

# **Entwurf und Implementation einer Daten-Schnittstelle zum Betrieb des Laserscanners 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 Laser-scanners. 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 Laserscanings</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 . .	7
2.3 Inertiale Messeinheit . . . . .	7
2.4 Kombination des Messsystems . . . . .	9
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 . . . . .	15
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>23</b>
4.1 Verwendung von Python . . . . .	23

4.2	Datenlieferung vom Laserscanner . . . . .	23
4.3	Geplantes Datenmodell . . . . .	24
4.4	Weiterverarbeitung der Daten zu Koordinaten . . . . .	25
4.5	Anforderungen an das Skript . . . . .	26
<b>5</b>	<b>Entwicklung des Skriptes</b>	<b>29</b>
5.1	Klassenentwurf . . . . .	29
5.2	Evaluation einzelner Methoden . . . . .	29
5.3	Multikern-Verarbeitung der Daten . . . . .	31
5.4	Klassen . . . . .	32
5.4.1	VdAutoStart . . . . .	32
5.4.2	VdInterface . . . . .	33
5.4.3	VdHardware . . . . .	33
5.4.4	VdPoint . . . . .	33
5.4.5	VdDataset . . . . .	33
5.4.6	VdFile und Subklassen . . . . .	34
5.4.7	VdBuffer . . . . .	34
5.4.8	VdTransformer . . . . .	35
5.5	Steuerung des Skriptes . . . . .	35
5.6	Beispiel-Quelltext-Zitat . . . . .	35
<b>6</b>	<b>Konfiguration des Raspberry Pi</b>	<b>36</b>
6.1	Installation von Raspbian . . . . .	36
6.2	Befehle mit Root-Rechten . . . . .	37
6.3	IP-Adressen-Konfiguration . . . . .	37
6.4	Konfiguration als WLAN-Access-Point . . . . .	38
6.5	Autostart des Skriptes . . . . .	39
<b>7</b>	<b>Systemüberprüfungen</b>	<b>40</b>
7.1	Untersuchung der Gleichzeitigkeit von PPS-Signalen von verschiedenen GNSS-Empfängern . . . . .	40
7.2	Messgenauigkeit des Laserscanners im Vergleich . . . . .	40
7.2.1	Vergleichsmessung mittels Fassadenfront . . . . .	41
<b>8</b>	<b>Ausblick</b>	<b>43</b>
<b>Literaturverzeichnis</b>		<b>44</b>
<b>Abbildungsverzeichnis</b>		<b>46</b>
<b>Tabellenverzeichnis</b>		<b>47</b>

<b>Anhang</b>	<b>48</b>
<b>A Python-Skripte</b>	<b>49</b>
A.1 vdAutoStart.py . . . . .	49
A.2 vdBuffer.py . . . . .	57
A.3 vdTransformer.py . . . . .	59
A.4 vdInterface.py . . . . .	61
A.5 vdGNSSTime.py . . . . .	63
A.6 vdHardware.py . . . . .	65
A.7 vdFile.py . . . . .	68
A.8 vdDataset.py . . . . .	73
A.9 vdPoint.py . . . . .	76
A.10 config.ini . . . . .	79
A.11 convTxt2Obj.py . . . . .	80
A.12 convBin2Obj.py . . . . .	80
<b>B Beispieldateien</b>	<b>82</b>
B.1 Rohdaten vom Scanner . . . . .	82
B.2 Dateiformat für Datenspeicherung als Text . . . . .	82
B.3 Dateiformat für Datenspeicherung als OBJ . . . . .	83

# 1 Einleitung

## 1.1 Problemstellung

Daten aus Airborne Laserscanning, dem Abtasten von Oberflächen mit einem Laser-scanner 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 Laserscanners Helikopter oder Flugzeuge verwendet, die mit entsprechender Sensorik ausgerüstet sind. Diese Messmethode lohnt sich allerdings nicht für die Vermessung kleinerer 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 sehr hoch, so dass sich eine solche Messung oft nur für sehr große Gebiete lohnt. Alternativ bietet sich die terrestrische Messung mittels Tachymeter oder auch per Laserscanner um kleinere Gebiete abzubilden an – 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, die Multikopter nutzen, heutzutage meist auf photogrammetrischen Prinzipien, welche Luftbilder zur Erfassung nutzen. Hierzu muss jedoch ausreichend Beleuchtung vorhanden sein, welches wiederum die Einsetzbarkeit des Systems 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 Laserscannings erläutert werden. Außerdem wird die benötigte Hardware zur Durchführung eines solchen Laserscannings 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). Einzelne Komponenten werden in Kapitel 7 auf ihre Genauigkeit und Zuverlässigkeit geprüft. Zum Abschluss soll in Kapitel 8 noch ein Einblick in die Zukunft des Systems geworfen werden.

# 2 Grundlagen des Airborne Laserscannings

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 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 Navigationssatellitenempfänger (siehe Abschnitt 2.2). Auf diese Bestandteile wird im Folgenden eingegangen. 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 die Messung von Entfernungen mittels Laserscanners gibt es zwei meistgenutzte Verfahren:

**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 Laufzeit lässt sich die zurückgelegte Strecke des Lichtstrahles und somit die doppelte Entfernung zu der Oberfläche bestimmen. Hierzu wird der Brechungsindex  $n$  des vom Laser durchlaufenen Mediums

benötigt. Bei der Messung in der Luft lässt sich dieser aus den Daten von Temperatur-, Druck- und Luftfeuchtemessung 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 Sinuswelle 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 wird normalerweise Infrarotlicht verwendet, da dieses gute Ausbreitungseigenschaften hat. Die hierfür benötigte Modulationswelle wird durch einen 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 Phasenverschiebungsmessung begrenzt ist, wird durch Nutzung verschiedener Wellenlängen eine Genauigkeitssteigerung durchgeführt. 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 die genaue Zeitmessung ein Problem. 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 verschiedene 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 in Richtung der Schwingachse bewegt. Es entsteht eine Zick-Zack-Linie auf der Oberfläche 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.

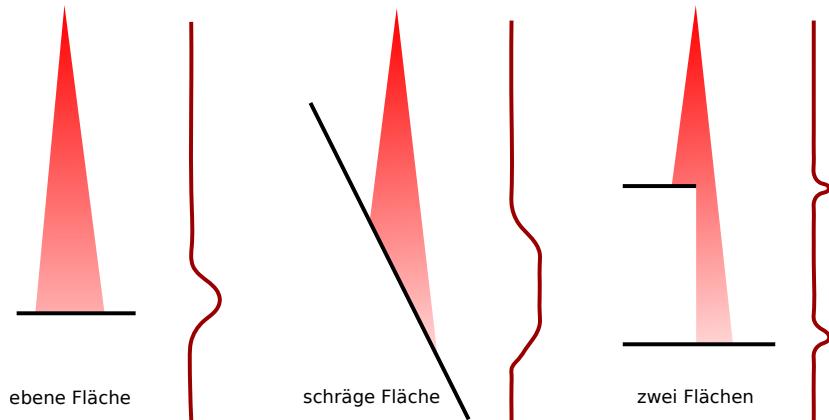


Abbildung 2.1: Reflektiertes Signal nach Oberfläche, nach Beraldin et al. (2010, S. 28)

**Zusätzliche Achsen** Zusätzlich zu den Spiegelmechanismen verfügen die terrestrischen Laserscanner über eine weitere Achse. Während beim Airborne Laserscanning die weitere Bewegung des Lasers durch die Fortbewegung des Fluggerätes durchgeführt wird, muss dies bei der terrestrischen Messung 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. Abbildung 2.1 zeigt die Echos in grafischer Form. (Beraldin et al., 2010, S. 28)

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)

## **2.2 Positionsbestimmung mittels globalen Navigationssatellitensystemen**

Zur Bestimmung der Position des Fluggerätes wird ein Empfänger für globale Navigationssatellitensysteme (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 befindliche 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 Funkignal aufmodellierte Codemuster ausgewertet, sondern auch die Phase des Signals. Hierdurch ist es auch möglich, dass verschlüsselte L2-Signal mitzunutzen. Durch die Nutzung von Referenzstationsnetzen wie SAPOS können 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 verwindungssteif 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

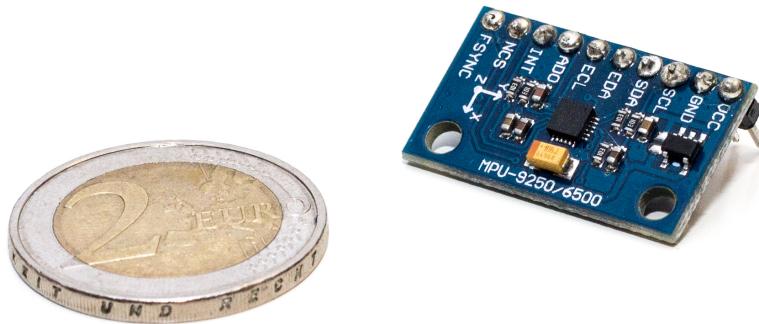


Abbildung 2.2: MPU-9250 - Low-Cost-MEMS-IMU-Modul wie es in vielen Consumer-Geräten und Multikoptern verwendet wird (schwarzes Bauteil mittig auf der Platine, eigene Aufnahme)

Messeinheit ist ihre Messfrequenz: Im Gegensatz zum GNSS, dass im Normalfall nur eine Messung pro Sekunde durchführt, 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 zueinander 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 ihre zeitliche Abweichung unterteilt. Die zeitliche Stabilität ist vor allem bei der Inertialnavigation, die ausschließlich auf deren Messungen basiert, wichtig.

