ieee80211d=1
13 country_code=DE
14 wmm_enabled=1

16 #WLAN-Verschluesselung
17 auth_algs=1
18 wpa=2
19 wpa_key_mgmt=WPA-PSK
20 rsn_pairwise=CCMP
21 wpa_passphrase=raspberry

```

## 6.5 Autostart des Skriptes

Damit das Skript vor der Messung mittels SSH-Zugang gestartet werden muss, wurde das Skript in den Autostart des Raspberry Pi eingetragen. Hierdurch erfolgt der Start des Skriptes unmittelbar nach dem Hochfahren des Betriebssystems.

Listing 6.5: Startskript startVLP.sh

```

1 su pi -c "python3 VdAutoStart.py"
2 exit 0

```

zu  
Ende  
schrei-  
ben

## **7 Ausblick**

# Literaturverzeichnis

Bachfeld, Daniel (März 2013): Quadrokopter-Know-how. *c't Hacks*, (3).

Beraldin, J.-Angelo; Blais, François; Lohr, Uwe (2010): Laser Scanning Technology. In: Vosselman, George; Maas, Hans-Gerd (Hg.), Airborne and terrestrial laser scanning, S. 1 – 42, Whittles Publishing, Dunbeath, Vereinigtes Königreich, ISBN 978-1-4398-2798-7.

Ehring, Ehling; Klingbeil, Lasse; Kuhlmann, Heiner (2016): Warum UAVs und warum jetzt? In: Ehring, Ehling; Klingbeil, Lasse (Hg.), UAV 2016 – Vermessung mit unbemannten Flugsystemen, Band 82/2016, S. 9–30, DVW - Gesellschaft für Geodäsie, Geoinformation und Landmanagement e.V., ISBN 978-3-95786-067-5.

Heise Online (2017): Quadrocopter - Drohnen & Multikopter. <https://www.heise.de/thema/Quadrocopter>. (Aufruf: 27. Sep. 2017).

iMAR Navigation GmbH (2015): iNAT-M200-FLAT.

Möcker, Andrijan (28. Feb. 2017): 5 Jahre Raspberry Pi: Wie ein Platinchen die Welt eroberte. Heise Online, <https://heise.de/-3636046>. (Aufruf: 21. Sep. 2017).

Pack, Robert T.; Brooks, Valerie; Young, Jamie; Vilaça, Nuno; Vatslid, Svein; Rindle, Peter; Kurz, Sven; Parrish, Christopher E.; Craig, Rex; Smith, Philip W. (2012): An overview of ALS technology. In: Renslow, Michael S. (Hg.), Manual of airborne topographic lidar, S. 7 – 97, American Society for Photogrammetry and Remote Sensing, ISBN 1-570-83097-5.

Raspberry Pi Foundation (2017): AccessPoint. <https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md>. (Aufruf: 25. Okt. 2017).

RS Components Limited (2015): Raspberry Pi 3 Model B Datasheet. <http://docs-europe.electrocomponents.com/webdocs/14ba/0900766b814ba5fd.pdf>. (Aufruf: 21. Sep. 2017).

Schnabel, Patrick (2017): Raspberry Pi: Belegung GPIO (Banana Pi und WiringPi). Elektronik Kompendium, <https://www.elektronik-kompendium.de/sites/raspberry-pi/1907101.htm>. (Aufruf: 21. Sep. 2017).

Schulz, Jasper (2016): Aufbau und Betrieb eines Zeilenlaserscanners an einem Multikopter. <http://edoc.sub.uni-hamburg.de/hcu/volltexte/campus/2016/259/>. (Aufruf: 30. Sep. 2017).

Theis, Thomas (2011): Einstieg in Python. 3. Auflage, Galileo Press, Bonn.

ubuntuusers.de (2017): Herunterfahren. <https://wiki.ubuntuusers.de/Herunterfahren/>. (Aufruf: 22. Okt. 2017).

Velodyne Lidar (2014): VLP-16 Envelope Drawing (2D). <http://velodynelidar.com/docs/drawings/86-0101%20REV%20B1%20ENVELOPE,VLP-16.pdf>. (Aufruf: 25. Sept. 2017).

Velodyne Lidar (2016): VLP-16 User's Manual and Programming Guide.

Velodyne Lidar (2017a): HDL-32E & VLP-16 Interface Box. [http://velodynelidar.com/docs/notes/63-9259%20REV%20C%20MANUAL,INTERFACE%20BOX,HDL-32E,VLP-16,VLP-32\\_Web-S.pdf](http://velodynelidar.com/docs/notes/63-9259%20REV%20C%20MANUAL,INTERFACE%20BOX,HDL-32E,VLP-16,VLP-32_Web-S.pdf). (Aufruf: 22. Okt. 2017).

Velodyne Lidar (2017b): VLP-16 Data Sheet.

Witte, Bertold; Schmidt, Hubert (2006): Vermessungskunde und Grundlagen der Statistik für das Bauwesen. 6. Auflage, Herbert Wichmann Verlag, Heidelberg, ISBN 978-3-879-07435-8.

# Abbildungsverzeichnis

2.1	MPU-9250 - Low-Cost-MEMS-IMU-Modul wie es in vielen Consumer-Geräten und Multikoptern verwendet wird (schwarzes Bauteil mittig auf der Platine, eigene Aufnahme) . . . . .	8
3.1	Laserscanner Velodyne VLP-16 (eigene Aufnahme) . . . . .	11
3.2	iMAR iNAT-M200-Flat im Prototypen des modularen Gehäuses, Leitungen führen zu den GNSS-Antennen (eigene Aufnahme) . . . . .	12
3.3	GNSS-Antennen des (links und rechts) iMAR iNAT-M200-Flat an Prototypen des modularen Gehäuses (eigene Aufnahme) . . . . .	12
3.4	Raspberry Pi 3 (eigene Aufnahme) . . . . .	14
3.5	Multikopter Copterproject CineStar 6HL mit Gimbal Freefly MöVI M5 (eigene Aufnahme) . . . . .	15