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 2.2), 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 Gimbals 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 Kufen 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 Navigationssatellitensystems (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 wahrscheinlich um einen Faserscanner (siehe Abschnitt 2.1.2) mit 16 Messstrahlen, der sich zusätzlich um seine Hochachse dreht. Genaue Angaben macht der Hersteller hierzu keine. Seine Messgenauigkeit beträgt laut Datenblatt 3 *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  $2,3\mu s$  hintereinander aus, gefolgt von einer Nachladezeit von  $18,4\mu s$ , so dass jeder Messstrahl alle  $55,3 \mu s$  ausgesendet werden kann (Velodyne Lidar, 2016, S. 16). Es ergibt sich somit eine durchschnittliche Messfrequenz von  $289.357 Hz$  (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 Zeitstern-



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

pel zu versehen und die Messungen des Scanners so in der Nachbearbeitung mit den Daten aus der inertialen Messeinheit zu verknüpfen.

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

$$n = 289.357 \frac{\text{Messung}}{\text{Sekunde}} \cdot 2 \frac{\text{Messwerte}}{\text{Messtrahl}} = 578.714 \frac{\text{Messwerte}}{\text{Sekunde}}$$
(3.1)

### 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. Wilken (2017) Durch die Verwendung von zwei Empfängern, die an jeweils einem Ausleger befestigt sind (siehe Bild Abbildung 3.3), kann auch die Orientierung des Scanners bestimmt werden. Hierdurch wird die ungenaue Messung des magnetischen Nordpols

mehr  
Daten

alternativ:  
Matt-  
hias  
Wil-  
kens



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

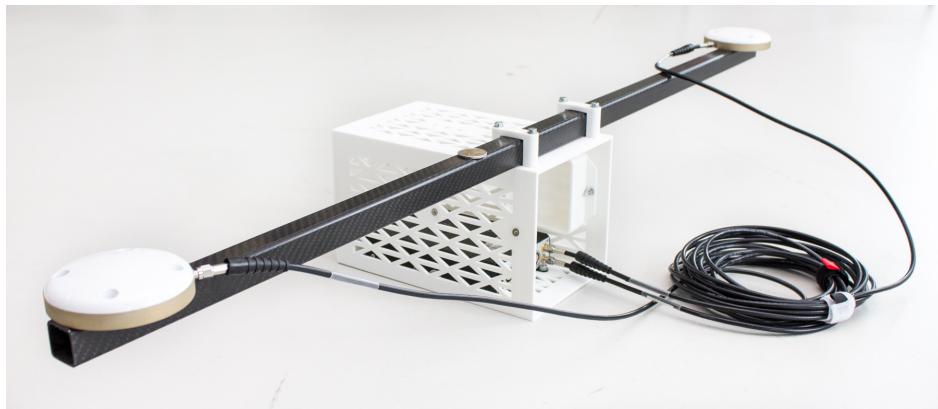


Abbildung 3.3: GNSS-Antennen des (links und rechts) iMAR iNAT-M200-Flat an Prototypen des modularen Gehäuses (eigene Aufnahme)

überflüssig. Außerdem kann die Positionssicherheit durch Mittlung der beiden Positionen erhöht werden.

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-Systemen des Halbleiterherstellers Broadcom. In diesem Chip sind die wichtigsten Bauteile des Systems 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 (Schnabel, 2017)
  - 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 für die Nutzung am Multikopter ist seine geringe Größe und sein relativ geringer Stromverbrauch von maximal 12,5 Watt (RS Components Limited, 2015), welche aber im Betrieb ohne Peripherie nicht erreicht wird.

### **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



Abbildung 3.4: Raspberry Pi 3 (eigene Aufnahme)

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 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. Der verwendete Hexakopter (siehe Abbildung 3.5) hat eine Tragkraft von maximal 5 Kilogramm und eine Flugdauer von bis zu 20 Minuten (Schulz, 2016).



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

### 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 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

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)

MicroSD-Karten- und der USB-Schnittstelle auch die komplette benötigte Hardware, die so nicht einzeln zusammengestellt und -gebaut werden muss.

### 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 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.

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

### 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 selbst-

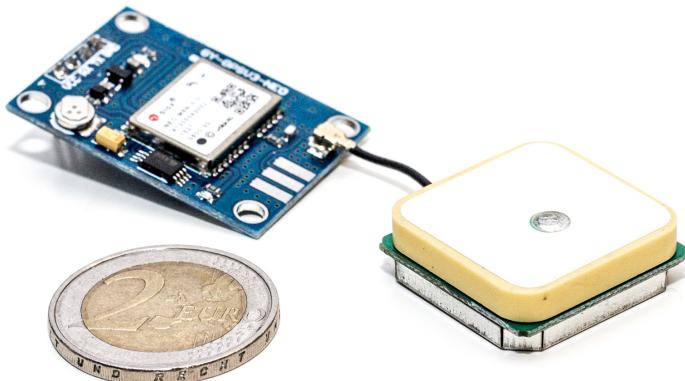


Abbildung 3.6: uBlox NEO-M8N, das Vorgängermodell NEO-6M mit PPS-Ausgang wurde verwendet (eigene Aufnahme)

ständig 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.

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  $9600 \frac{\text{bit}}{\text{s}}$  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 Arduino (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



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

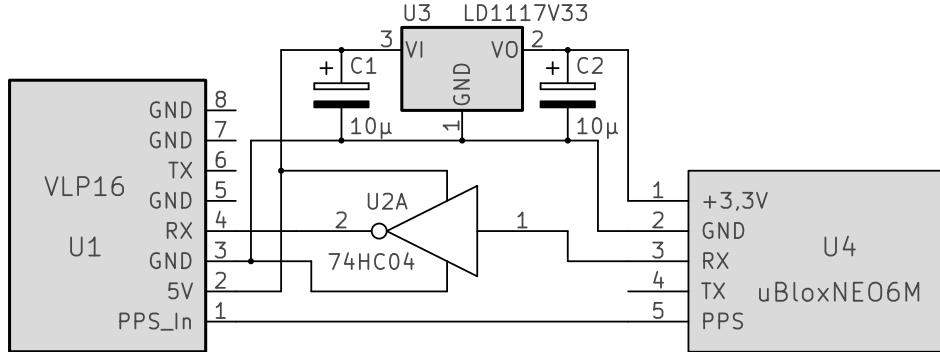


Abbildung 3.8: Entwurf der Schaltung zum Anschluss des GNSS-Modules an den Laser-scanner

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-Modull nur mit 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.

## **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.

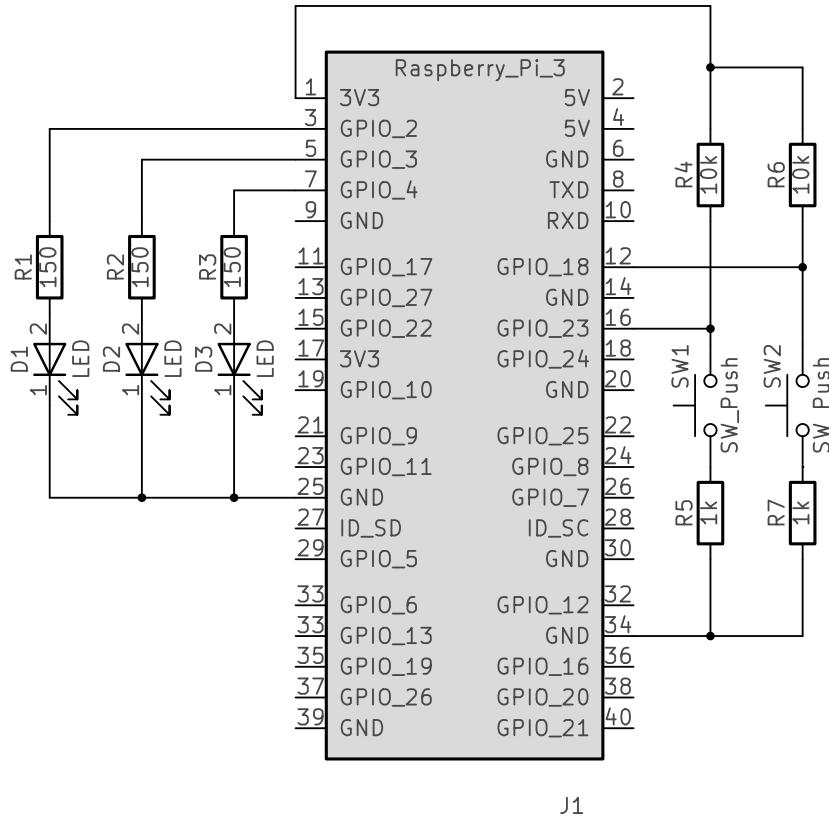


Abbildung 3.9: Entwurf der Schaltung für die Steuerung des Raspberry Pi

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 18 und 25 erreichbar. Ohne Betätigung werden die Eingänge über die Pull-Up-Widerstände () auf ein High-Level gezogen. Durch Drücken des Tasters wird die Spannung über die Widerstände auf ein Low-Level gezogen, welches durch das Python Skript zur Laufzeit ausgelesen werden kann. Der Widerstand dient zur Strombegrenzung und als Sicherheit, falls die GPIO-Pins falsch geschaltet werden. In der weiteren Optimierung der Schaltung wurden die externen Pull-Up-Widerstände durch die interne, schaltbaren Pull-Up-Widerstände ersetzt. Dieses erwies sich vor allem bei der Nutzung ohne angestecktes Steuerungsmodul als vorteilhaft, da so auch ohne Modul diese immer geschaltet sein können. Ansonsten könnten im Betrieb ohne Steuerungsmodul auftretende Spannungsschwankungen fehlerhafterweise zu einer Erkennung als Tastendruck führen. Die drei Leuchtdioden wurden mit jeweils einem 150 Ohm Vorwiderstand direkt zwischen einem 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

R?

Widerstände

R?

10mA bei 3,3 Volt gesprochen (zum Beispiel Schnabel (2017)). Alternativ, bei Nutzung leistungsstärkerer LEDs könnten diese auch unter Nutzung eines Transistors geschaltet werden. Diese lassen mit einem geringen Steuerungsstrom höhere Ströme schalten.

$$U_R = U_{GPIO} - U_{LED} = 3,3V - 2,0V = 1,3V \quad | \text{ Benötigter Spannungsabfall}$$

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

## 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 verlöten Bauteilen erstellt werden. Vorteile der gelöteten 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

Da nur wenige Bauteile benötigt wurden, wurde eine Streifenrasterplatine gewählt. Bei einer solchen Platine sind alle Kontakte in einer Reihe mit einer Leiterbahn verbunden. Falls keine Verbindung gewünscht ist, kann diese Leiterbahn mit einem Messer oder ähnlichem unterbrochen werden. Da das Unterbrechen der Leiterbahn jedoch zeitaufwändig und fehlerträchtig ist, beispielsweise durch nicht vollständig getrennte Leiterbahnen, sollten diese beim Layouten der Platine möglichst vermieden werden. Auch sollte möglichst viele der benötigten Verbindungen durch diese Leiterbahnen erfolgen

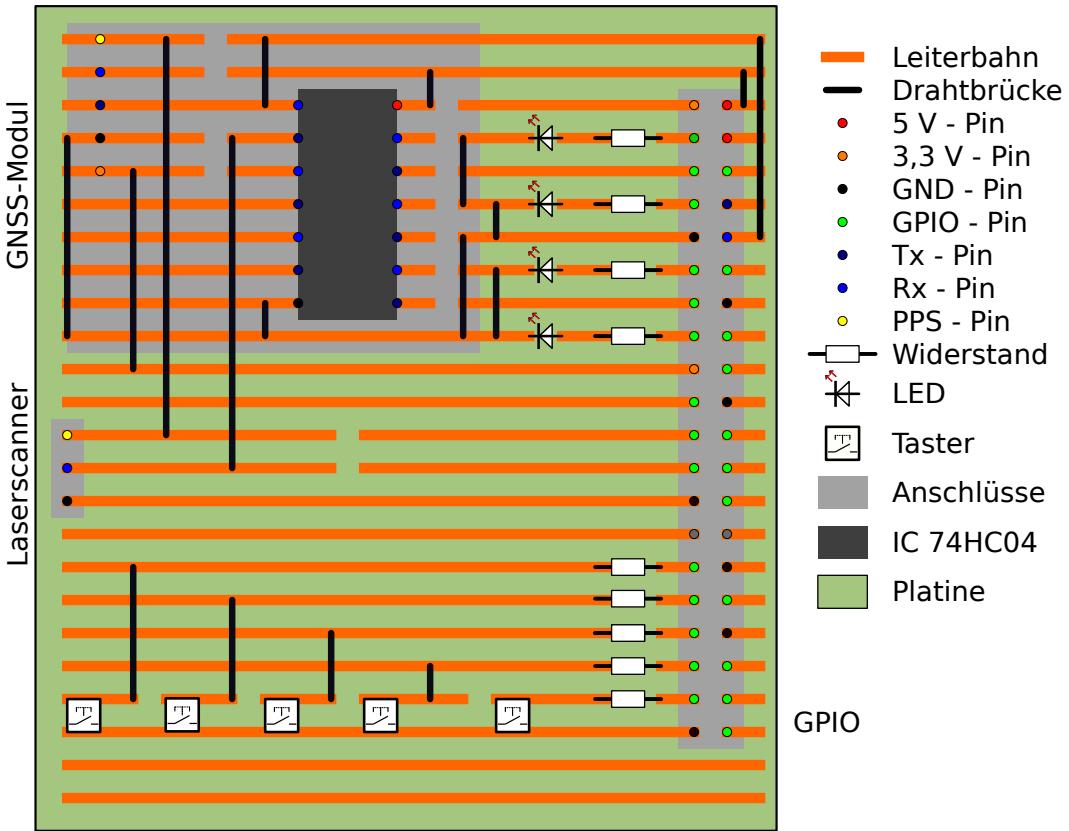


Abbildung 3.10: Layout der Lochstreifenplatine

und möglichst wenig Drahtbrücken verwendet werden, die diese Leiterbahnreihen verbinden, da diese zusätzlichen Lötaufwand erfordern. Das endgültige Layout der Platine ist der Abbildung 3.10 zu entnehmen.

Beim Routing wurden noch einige Teile der Schaltung optimiert und versucht, einige Bauteile einzusparen, in dem zum Beispiel die Stromversorgung vom Raspberry Pi für den integrierten Schaltkreis und das GNSS-Modul verwendet wurden. Außerdem wurde die Auswahl der GPIO-Pins des Raspberry Pi platzsparender optimiert und nur die auch an dem ersten Typ des Raspberry Pi vorhandenen PINs genutzt. Hierdurch ist die Schaltung abwärtskompatibel zu allen Versionen des Raspberry Pi. Der endgültige Schaltplan ist Abbildung 3.11 zu entnehmen. Für die Taster wurden hier die internen Pull-Up-Widerstände genutzt, so dass hier zwei Widerstände eingespart werden konnten. Außerdem wurde der Datensendeport (Tx) vom GNSS-Modul an die serielle Schnittstelle des Raspberry Pi angeschlossen, so dass der Raspberry Pi nun auch ohne den Umweg über den Laserscanner die Daten vom GNSS-Modul empfangen kann.