3.6	uBlox NEO-M8N, das Vorgängermodell NEO-6M mit PPS-Ausgang wurde verwendet (eigene Aufnahme) . . . . .	17
3.7	Messung des Signals am uBlox NEO-6M (grün: Ausgangssignal; rot: Signal nach Nutzung eines Pegelwandler; 1000 Punkte entsprechen 5 Volt)	18
3.8	Schaltplan zum Anschluss des GNSS-Modules an den Laserscanner, gezeichnet in Fritzing . . . . .	18
3.9	Schaltplan für Steuerung des Raspberry, gezeichnet in Fritzing . . . . .	19
4.1	Strahlengang im Laserscanner VLP-16, Werte in Millimetern, nach Velodyne Lidar (2014) . . . . .	24
4.2	Vereinfachter Ablaufplan des Skriptes . . . . .	26
5.1	UML-Klassendiagramm . . . . .	28

# **Tabellenverzeichnis**

3.1	Spannungs- und Strombedarf der einzelnen Module (Velodyne Lidar, 2017b; iMAR Navigation GmbH, 2015; RS Components Limited, 2015)	16
4.1	Aufbau der Daten des Netzwerkpaketes, nach Velodyne Lidar (2016) . . .	22
6.1	IP-Adressen-Verteilung . . . . .	34

# **Anhang**

# A Python-Skripte

## A.1 vdAutoStart.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-

4  '''
5  @author: Florian Timm
6  @version: 2017.11.02
7  '''

9  from vdBuffer import VdBuffer
10 from vdInterface import VdInterface
11 from vdConfig import VdConfig
12 from vdTransformer import VdTransformer

14 from datetime import datetime
15 from multiprocessing import Queue, Manager
16 import multiprocessing
17 from threading import Thread
18 from flask import Flask
19 import os

21 # Pruefen, ob es sich um einen Raspberry handelt
22 try:
23     import RPi.GPIO as GPIO
24     raspberry = True
25 except ModuleNotFoundError as mne:
26     raspberry = False

30 class VdAutoStart(object):
31     '''
32     Startskript
33     '''
34     #buffer = None

36     def __init__(self):
37         '''
38         Constructor
39         '''
40         print("Datenschnittstelle fuer VLP-16\n")

42         if VdConfig.activateTransformer:
43             # Liste fuer Transformerprozesse
44             self.transformer= []
45             # Warteschlange fuer Transformer
```

```

46         self.warteschlange = Queue()

48     # Pipes fuer Prozesskommunikation erzeugen
49     manager = Manager()
50     self.gnssStatus = "unbekannt"
51     #self.gnssReady = manager.Value('gnssReady', False)
52     self.noBreak = manager.Value('noBreak', True)
53     self.scannerStatus = manager.Value('scannerStatus', "unbekannt")
54     self.datensaetze = manager.Value('datensaetze', 0)

56     # pruefe, ob root-Rechte vorhanden sind
57     try:
58         os.rename('/etc/foo', '/etc/bar')
59         self.admin = True
60     except IOError as e:
61         self.admin = False

63     def run(self):
64         ''' Starte das Programm '''

66         if raspberry:
67             print ("Raspberry Pi wurde erkannt")
68             # Hardwaresteuerung starten
69             from vdHardware import VdHardware
70             vdH = VdHardware(self)
71             vdH.start()
72         else:
73             print ("Raspberry Pi wurde nicht erkannt")
74             print ("Hardwaresteuerung wurde deaktiviert")

77         # Zeit gemaess GNSS einstellen
78         if VdConfig.aufGNSSwarten:
79             print ("Warte auf GNSS-Signal...")
80             VdInterface.getGNSSTime(ms)

82         # Uhrzeit abfragen fuer Laufzeitlaenge und Dateinamen
83         self.date = datetime.now()
84         folder = self.date.strftime(VdConfig.fileTimeFormat)

86         # Speicherordner anlegen und ausgeben
87         os.makedirs(folder)
88         print ("Speicherordner: " + folder)

90         # Speicherprozess starten
91         self.pBuffer = VdBuffer(folder, self.noBreak, self.scannerStatus,
92                               self.datensaetze, self.warteschlange, self.admin);
93         self.pBuffer.start()

95         # Transformerprozess gemaess Prozessoranzahl
96         if VdConfig.activateTransformer:
97             n = multiprocessing.cpu_count() - 1
98             if n < 2:
99                 n = 1
100            self.pTransformer = []
101            for i in range (n):
102                t = VdTransformer(self.warteschlange, i, folder, self.admin)
103                t.start()
104                self.pTransformer.append(t)

```

```

106     def aufzeichnungStarten (self):
107         self.noBreak.value = True
108         print("Aufzeichnung wird gestartet...")
109         self.scannerStatus.value = "Aufnahme gestartet"
110
111     def aufzeichnungStoppen (self):
112         self.noBreak.value = False
113         print("Aufzeichnung wird innerhalb von 5 Sekunden gestoppt")
114         os.wait(5)
115         self.pBuffer.terminate()
116         self.scannerStatus.value = "Aufnahme gestoppt"
117     def herunterfahren(self):
118         self.aufzeichnungStoppen();
119         os.system("sudo shutdown -h now")
120         print ("Fährt herunter")
121
122 # Websteuerung
123 app = Flask(__name__)
124
125 @app.route("/")
126 def webIndex():
127     timediff = datetime.now() - ms.date
128     laufzeit = str(timediff.seconds + (int(timediff.microseconds/1000)/1000.))
129     return """<html>
130     <head>
131         <title>VLP16-Datenschnittstelle</title>
132         <style>
133             table {
134                 border-collapse: collapse;
135             }
136             td {
137                 border: 1px solid black;
138             }
139         </style>
140     </head>
141     <body>
142         <h2>VLP16-Datenschnittstelle</h2>
143         <table style="">
144             <tr><td>GNSS-Status:</td><td>"""+ms.gnssStatus+"""</td></tr>
145             <tr><td>Scanner:</td><td>"""+ms.scannerStatus.value+"""</td></tr>
146             <tr><td>Datensaetze:</td>
147                 <td>"""+str(ms.datensaetze.value)+"""</td></tr>
148             <tr><td>Warteschleife:</td>
149                 <td>"""+str(ms.warteschlange.qsize())+"""</td></tr>
150             <tr><td>Laufzeit</td>
151                 <td>"""+laufzeit+"""</td></tr>
152             <tr><td colspan="2">
153                 <a href="/starten">Aufzeichnung starten</a></td></tr>
154             <tr><td colspan="2">
155                 <a href="/stoppen">Aufzeichnung stoppen</a></td></tr>
156             <tr><td colspan="2">
157                 <a href="/shutdown">Raspberry herunterfahren</a></td></tr>
158             </tr>
159         </table>
160     </body>
161 </html>"""

```

```

164  @app.route("/shutdown")
165  def webShutdown():
166      ms.herunterfahren()
167      return "Wird in 10 Sekunden heruntergefahren..."
168
169  @app.route("/stoppen")
170  def webStoppen():
171      ms.aufzeichnungStoppen()
172      return "Aufzeichnung wird gestoppt..."
173
174  @app.route("/starten")
175  def webStarten():
176      ms.noBreak.value = True
177      print("Aufzeichnung wird gestartet")
178      ms.scannerStatus.value = "Aufnahme gestartet"
179      return "Aufzeichnung wird gestartet..."
180
181  def startWeb():
182      print("Webserver startet...")
183      app.run('0.0.0.0', 8080)
184
185  if __name__ == '__main__':
186      ms = VdAutoStart()
187      t = Thread(target=startWeb)
188      t.start()
189      ms.run()

```

## A.2 vdBuffer.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.10.27
7  """
8
9  from vdInterface import VdInterface
10 from multiprocessing import Process
11 import os
12 from vdConfig import VdConfig
13
14 class VdBuffer(Process):
15     """
16     Prozess zum buffern von binaeren Dateien
17     """
18
19     def __init__(self, folder, noBreak, scannerStatus, datensaetze,
20                  warteschlange, admin):
21         """
22         Constructor
23         """
24         # Konstruktor der Elternklasse
25         Process.__init__(self)
26
27         # Pipes sichern
28         self.folder = folder

```