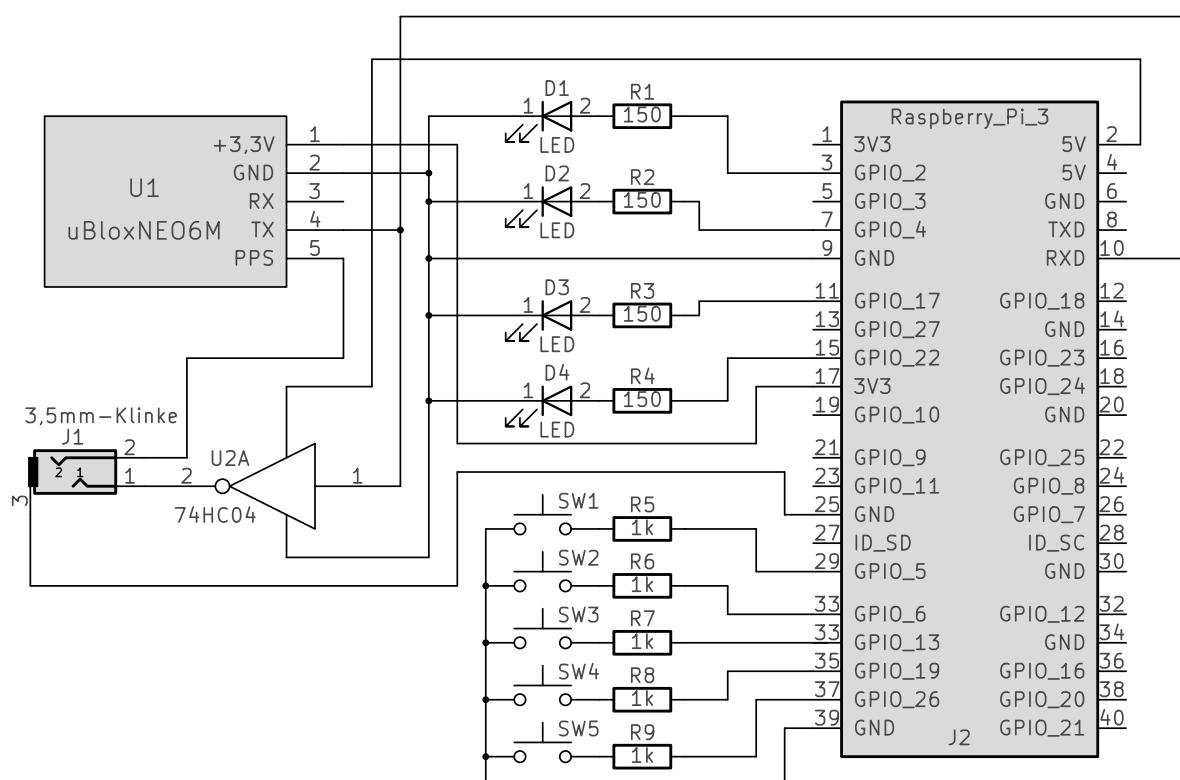


Abbildung 3.11: Endgültiger Schaltplan

# 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 lösendem 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

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)

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 aufeinander folgenden Messungen interpoliert werden.

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 georeferenzieren 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 Tests 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.

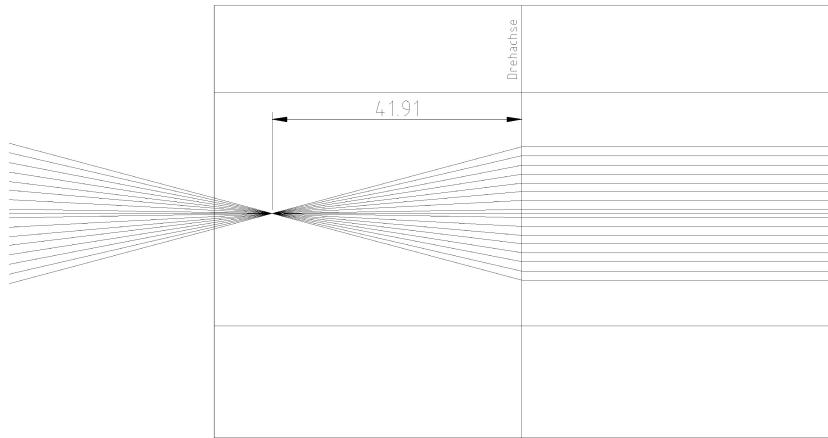


Abbildung 4.1: Strahlengang im Laserscanner VLP-16, Werte in Millimetern, nach Velodyne Lidar (2014)

$h$  : 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.

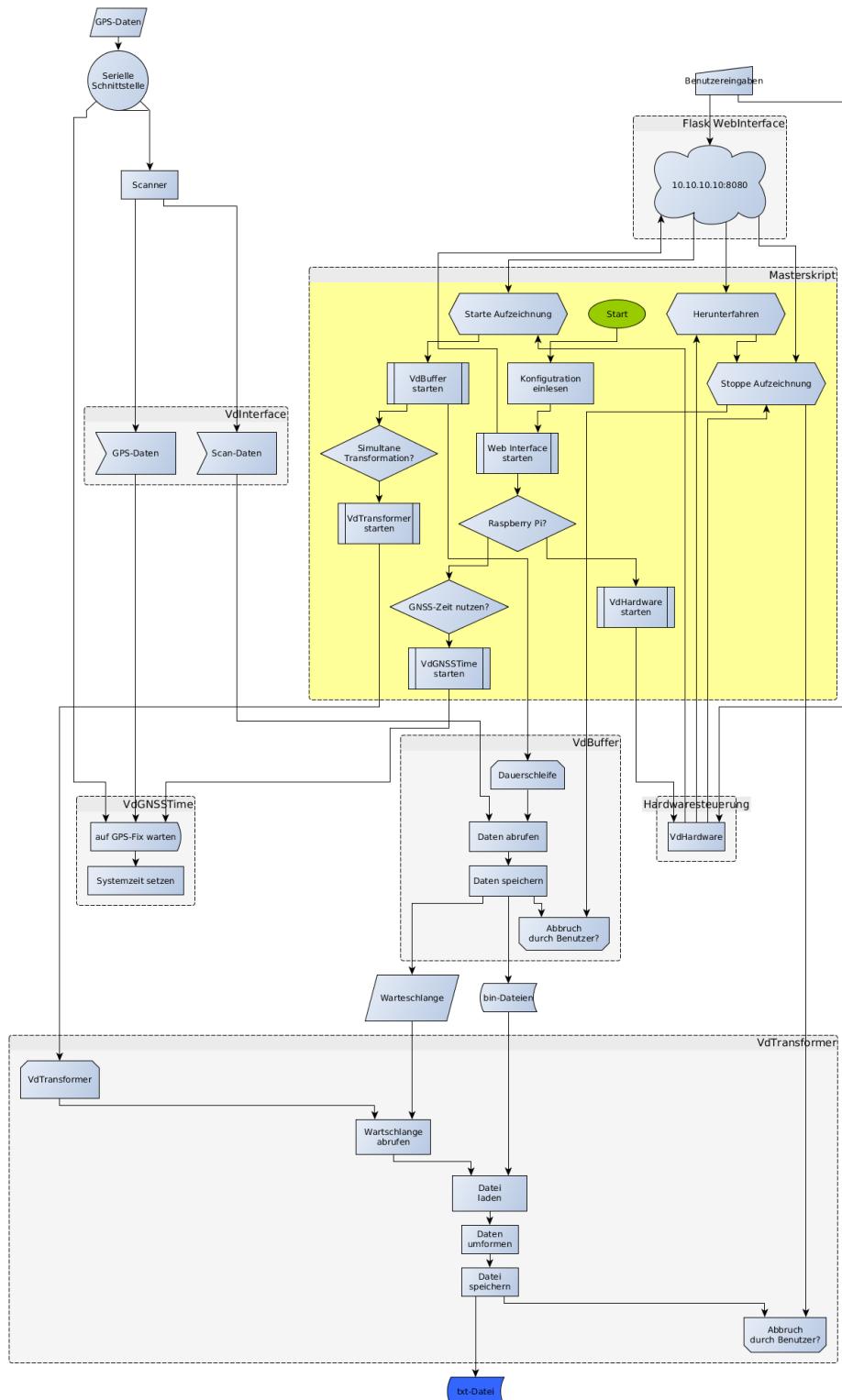


Abbildung 4.2: Vereinfachter Ablaufplan des Skriptes

# 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 5.4 eingegangen.

## 5.2 Evaluation einzelner Methoden

Um eine einfache Fehlersuche zu ermöglichen, wurden die grundlegenden Funktionen in einzelnen Skripten entwickelt und geprüft. Diese kleineren Skripte haben den Vorteil, dass Fehler schneller eingegrenzt und auch schon früh konzeptionelle Fehler entdeckt werden können. In diesem Schritt wurde bemerkt, dass 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 das Fünffache der Mess- und Aufzeichnungzeit beanspruchte. Wie erwartet, brachte auch das direkte Einlesen der Daten vom Scanner keinen Erfolg. Es folgte ein Überlauf des Netzwerk-Buffers 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 daher von dem Datenempfang entkoppelt

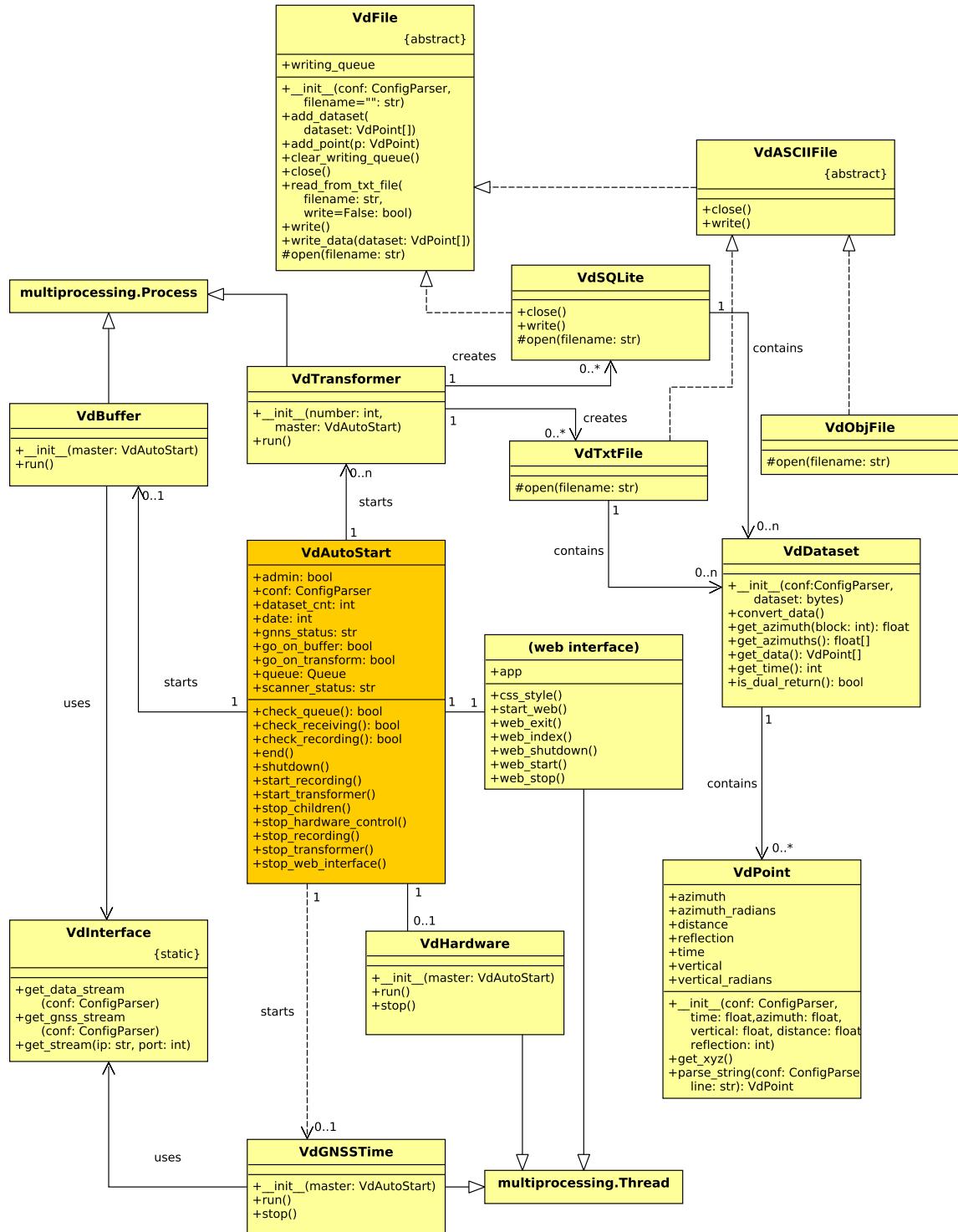


Abbildung 5.1: UML-Klassendiagramm

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 automatisch mit dem Start des Raspberry Pi gestartet wurde und das Steuermodul 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 zuerst eine vorherige Abfrage hinzugefügt, die beim Start überprüft, ob die beiden Taster sich auf einem Low-Niveau befinden, dass durch die beiden angeschlossenen Pull-Down-Widerstände erreicht wird. Falls dieses nicht der Fall ist, beendet sich die Hardwaresteuerung selbstständig. Im weiteren Verlauf der Entwicklung wurden dann jedoch das Signal gedreht und die internen Pull-Down-Widerstände des Raspberry Pi verwendet. Somit wurde diese Abfrage überflüssig.

## 5.3 Multikern-Verarbeitung der Daten

Da bei der Evaluation der einzelnen Methoden 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 Verzögerung. 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 Laser-scanners 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 binären 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 dieses Prinzip stört eine stockende Datenumformung nicht die Aufzeichnung 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 **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. [erweitern](#)

**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. Die Weboberfläche ist entsprechend ihrem geplanten Einsatzzweck optimiert für die mobile Anzeige auf Smartphones, lässt sich aber auch vom Laptop bedienen. Die Oberfläche selbst nutzt nur HTML und CSS - ist also nicht zusätzlichen Skriptsprachen auf dem verwendeten Gerät abhängig.

### 5.4.2 VdInterface

Die Klasse `VdInterface` (siehe Anhang A.4) übernimmt die Kommunikation mit dem Laserscanner. Sie stellt die UDP-Socket-Verbindungen zum Datenstream des Scanners her.

zu kurz

### 5.4.3 VdHardware

Die Klasse `VdHardware` übernimmt die Hardwaresteuerung des Raspberry Pi. Um etwas unabhängiger zu sein, stellt die Klasse einen eigenen Thread dar, der von der Klasse `VdAutoStart` je nach Hardware gestartet wird. Hierdurch werden die Abläufe des Hauptskriptes nicht durch die Abfrageschleifen der Hardwaresteuerung unterbrochen.

Bei der Initialisierung der Klasse werden die GPIO-Ports des Raspberry Pi entsprechend des Hardwaresteuerungsmodules aus Abschnitt 3.6 eingerichtet. Hierbei werden bei den Eingangssports die internen Pull-Up-Widerstände aktiviert. Beim Start des Threads wird dann zusätzlich ein Eventhandler für die Eingangspins eingerichtet, der entsprechende Funktionen zum Starten oder Stoppen der Aufzeichnung sowie zum Herunterfahren aufrufen. Des Weiteren wird ein Timer aktiviert, der dafür sorgt, dass die LED-Anzeigen einmal sekündlich aktualisiert werden.

### 5.4.4 VdPoint

Die `VdPoint`-Klasse stellt einen Messpunkt der Velodyne dar. Er nimmt als Attribute die Messdaten auf. Außerdem bietet die Klasse die Möglichkeit, die Messdaten in lokale kartesische Koordinaten mit dem Laserscanner als Ursprung umzurechnen. Dieses wird zum Beispiel bei der Erzeugung von OBJ-Dateien genutzt.

ergänzung

### 5.4.5 VdDataset

Die Klasse `VdDataset` nimmt die Messdaten auf. Diese Klasse sorgt auch für das Interpretieren der binären Daten vom Laserscanner. Die Daten werden dann als `VdPoint`-Objekte in einer Liste gesichert. Durch die Übergabe der Daten an die Implementa-

tierungen der Klasse `VdFile` können diese dann zum Beispiel als Datei gespeichert werden.

### 5.4.6 `VdFile` und Subklassen

Bei der Klasse `VdFile` handelt es sich um eine abstrakte Klasse. Die Klasse stellt ein Interface da, das die Speicherung der umgewandelten Daten übernimmt. Die genauen Datenformate werden in Klassen festgelegt, die von ihr erben:

**`VdASCIIFile` (`VdTxtFile` / `VdObjFile` / `VdXYZFile`)** Die abstrakte Klasse `VdASCIIFile` erweitert die Klasse `VdFile` um Funktionen, um ASCII-Dateien zu schreiben. `VdTxtFile` und `VdObjFile` implementieren dann die eigentlichen Dateiformate. Die Klasse `VdTxtFile` schreibt hierbei die Rohdaten als einfache Textdateien. `VdObjFile` formt die Rohdaten in lokale Koordinaten mit dem Ursprung im Standpunkt des Scanners um und speichert die Daten als OBJ-Datei. Das Dateiformat ist für das einfache Betrachten der Daten hilfreich. Es lässt sich als Punktfolge in vielen Programmen, wie zum Beispiel `MeshLab`, einlesen und darstellen. Im Zusammenhang mit der Vergleichsmessung aus Kapitel 7 wurde zusätzlich noch eine Klasse zur Implementierung von XYZ-Dateien erstellt. Hier werden die schon für die OBJ-Dateien aufbereiteten Daten in anderer Formatierung genutzt.

**`VdSQLite`** Die Klasse `VdSQLite` ermöglicht das Speichern der Rohdaten in einer SQLite-Datenbank-Datei. Das Format bietet sich zum einfachen Übertragen und Auswerten der Daten an. Alle Daten werden in einer Datenbank-Datei gesichert. Dies erleichtert das Einladen der Daten in weitere Skripte, zum Beispiel zum Durchführen von Berechnungen.

### 5.4.7 `VdBuffer`

Die Klasse `VdBuffer` übernimmt die Zwischenspeicherung der binären Rohdaten vom Laserscanner. Über die Klasse `VdInterface` wird eine Socketverbindung zum UDP-Port des Laserscanners hergestellt. Anschließend werden die Daten vom Socket in einer Variable gespeichert und regelmäßig in kurzen Intervallen als bin-Datei auf dem Speicher des Raspberry Pi abgelegt. Hierzu wird beim ersten Datenempfang ein Verzeichnis für die Messung angelegt. Nach dem Schreiben der Datei wird ihr Pfad in die Warteschleife für `VdTransformer` eingetragen.