```

30         self.noBreak = noBreak
31         self.scannerStatus = scannerStatus
32         self.datensaetze = datensaetze
33         self.warteschlange = warteschlange
34         self.admin = admin

36         self.dateiNummer = 0

38     def run(self):
39         # Socket zum Scanner oeffnen
40         sock = VdInterface.getDataStream()
41         self.scannerStatus.value = "Socket verbunden"

43         # Variablen initialisieren
44         minibuffer = b' '
45         j = 0

48         # Prozessprioritaet hochschalten, sofern Adminrechte
49         if self.admin:
50             os.nice(-18)

52         # Dauerschleife, solange kein Unterbrechen-Befehl kommt
53         while self.noBreak.value:
54             # Daten vom Scanner holen
55             data = sock.recvfrom(1248)[0]
56             # RAM-Buffer
57             minibuffer += data
58             j += 1
59             # Alle 5 bzw. 10 Sekunden Daten speichern
60             # oder wenn Abbrechenbefehl kommt
61             if (j >= 1500*5 or not self.noBreak.value):
62                 # Datei schreiben
63                 f = open(self.folder+"/"+str(self.dateiNummer)+".bin", "wb")
64                 f.write(minibuffer)

66             f.close()

68             self.datensaetze.value += j
69             if VdConfig.activateTransformer:
70                 self.warteschlange.put(self.dateiNummer)

72             #Buffer leeren
73             minibuffer = b' '#
74             j = 0

76             # Dateizahler
77             self.dateiNummer += 1

79             if data== 'QUIT':
80                 break
81             f.close()
82             sock.close()
83             self.scannerStatus.value = "Aufnahme gestoppt"
84             print ("Verbindung getrennt")

```

## A.3 vdTransformer.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-

4  '''
5  @author: Florian Timm
6  @version: 2017.10.27
7  '''
8  from multiprocessing import Process
9  import os
10 from vdDataset import VdDataset
11 from vdFile import VdFile
12 from vdConfig import VdConfig

14 class VdTransformer(Process):
15     '''
16     Erzeugt einen Prozess zum Umwandeln von binaeren Daten des
17     Velodyne VLP-16 zu TXT-Dateien
18     '''

20     def __init__(self, warteschlange, nummer, folder, admin):
21         '''
22             Konstruktor fuer Transformer-Prozess, erbt von multiprocessing.Process
23         '''
24         # Konstruktor der Elternklasse
25         Process.__init__(self)

27         # Parameter sichern
28         self.warteschlange = warteschlange
29         self.nummer = nummer
30         self.folder = folder
31         self.admin = admin

33     def run(self):
34         '''
35             Ausfuehrung des Prozesses
36         '''

38         if self.admin:
39             os.nice(-15)
40             # Erzeugen einer TXT-Datei pro Prozess
41             v = VdFile("txt",self.folder+"/file"+str(self.nummer))

43             # Dauerschleife
44             while True:
45                 # Dateinummer aus Warteschleife abfragen und oeffnen
46                 fileno = self.warteschlange.get()
47                 f = open(self.folder+"/"+str(fileno)+".bin", "rb")

49                 # Anzahl an Datensaetzen in Datei pruefen
50                 fileSize = os.path.getsize(f.name)
51                 cntDatasets = int(fileSize/1206)

53                 for i in range(cntDatasets):
54                     # naechsten Datensatz lesen
55                     vdData = VdDataset(f.read(1206))

57                     # Daten konvertieren und speichern
58                     vdData.convertData()

```

```

60         # Datensatz zu Datei hinzufuegen
61         v.addDataset(vdData)

63         # Datei schreiben
64         v.writeTxt()
65         # Txt-Datei schliessen
66         f.close()
67         # Bin-Datei ggf. loeschen
68         if VdConfig.binNachTransLoeschen:
69             os.remove(f.name)

72     #

```

## A.4 vdInterface.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-

4  """
5  @author: Florian Timm
6  @version: 2017.11.02
7  """

9  import socket
10 import sys
11 import os
12 from datetime import datetime
13 from vdConfig import VdConfig

15 class VdInterface(object):
16     """
17     classdocs
18     """

20     @staticmethod
21     def getDataStream():
22         return VdInterface.getStream(VdConfig.UDP_IP, VdConfig.UDP_PORT_DATA)

24     @staticmethod
25     def getGNSSStream():
26         return VdInterface.getStream(VdConfig.UDP_IP, VdConfig.UDP_PORT_GNSS)

28     @staticmethod
29     def getStream(ip, port):
30         # Create Datagram Socket (UDP)
31         try :
32             sock = socket.socket(socket.AF_INET,      # Socket Family: IPv4
33                               socket.SOCK_DGRAM) # Socket Type: UDP
34             print('Socket erstellt')
35         except socket.error as msg :
36             print('Socket konnte nicht erstellt werden! Fehler ' +
37                   str(msg[0]) + ': ' + msg[1])
38             sys.exit()

40     # Sockets Options
41     sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
42     # Erlaubt, dass man sich auf einem Rechner mehrfach mit

```

```

43         # einem Port verbinden kann. (optional)
44         sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
45         # Allows broadcast UDP packets to be sent and received.