## 5.4.8 VdTransformer

`VdTransformer` übernimmt die Umformung der als Datei gespeicherten binären Daten in `VdPoint`-Objekte. Um die Komponenten des Computersystems auszunutzen, werden durch `VdAutoStart` eine Instanz des Prozesses weniger gestartet, wie Prozessorkerne vorhanden sind - also auf dem Raspberry Pi als Vierkern-System 3 Prozesse der Klasse `VdTransformer`. Sofern die Daten als ASCII-File gesichert werden sollen, wird für jede Instanz eine Datei erstellt, damit sich die Schreibprozesse nicht gegenseitig stören. Aus der Warteschlange wird jeweils eine bin-Datei ausgewählt, eingelesen und an die Klasse `VdDataset` übermittelt. Nachdem diese die Daten zu einer Liste von `VdPoint` umgewandelt hat, werden diese Daten an die Klasse `VdFile` übergeben. Diese schreibt dann je nach Einstellung in der Konfigurationsdatei die Daten im gewünschten Dateiformat. Nach dem erfolgreichen Schreibvorgang, wird die bin-Datei gelöscht (oder je nach Einstellungen verschoben).

## 5.5 Steuerung des Skriptes

Die grundlegenden Einstellungen erfolgen über die Konfigurationsdatei `config.ini`. Hier werden \_\_\_\_\_

weiter

## 5.6 Beispiel-Quelltext-Zitat

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

Listing 5.1: Quelltext-Test

```
16  class VdFile(ABC):  
18      """ abstract class for saving data """  
20      def __init__(self, conf, filename=""):
```

# 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 erfolgten mit dem vorinstallierten Editor `nano` unter Nutzung von Administratorrechten. 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 wurden 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 Pi erfolgt in der Konfigurationsdatei `/etc/network/interfaces` (siehe Listing 6.2. Raspberry Pi Foundation (2017))

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 allow-hotplug wlan0
13 iface wlan0 inet static
14   address 10.10.10.10
15   netmask 255.255.255.0
16   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 Netzmasse 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 wurde 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

```

Der Pfad zur dem Startskript wurde dann in der Autostart-Konfigurationsdatei `/etc/rc.local` eingetragen.

zu  
Ende  
schrei-  
ben

# 7 Systemüberprüfungen

Nachdem bei der Systemkonfiguration bisher auf die Angaben aus Handbüchern und Anleitungen vertraut wurde, sollte zusätzlich die Genauigkeit von einigen Systemkomponenten überprüft werden. Hierfür wurde einmal die Messgenauigkeit des Scanners ausgewählt sowie die Genauigkeit der Zeitangaben der GNSS-Systeme. Die Genauigkeit des GNSS und der in Zusammenhang mit der IMU zu berechnenden Trajektorie hatte bereits Wilken (2017) in seiner Bachelorthesis untersucht.

## 7.1 Untersuchung der Gleichzeitigkeit von PPS-Signalen von verschiedenen GNSS-Empfängern

Für die Synchronisierung der Daten des Laserscanners und der inertialen Messeinheit wurden zwei verschiedene GNSS-Empfänger verwendet. Der für den Einbau in Consumer-Geräte gedachte uBlox-Chip, der am Raspberry Pi und am Laserscanner verwendet wird, ist leichter, unabhängiger und einfacher zu realisieren als die Übernahme der Daten mittels Adapterkabeln von der inertialen Messeinheit. Voraussetzung hierfür ist jedoch, dass beide Signale wirklich gleichzeitig erzeugt werden. Um dies zu überprüfen, soll das PPS-Signal (Impulssignal im Sekundentakt) von beiden Messsystemen mit einem Arduino überprüft werden. Am Arduino werden hierzu an zwei digitalen Eingängen die PPS-Anschlüsse der GNSS-Empfänger angeschlossen. Ein Skript misst dann die Zeit zwischen den beiden Signalen. Diese Messungen werden dann mehrfach hintereinander sowie nach einem Neustart der Systeme durchgeführt. Hiermit soll überprüft werden, ob bei einem Neustart der Messung sich die Zeitdifferenz ändert beziehungsweise überhaupt eine Zeitdifferenz dann entsteht.

## 7.2 Messgenauigkeit des Laserscanners im Vergleich

Um die Strecken und Winkelmessung des Laserscanners zu überprüfen, wurden an zwei Standpunkt an der Hafencity Universität jeweils Scans mit dem Velodyne VLP-16 durchgeführt. Das erste Messzenario fand etwa 20 Meter von einer weißen, rauen und ebenen Fassade (Verputztes Wärmedämmverbundsystem) statt. Hier wurden



Abbildung 7.1: Vergleichsmessungen mit dem Velodyne VLP-16, Zoller+Fröhlich Imager 5010 und dem Trimble S7

zum Vergleich mit einem terrestrischem Laserscanner Zöller+Fröhlich Imager 5010 der Messaufbau komplett von zwei Standpunkten aus gescannt. Hierbei sollte vor allem die Streckenmessung überprüft werden. Das zweite Messszenario fand in der Tiefgarage der Universität statt. Sie ist etwa 100 Meter lang und von vielen eckigen Säulen durchzogen. Hier bot es sich an, die Wiederholungsgenauigkeit in Strecke und Richtung des Laserscanners zu überprüfen und zusätzlich boten die Säulen verschiedene Entfernung und Auftreffwinkel. Die meisten Oberflächen bestanden hier aus weiß gestrichenen Beton.

**Auswertung der Daten von Velodyne VLP-16** Die Daten des Velodyne VLP-16 wurden mit dem in dieser Arbeit entwickelten Skript aufgezeichnet und umgewandelt. Da hierbei auffiel, dass einige Programme besser mit XYZ-Dateien umgehen können, wurde das Klassenmodell um dieses Dateiformat erweitert.

**Fehler im Skript** Bei der Auswertung der Messung der Tiefgarage (Abbildung 7.1, rechts) fiel schon vor dem Vergleich mit der Tachymettermessung auf, dass die Messwerte sich von Drehung zu Drehung des VLP-16 in Rotationsrichtung verschoben waren. Die Analyse ergab einen Umformungsfehler in der Klasse `VdDataset`. Hierdurch wurden die Horizontalrichtungen um bis zu 2 Grad (bei 1200 Umdrehungen pro Minute) fehlerhaft berechnet. Bei der Messung der Tiefgarage wurden glücklicherweise auch die Rohdaten von 2 von 5 Messreihen gespeichert, so dass hier keine erneute Messung vor Ort nötig war. Die Daten der Fassadenmessung (Abbildung 7.1, links) mussten nochmals wiederholt werden, da die Daten auch durch speziell entwickelte Skripts nicht auswertbar zu korrigieren waren.

### 7.2.1 Vergleichsmessung mittels Fassadenfront

**Nachbearbeitung der Messdaten** Die Punktwolken der zwei Standpunkte wurden in der Software Z+F LaserControl des Herstellers des Laserscanners über an der Gebäude

angebrachten Targets zusammengeführt. Anschließend wurden die Punktwolken gefiltert und in die Software Geomagic Wrap importiert. Hier wurde der Standpunkt des Velodyne VLP-16 bestimmt, in dem ein Zylinder gemäß der Größe aus dem Datenblatt (Velodyne Lidar, 2017b) in die dem VLP-16 zuzuordnenden Punkte der Punktwolke automatisch eingepasst wurde. Hieraus konnte dann durch Bestimmung der Mittellachse und der Daten des Fokuspunktes der Nullpunkt der Punktwolken des VLP-16 bestimmt werden. Die Einpassung der Rotation um die Z-Achse wurde durch zwei Fassadenecken durchgeführt, der Abgleich der Kippachsen wurde über jeweils in die Punktwolken eingepasste Bodenebenen durchgeführt.

## **8 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.

Copperwaite, Matt (2015): Learning Flask Framework. Packt Publishing, Birmingham, ISBN 978-1-783-98336-0.

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.

Kleuker, Stephan (2013): Grundkurs Software-Engineering mit UML - Der pragmatische Weg zu erfolgreichen Softwareprojekten. Springer-Verlag, Berlin Heidelberg New York, ISBN 978-3-658-00642-6.

Lott, Steven F. (2014): Mastering Object-oriented Python. Packt Publishing Ltd, Birmingham, ISBN 978-1-783-28098-8.

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), (unveröffentlicht).

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.

Wilken, Mathias (2017): Untersuchung der RTK-Performance des INS/GNSS iNAT M200 Systems. (unveröffentlicht).

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	Reflektiertes Signal nach Oberfläche, nach Beraldin et al. (2010, S. 28)	6
2.2	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	Entwurf der Schaltung zum Anschluss des GNSS-Modules an den Laserscanner . . . . .	18
3.9	Entwurf der Schaltung für die Steuerung des Raspberry Pi . . . . .	19
3.10	Layout der Lochstreifenplatine . . . . .	21
3.11	Endgültiger Schaltplan . . . . .	22
4.1	Strahlengang im Laserscanner VLP-16, Werte in Millimetern, nach Velodyne Lidar (2014) . . . . .	26
4.2	Vereinfachter Ablaufplan des Skriptes . . . . .	28
5.1	UML-Klassendiagramm . . . . .	30
7.1	Vergleichsmessungen mit dem Velodyne VLP-16, Zoller+Fröhlich Imager 5010 und dem Trimble S7 . . . . .	41

# **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) . . .	24
6.1	IP-Adressen-Verteilung . . . . .	37

# **Anhang**

# A Python-Skripte

## A.1 vdAutoStart.py

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

4  """
5  @author: Florian Timm
6  @version: 2017.11.20
7  """

9  import configparser
10 import multiprocessing
11 import os
12 import signal
13 import sys
14 import time
15 from datetime import datetime
16 from multiprocessing import Queue, Manager
17 from threading import Thread

19 from flask import Flask
20 from velodyneInterface.vdBuffer import VdBuffer
21 from velodyneInterface.vdGNSSTime import VdGNSSTime
22 from velodyneInterface.vdTransformer import VdTransformer

25 class VdAutoStart(object):

27     """ main script for automatic start """

29     def __init__(self, web_interface):
30         """
31             Constructor
32             :param web_interface: Thread with Flask web interface
33             :type web_interface: Thread
34         """
35         self.__vd_hardware = None
36         print("Data Interface for VLP-16\n")

38     # load config file
39     self.__conf = configparser.ConfigParser()
40     self.__conf.read("velodyneInterface/config.ini")
41     print(self.__conf.sections())

43     # variables for child processes
44     self.__pBuffer = None
45     self.__pTransformer = None
```

```

47     # pipes for child processes
48     manager = Manager()
49     self.__gnss_status = "(unknown)"
50     # self.__gnssReady = manager.Value('gnssReady',False)
51     self.__go_on_buffer = manager.Value('_go_on_buffer', False)
52     self.__go_on_transform = manager.Value('_go_on_transform', False)
53     self.__scanner_status = manager.Value('__scanner_status', "(unknown)")
54     self.__dataset_cnt = manager.Value('__dataset_cnt', 0)
55     self.__date = manager.Value('_date', None)

56
57     # queue for transformer
58     self.__queue = Queue()

59
60     # attribute for web interface
61     self.__web_interface = web_interface

62
63     # check admin
64     try:
65         os.rename('/etc/foo', '/etc/bar')
66         self.__admin = True
67     except IOError:
68         self.__admin = False

69
70     # check raspberry pi
71     try:
72         import RPi.GPIO
73         self.__raspberry = True
74     except ModuleNotFoundError:
75         self.__raspberry = False

76
77     self.__gnss = None

78
79     def run(self):
80         """ start script """
81
82         # handle SIGINT
83         signal.signal(signal.SIGINT, self.__signal_handler)
84
85         # use hardware control on raspberry pi
86         if self.__raspberry:
87             print("Raspberry Pi was detected")
88             from velodyneInterface.vdHardware import VdHardware
89             self.__vd.hardware = VdHardware(self)
90             self.__vd.hardware.start()
91         else:
92             print("Raspberry Pi could not be detected")
93             print("Hardware control deactivated")

94
95         # set time by using gnss
96         if self.__conf.get("functions", "use_gnss_time") == "True":
97             self.__gnss = VdGNSSTime(self)
98             self.__gnss.start()

99
100    def start_transformer(self):
101        """ Starts transformer processes"""
102        print("Start transformer...")
103        # number of transformer according number of processor cores
104        if self.__conf.get("functions", "activateTransformer") == "True":
```

```

105         self.__go_on_transform.value = True
106         n = multiprocessing.cpu_count() - 1
107         if n < 2:
108             n = 1
109         self.__pTransformer = []
110         for i in range(n):
111             t = VdTransformer(i, self)
112             t.start()
113             self.__pTransformer.append(t)

115         print(str(n) + " transformer started!")

117     def start_recording(self):
118         """ Starts recording process """
119         if not (self.__go_on_buffer.value and self.__pBuffer.is_alive()):
120             self.__go_on_buffer.value = True
121             print("Recording is starting...")
122             self.__scanner_status.value = "recording started"
123             # buffering process
124             self.__pBuffer = VdBuffer(self)
125             self.__pBuffer.start()
126             if self.__pTransformer is None:
127                 self.start_transformer()

129     def stop_recording(self):
130         """ stops buffering data """
131         print("Recording is stopping... (10 seconds timeout before kill)")
132         self.__go_on_buffer.value = False
133         self.__date.value = None
134         if self.__pBuffer is not None:
135             self.__pBuffer.join(10)
136             if self.__pBuffer.is_alive():
137                 print("Could not stop process, it will be killed!")
138                 self.__pBuffer.terminate()
139                 print("Recording terminated!")
140             else:
141                 print("Recording was not started!")

143     def stop_transformer(self):
144         """ Stops transformer processes """
145         print("Transformer is stopping... (10 seconds timeout before kill)")
146         self.__go_on_transform.value = False
147         if self.__pTransformer is not None:
148             for pT in self.__pTransformer:
149                 pT.join(15)
150                 if pT.is_alive():
151                     print(
152                         "Could not stop process, it will be killed!")
153                     pT.terminate()
154                     print("Transformer terminated!")
155             else:
156                 print("Transformer was not started!")

158     def stop_web_interface(self):
159         """ Stop web interface -- not implemented now """
160         # Todo
161         # self.__web_interface.exit()
162         # print("Web interface stopped!")
163         pass