48             # Bind socket to local host and port
49             try:
50                 sock.bind((ip, port))
51             except socket.error as msg:
52                 print('Bind failed. Fehler ' + str(msg[0]) + ':' + msg[1])
53                 sys.exit()

55             print('Socket verbunden')

57             #now keep talking with the client
58             print('Listening on: ' + ip + ':' + str(port))

60             return sock

62     @staticmethod
63     def getGNSSTime(ms):
64         ms.gnssStatus = "Verbinde..."
65         sock = VdInterface.getGNSSStream()
66         print ("Warte auf GNSS-Fix")
67         while True:
68             # Daten empfangen vom Scanner
69             #print("Daten kommen...")
70             data = sock.recvfrom(2048)[0] # buffer size is 2048 bytes
71             message = data[206:278].decode('utf-8', 'replace')
72             if message[0:6] == "$GPRMC":
73                 p = message.split(",")
74                 if p[2]=="A":
75                     print("GNSS-Fix")
76                     timestamp = datetime.strptime(p[1]+"D"+p[9],
77                                         '%H%M%S.00D%d%m%y')
78                     VdInterface.setSystemZeit(timestamp)
79                     break
80                 else:
81                     print("noch kein Fix")
82                     print
83                 if data=="QUIT":
84                     break
85             sock.close()

88     @staticmethod
89     def setSystemZeit(timestamp):
90         """
91             Uhrzeit des Systems setzen
92             """
93             os.system("timedatectl set-ntp 0")
94             os.system("timedatectl set-time \" " +
95                         timestamp.strftime("%Y-%m-%d %H:%M:%S") + " \"")
96             os.system(" timedatectl set-ntp 1")

```

## A.5 vdHardware.py

```
1  #!/usr/bin/env python
```

```

2  # -*- coding: utf-8 -*-
3
4  '''
5  @author: Florian Timm
6  @version: 2017.10.27
7  '''
8
9  # Modul zur Steuerung der GPIO-Pins
10 import RPi.GPIO as GPIO
11 import time
12 from threading import Thread
13
14 class VdHardware(Thread):
15     '''
16     classdocs
17     '''
18
19     def __init__(self, masterSkript):
20         '''
21         Constructor
22         '''
23         Thread.__init__(self)
24
25         GPIO.setmode(GPIO.BEAN)
26
27         # Eingaenge aktivieren
28         GPIO.setup(18, GPIO.IN) # Aufzeichnung stoppen
29         GPIO.setup(16, GPIO.IN) # Herunterfahren
30
31         # Ausgaenge aktivieren
32         GPIO.setup(13, GPIO.OUT) # GPS-Fix
33         GPIO.setup(15, GPIO.OUT) # Laserscanner (blink)
34
35         # Ausgaenge auf Low schalten
36         GPIO.output(13, GPIO.LOW)
37         GPIO.output(15, GPIO.LOW)
38
39         # Masterobjekt sichern
40         self.masterSkript = masterSkript
41
42     def run(self):
43         # Counter fuer gedrueckt halten initialisieren
44         herunterfahren = 0
45         stoppen = 0
46
47         # pruefen, ob Modul angeschlossen ist
48         if GPIO.input(16) != GPIO.LOW or GPIO.input(18) != GPIO.LOW:
49             print("Hardwaremodul nicht erkannt")
50             self.exit()
51
52         # Dauerschleife zur Input/Output-Steuerung
53         while True:
54             # "Aufzeichnung stoppen" gedrueckt
55             if GPIO.input(18) == GPIO.HIGH:
56                 stoppen += 1
57
58             # "Herunterfahren" gedrueckt
59             if GPIO.input(16) == GPIO.HIGH:
60                 herunterfahren += 1

```