```

```

165     def stop_hardware_control(self):
166         """ Stop hardware control """
167         if self.__vd.hardware is not None:
168             self.__vd.hardware.stop()
169             self.__vd.hardware.join(5)
170
171     def stop_children(self):
172         """ Stop child processes and threads """
173         print("Script is stopping...")
174         self.stop_recording()
175         self.stop_transformer()
176         self.stop_web_interface()
177         self.stop_hardware_control()
178         print("Child processes stopped")
179
180     def end(self):
181         """ Stop script complete """
182         self.stop_children()
183         sys.exit()
184
185     def __signal_handler(self, sig_no, frame):
186         """
187             handles SIGINT-signal
188             :param sig_no: signal number
189             :type sig_no: int
190             :param frame: execution frame
191             :type frame: frame
192         """
193         del sig_no, frame
194         print('Ctrl+C pressed!')
195         self.stop_children()
196         sys.exit()
197
198     def shutdown(self):
199         """ Shutdown Raspberry Pi """
200         self.stop_children()
201         os.system("sleep 5s; sudo shutdown -h now")
202         print("Shutdown Raspberry...")
203         sys.exit(0)
204
205     def check_queue(self):
206         """ Check, whether queue is filled """
207         if self.__queue.qsize() > 0:
208             return True
209         return False
210
211     def check_recording(self):
212         """ Check data recording by pBuffer """
213         if self.__pBuffer is not None and self.__pBuffer.is_alive():
214             return True
215         return False
216
217     def check_receiving(self):
218         """ Check data receiving """
219         x = self.__dataset_cnt.value
220         time.sleep(0.2)
221         y = self.__dataset_cnt.value
222         if x - y > 0:

```

```

223         return True
224     return False

226     # getter/setter methods
227     def __get_conf(self):
228         """
229             Gets config file
230             :return: config file
231             :rtype: configparser.ConfigParser
232         """
233         return self.__conf
234     conf = property(__get_conf)

236     def __get_gnss_status(self):
237         """
238             Gets GNSS status
239             :return: GNSS status
240             :rtype: Manager
241         """
242         return self.__gnss_status

244     def __set_gnss_status(self, gnss_status):
245         """
246             Sets GNSS status
247             :param gnss_status: gnss status
248             :type gnss_status: str
249         """
250         self.__gnss_status = gnss_status
251     gnss_status = property(__get_gnss_status, __set_gnss_status)

253     def __get_go_on_buffer(self):
254         """
255             Should Buffer buffer data?
256             :return: go on buffering
257             :rtype: Manager
258         """
259         return self.__go_on_buffer
260     go_on_buffer = property(__get_go_on_buffer)

262     def __get_go_on_transform(self):
263         """
264             Should Transformer transform data?
265             :return: go on transforming
266             :rtype: Manager
267         """
268         return self.__go_on_transform
269     go_on_transform = property(__get_go_on_transform)

271     def __get_scanner_status(self):
272         """
273             Gets scanner status
274             :return:
275             :rtype: Manager
276         """
277         return self.__scanner_status
278     scanner_status = property(__get_scanner_status)

280     def __get_dataset_cnt(self):
281         """