```
62         # Stoppen laenger als 2 Sekunden gedrueckt
63         if stoppen > 10:
64             self.masterSkript.aufzeichnungStoppen()
65             stoppen = 0
66
67         # Herunterfahren laenger als 2 Sekunden gedrueckt
68         if herunterfahren > 10:
69             self.masterSkript.herunterFahren()
70             herunterfahren = 0
71
72         # Pause zur Festlegung der Abtastrate
73         time.sleep(0.2)
```

## A.6 vdFile.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  '''
5  @author: Florian Timm
6  @version: 2017.10.27
7  '''
8
9  import datetime
10 from vdConfig import VdConfig
11 import math
12
13
14 class VdFile(object):
15
16     def __init__(self, fileType = "txt", fileName = ""):
17         # Dateiname erzeugen, sofern kein Dateiname mitgeliefert
18         if (fileName == ""):
19             #Jahr-Monat-TagTStunde:Minute:Sekunde an Dateinamen anhaengen
20             fileName = VdConfig.fileNamePre
21             fileName += datetime.datetime.now().strftime(VdConfig.dateFormat)
22
23             fileName += '.' + fileType
24
25         # Datei erzeugen
26         self.txtFile = open(fileName, 'w')
27         self.datasets = []
28
29
30     def write(self, data):
31         ''' Daten in Datei schreiben '''
32         self.txtFile.write(data)
33
34     def writeDatasetToTxt(self, dataset):
35         self.addDataSet(dataset)
36         self.writeTxt()
37
38
39     def writeTxt(self):
40         txt = ""
41         for ds in self.datasets:
42             for d in ds.getData():
43                 txt += str(d) + "\n"
44
45         return txt
```

```

43         if d.distanz > 0.0:
44             txt += VdConfig.fileFormatTXT(d.zeit,
45                                         d.azimut, d.vertikal, d.distanz, d.reflexion)
46         self.write(txt)
47         self.datasets = []
48
49     def addDataset(self, dataset):
50         self.datasets.append(dataset)
51
52     def writeDataToObj(self, data):
53         obj = ""
54         for p in data:
55             # Schraegstrecke zum Strahlenzentrum
56             d = p.distanz - VdConfig.beamCenter
57
58             # Vertikalwinkel in Bogenmass
59             v = p.vertikal / 180.0 * math.pi
60
61             # Azimut in Bogenmass
62             a = p.azimut / 180.0 * math.pi
63
64             # Z-Komponente
65             z = d * math.sin(v)
66
67             # Horizontalstrecke bis Drehpunkt
68             s = d * math.cos(v) + VdConfig.beamCenter
69
70             # X-Komponente
71             x = s * math.sin(a)
72
73             # Y-Komponente
74             y = s * math.cos(a)
75
76             formatsS = 'v {} {} {}\n'
77             obj += formatsS.format(x,y,z)
78         self.write(obj)
79
80
81     def close (self):
82         ''' Datei schliessen '''
83         self.txtFile.close()
84

```

## A.7 vdDataset.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.10.27
7  """
8
9  from vdConfig import VdConfig
10 from vdPoint import VdPoint
11
12 class VdDataset(object):
13     """

```

```

14     Klasse zur Repraesentation eines Datensatzes des VLP-16
15     """
16     def __init__(self, dataset):
17         """
18         Constructor
19         """
20         self.dataset = dataset
21         self.data = []
22
23
24     def getAzimut (self, block):
25
26         """ Gibt den Horizontalwinkel aus Datensatz an """
27         # Horizontalrichtung zusammensetzen, Bytereihenfolge drehen
28         azi = ord(self.dataset[VdConfig.offset[block]+
29                         2:VdConfig.offset[block]+3])
30         azi += ord(self.dataset[VdConfig.offset[block]+
31                         3:VdConfig.offset[block]+4]) << 8
32         azi /= 100.0
33         return azi
34
35     def getZeit (self):
36         """Zeit aus Datensatz mit gedrehter Bytereihenfolge zusammensetzen"""
37         zeit = ord(self.dataset[1200:1201])
38         zeit += ord(self.dataset[1201:1202]) << 8
39         zeit += ord(self.dataset[1202:1203]) << 16
40         zeit += ord(self.dataset[1203:1204]) << 24
41         return zeit
42
43     def isDualReturn(self):
44         """
45             Prueft, ob es sich um einen Datensatz mit zwei Ergebnissen
46             pro Messung (DualReturn) handelt
47         """
48         mode = ord(self.dataset[1204:1205])
49         if (mode == 57):
50             return True
51         else:
52             return False
53
54     def getAzimuts (self):
55         """Ruft alle Horizontalwinkel aus dem Datensatz ab"""
56
57         # Leere Listen erzeugen
58         azimuts = [None] * 24
59         drehung = [None] * 12
60
61         # Explizit uebermittelte Azimut-Werte einlesen
62         for j in range(0, 24, 2):
63             a = self.getAzimut(j//2)
64             azimuts[j]=a
65
66         # Drehwinkelvariable initialisieren
67         d = 0
68
69         ## DualReturn aktiv?
70         if self.isDualReturn():
71             for j in range(0, 19, 4):
72                 d2 = azimuts[j + 4] - azimuts[j]

```

```

73         if d2 < 0:
74             d2 += 360.0
75             d = d2 / 2.0
76             a = azimuts[j] + d
77             azimuts[j+1] = a
78             azimuts[j+3] = a
79             drehung[j//2] = d
80             drehung[j//2 + 1] = d
81             # Zweiten
82             drehung[10] = d
83             azimuts[21] = azimuts[20] + d
84
85         ## Strongest / Last-Return
86     else:
87         for j in range(0, 22, 2):
88             d2 = azimuts[j + 2] - azimuts[j]
89             if d2 < 0:
90                 d2 += 360.0
91                 d = d2 / 2.0
92                 a = azimuts[j] + d
93                 azimuts[j+1] = a
94                 drehung[j//2] = d
95
96         # letzter Drehwinkel wird immer vom vorherigen uebernommen,
97         # letzte Horizontalrichtung ergibt sich aus diesem
98         drehung[11] = d
99         azimuts[23] = azimuts[22] + d
100
101     # Auf Werte ueber 360 Grad pruefen
102     for j in range(24):
103         if azimuts[j] > 360.0:
104             azimuts[j] -= 360.0
105
106     return [azimuts, drehung]
107
108 def convertData(self):
109     """
110         Wandelt die Daten vom Scanner in Text um
111         @return: String mit Tab-getrennten Daten
112     """
113
114     #Zeitstempel aus den Daten auslesen
115     zeit = self.getZeit()
116
117     #Richtung und Drehwinkel auslesen
118     [azimut, drehung] = self.getAzimuts()
119
120     # Datenpaket besteht aus 12 Bloecken aus jeweils 32 Messergebnissen
121     for i in range(12):
122         versatz = VdConfig.offset[i]
123         for l in range(2):
124             for k in range(16):
125
126                 # Entfernung zusammensetzen
127                 dist = ord(self.dataset[4 + versatz:5 + versatz])
128                 dist += ord(self.dataset[5 + versatz:6 + versatz]) << 8
129                 dist /= 500.0
130
131             # Reflektivitaet auslesen

```

```

132             refl = ord(self.dataset[6 + versatz:7 + versatz])
134
135             #Offset in Daten fuer den naechsten Durchlauf
136             versatz += 3
137
138             #Horizontalwinkel interpolieren
139             a = azimut[i+1] + drehung[i] * k * VdConfig.antDrehung
140             # Punkt erzeugen und anhaengen
141             p = VdPoint(round(zeit,1), a, VdConfig.vertAngle[k],
142                         dist, refl)
143             self.data.append(p)
144             zeit += VdConfig.tZwischenStrahl
145             zeit += VdConfig.tNachlade
146
147     def getData (self):
148         return self.data

```

## A.8 vdPoint.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.10.26
7  """
8
9  class VdPoint(object):
10    """ Stellt eine Messung da """
11
12    def __init__(self, zeit, azimut, vertikal, distanz, reflexion):
13        self.zeit = zeit
14        self.azimut = azimut
15        self.vertikal = vertikal
16        self.reflexion = reflexion
17        self.distanz = distanz

```

## A.9 vdConfig.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.02
7  """
8
9  class VdConfig(object):
10    """
11    Einstellungen des Skriptes
12    """
13
14    #IP- und Porteinstellungen
15    UDP_IP = '0.0.0.0' # Net Address
16    UDP_PORT_DATA = 2368
17    UDP_PORT_GNSS = 8308      # Port

```

```

19     # Zeitgleiche Transformation zu txt aktivieren
20     activateTransformer = True

22     # Vor Aufzeichnung auf GPS warten
23     aufGNSSwarten = False

25     # Binaere Dateien nach deren Transformation loeschen
26     binNachTransLoeschen = True

29     ## Geraetekonstanten
30     # Zeit zwischen den Messungen der Einzelstrahlen
31     tZwischenStrahl = 2.304

33     # Zeit zwischen zwei Aussendungen des gleichen Messlasers
34     tRepeat = 55.296

36     # Hoehenwinkel der 16 Messstrahlen
37     vertAngle = [-15, 1, -13, -3, -11, 5, -9, 7, -7, 9, -5, 11, -3, 13, -1, 15]

39     # Bytes pro Messdatenblock
40     offsetBlock = 3 * 32 + 4
41     # Versatz vom Start fuer jeden Messblock
42     offset = list(range(0, 1206, offsetBlock))[0:12]

44     # Anteil der Zeit zwischen Einzellasern an Wiederholungszeit ,
45     # fuer Interpolation des Horizontalwinkels
46     antDrehung = tZwischenStrahl / tRepeat

48     # Zeit nach letztem Strahl bis zum naechsten
49     tNachlade = tRepeat - 15 * tZwischenStrahl

51     # Dateipraefix der zu speichernden Datei
52     fileNamePre = 'data'

54     # Format der Zeit am Dateinamen
55     fileTimeFormat = '%Y-%m-%dT%H:%M:%S'

57     # Abstand des Strahlenzentrums von der Drehachse
58     beamCenter = 0.04191

60     @staticmethod
61     def fileFormatTXT (zeit, azimut, vertikal, distanz, reflexion):
62         ''' Einstellung fuer das Erzeugen der TXT-Datei '''
63         azimut = round(azimut, 3)
64         distanz = round(distanz, 3)
65         formatsS = '{}\t{}\t{}\t{}\t{}\n'
66         return formatsS.format(zeit, azimut, vertikal, distanz, reflexion)

```

## A.10 convTxt2Obj.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-

4  '''
5  @author: Florian Timm
6  @version: 2017.10.22

```

```

7  """
8  from vdPoint import VdPoint
9  from vdFile import VdFile
10
11 class ConvTxt2Obj:
12     def __init__(self):
13         self.fileName = "test.txt"
14
15     def run(self):
16         data = []
17         txt = open(self.fileName, 'rb')
18         for line in txt:
19             l = line.split()
20             data.append(VdPoint(float(l[0]), float(l[1]), float(l[2]),
21                                 float(l[3]), int(l[4])))
22             #print ("test")
23         f = VdFile("obj", self.fileName)
24         f.writeDataToObj(data)
25
26
27 if __name__ == '__main__':
28     ConvTxt2Obj().run()

```

# B Beispieldateien

## B.1 Rohdaten vom Scanner

## B.2 Dateiformat für Datenspeicherung als Text

```
1 210862488 36.18 -15 2.234 46
2 210862490.304 36.188 1 2.18 46
3 210862492.60799998 36.197 -13 2.214 41
4 210862494.91199997 36.205 -3 2.16 42
5 210862497.21599996 36.213 -11 2.204 50
6 210862499.51999995 36.222 5 2.164 55
7 210862501.82399994 36.23 -9 2.184 47
8 210862504.12799993 36.238 7 2.17 42
9 210862506.43199992 36.247 -7 2.192 43
10 210862508.7359999 36.255 9 2.21 40
11 210862511.0399999 36.263 -5 2.186 41
12 210862513.3439999 36.272 11 2.206 46
13 210862515.64799988 36.28 -3 2.182 47
14 210862517.95199987 36.288 13 2.192 42
15 210862520.25599986 36.297 -1 2.16 43
16 210862522.55999985 36.305 15 2.214 47
17 210862545.59999985 36.38 -15 2.224 45
18 210862547.90399984 36.388 1 2.168 48
19 210862550.20799983 36.397 -13 2.192 39
20 210862552.51199982 36.405 -3 2.152 44
```

## B.3 Dateiformat für Datenspeicherung als OBJ

```
1 v 1.2746902491848617 1.7429195788236858 -0.5673546405787847
2 v 1.2869596160904488 1.7591802795096634 0.037314815679491506
3 v 1.274630423722104 1.741752967944279 -0.48861393562976563
4 v 1.274145749563782 1.7405806715041912 -0.1108522655586169
5 v 1.2786300911436552 1.7461950253652434 -0.41254622081367376
6 v 1.2739693304356217 1.7392567142322626 0.1849523301273779
7 v 1.2752183957072614 1.7404521494414935 -0.33509670321802815
8 v 1.2733984465128143 1.7374593339109177 0.25934893100706025
9 v 1.2865822747749043 1.7548695003015347 -0.26203005656197353
10 v 1.291164267762529 1.7606036538453675 0.33916399930907415
11 v 1.2881769097946472 1.756015963302377 -0.1868697564678264
12 v 1.2815890463056323 1.7464602478174986 0.4129278388044268
13 v 1.2894233312788135 1.7566219901831923 -0.11200365659596165
14 v 1.264708293723956 1.7224476905470605 0.4836650124342007
15 v 1.2784663490171557 1.7406120436549135 -0.036965767550745834
16 v 1.2670514982720475 1.7245661497369091 0.5621782596767342
17 v 1.2750371359697306 1.730682783609054 -0.5647664501277595
18 v 1.2859745413187607 1.7450184890932041 0.0371053868022441
```

19 v 1.267982802947427 1.7200385827982603 -0.4836650124342007  
20 v 1.2754723412692703 1.729692556621396 -0.11043357790867334

## **Erklärung**

Hiermit versichere ich, dass ich die beiliegende Bachelor-Thesis ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe.

Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Hamburg, den 12. Dez. 2017

---

Ort, Datum

Florian Timm