```

```

282         Gets number of buffered datasets
283         :return: number of buffered datasets
284         :rtype: Manager
285         """
286         return self._dataset_cnt
287     dataset_cnt = property(_get_dataset_cnt)

288     def __get_date(self):
289         """
290             Gets recording start time
291             :return: timestamp starting recording
292             :rtype: Manager
293             """
294             return self._date

295     def __set_date(self, date):
296         """
297             Sets recording start time
298             :param date: timestamp starting recording
299             :type date: datetime
300             """
301             self._date = date
302             #: recording start time
303             date = property(_get_date, __set_date)

304     def __is_admin(self):
305         """
306             Admin?
307             :return: Admin?
308             :rtype: bool
309             """
310             return self._admin
311     admin = property(__is_admin)

312     def __get_queue(self):
313         """
314             Gets transformer queue
315             :return: transformer queue
316             :rtype: Queue
317             """
318             return self._queue
319     queue = property(__get_queue)

320     # web control
321     app = Flask(__name__)

322     @app.route("/")
323     def web_index():
324         """ index page of web control """
325         runtime = "(inactive)"
326         pps = "(inactive)"
327         if ms.date.value is not None:
328             time_diff = datetime.now() - ms.date.value
329             td_sec = time_diff.seconds + \
330                 (int(time_diff.microseconds / 1000) / 1000.)
331             seconds = td_sec % 60
332             minutes = int((td_sec // 60) % 60)

```

```

341     hours = int(td_sec // 3600)
343
344     runtime = '{:02d}:{:02d}:{:06.3f}'.format(hours, minutes, seconds)
345
346     pps = '{:.0f}'.format(ms.dataset_cnt.value / td_sec)
347
348     elif ms.go_on_buffer.value:
349         runtime = "(no data)"
350
351     output = """<html>
352         <head>
353             <title>VLP16-Data-Interface</title>
354             <meta name="viewport" content="width=device-width; initial-scale=1.0;" />
355             <link href="/style.css" rel="stylesheet">
356             <meta http-equiv="refresh" content="5; URL=/>
357         </head>
358         <body>
359             <content>
360                 <h2>VLP16-Data-Interface</h3>
361                 <table style="">
362                     <tr><td id="column1">GNSS-status:</td><td>"""+ ms.gnss_status + """</td></tr>
363                     <tr><td>Scanner:</td><td>"""+ ms.scanner_status.value + """</td></tr>
364                     <tr><td>Datasets</td>
365                         <td>"""+ str(ms.dataset_cnt.value) + """</td></tr>
366                     <tr><td>Queue:</td>
367                         <td>"""+ str(ms.queue.qsize()) + """</td></tr>
368                     <tr><td>Recording time:</td>
369                         <td>"""+ runtime + """</td>
370                     </tr>
371                     <tr><td>Points/seconds:</td>
372                         <td>"""+ pps + """</td>
373                     </tr>
374                 </table><br />
375                 """
376
377     if ms.check_recording():
378         output += """<a href="/stop" id="stop">
379             Stop recording</a><br />"""
380
381     else:
382         output += """<a href="/start" id="start">
383             Start recording</a><br />"""
384
385     output += """
386         <a href="/exit" id="exit">Terminate script<br />
387         (control by SSH available only)</a></td></tr><br />
388         <a href="/shutdown" id="shutdown">Shutdown Raspberry Pi</a>
389     </content>
390     </body>
391     </html>"""
392
393     return output
394
395     @app.route("/style.css")
396     def css_style():
397         """
398             css file of web control """
399
400             return """
401                 body, html, content {
402                     text-align: center;
403                 }

```

```

400     content {
401         max-width: 15cm;
402         display: block;
403         margin: auto;
404     }
405
406     table {
407         border-collapse: collapse;
408         width: 90%;
409         margin: auto;
410     }
411
412     td {
413         border: 1px solid black;
414         padding: 1px 2px;
415     }
416
417     td#column1 {
418         width: 30%;
419     }
420
421     a {
422         display: block;
423         width: 90%;
424         padding: 0.5em 0;
425         text-align: center;
426         margin: auto;
427         color: #fff;
428     }
429
430     a#stop {
431         background-color: #e90;
432     }
433
434     a#shutdown {
435         background-color: #b00;
436     }
437
438     a#start {
439         background-color: #1a1;
440     }
441
442     a#exit {
443         background-color: #f44;
444     }
445     """
446
447
448     @app.route("/shutdown")
449     def web_shutdown():
450         """ web control: shutdown """
451         ms.shutdown()
452         return """
453             <meta http-equiv="refresh" content="3; URL=/">
454             Shutdown...
455         """
456
457     @app.route("/exit")

```

```

458     def web_exit():
459         """ web control: exit """
460         ms.end()
461         return """
462         <meta http-equiv="refresh" content="3; URL=/">
463         Terminating..."""
464
465
466     @app.route("/stop")
467     def web_stop():
468         """ web control: stop buffering """
469         ms.stop_recording()
470         return """
471         <meta http-equiv="refresh" content="3; URL=/">
472         Recording is stopping..."""
473
474
475     @app.route("/start")
476     def web_start():
477         """ web control: start buffering """
478         ms.start_recording()
479         return """
480         <meta http-equiv="refresh" content="3; URL=/">
481         Recording is starting..."""
482
483
484     def start_web():
485         """ start web control """
486         print("Web server is starting...")
487         app.run('0.0.0.0', 8080)
488
489
490     if __name__ == '__main__':
491         w = Thread(target=start_web)
492         ms = VdAutoStart(w)
493         w.start()
494         ms.run()

```

## A.2 vdBuffer.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.19
7  """
8
9  import os
10 import signal
11 import socket
12 from datetime import datetime
13 from multiprocessing import Process
14
15 from velodyneInterface.vdInterface import VdInterface
16
17
18 class VdBuffer(Process):

```

```

20     """ process for buffering binary data """
21
22     def __init__(self, master):
23         """
24             Constructor
25             :param master: instance of VdAutoStart
26             :type master: VdAutoStart
27         """
28
29         # constructor of super class
30         Process.__init__(self)
31
32         # safe pipes
33         # self.__master = master
34         self.__go_on_buffering = master.go_on_buffer
35         self.__scanner_status = master.scanner_status
36         self.__datasets = master.dataset_cnt
37         self.__queue = master.queue
38         self.__admin = master.admin
39         self.__date = master.date
40         self.__conf = master.conf
41
42         self.__file_no = 0
43
44     @staticmethod
45     def __signal_handler(sig_no, frame):
46         """
47             handles SIGINT-signal
48             :param sig_no: signal number
49             :type sig_no: int
50             :param frame: execution frame
51             :type frame: frame
52         """
53
54         del sig_no, frame
55         # self.master.end()
56         print("SIGINT vdBuffer")
57
58     def __new_folder(self):
59         """
60             creates data folder """
61         # checks time for file name and runtime
62         self.__date.value = datetime.now()
63         self.__folder = self.__conf.get("file", "namePre")
64         self.__folder += self.__date.value.strftime(
65             self.__conf.get("file", "timeFormat"))
66         # make folder
67         os.makedirs(self.__folder)
68         print("Data folder: " + self.__folder)
69
70     def run(self):
71         """
72             starts buffering process """
73         signal.signal(signal.SIGINT, self.__signal_handler)
74
75         # open socket to scanner
76         sock = VdInterface.get_data_stream(self.__conf)
77         self.__scanner_status.value = "Socket connected"
78
79         buffer = b' '
80         datasets_in_buffer = 0

```

```

78         self.__datasets.value = 0
79
80         # process priority
81         if self.__admin:
82             os.nice(-18)
83
84         transformer = self.__conf.get(
85             "functions",
86             "activateTransformer") == "True"
87         measurements_per_dataset = int(self.__conf.get(
88             "device", "valuesPerDataset"))
89
90         sock.settimeout(1)
91         while self.__go_on_buffering.value:
92             try:
93                 # get data from scanner
94                 data = sock.recvfrom(1248)[0]
95
96                 if datasets_in_buffer == 0 and self.__file_no == 0:
97                     self.__new_folder()
98
99                     # RAM-buffer
100                    buffer += data
101                    datasets_in_buffer += 1
102
103                    self.__datasets.value += measurements_per_dataset
104
105                    # safe data to file every 1500 datasets
106                    # (about 5 or 10 seconds)
107                    if (datasets_in_buffer >= 1500) or \
108                        (not self.__go_on_buffering.value):
109
110                        # write file
111                        f = open(
112                            self.__folder + "/" + str(self.__file_no) + ".bin",
113                            "wb")
114
115                        f.write(buffer)
116
117                        f.close()
118
119                        if transformer:
120                            self.__queue.put(f.name)
121
122                        # clear buffer
123                        buffer = b''
124                        datasets_in_buffer = 0
125
126                        # count files
127                        self.__file_no += 1
128
129
130                        if data == 'QUIT':
131                            break
132
133                    except socket.timeout:
134                        print("No data")
135                        continue
136
137                    sock.close()
138
139                    self.__scanner_status.value = "recording stopped"
140
141                    print("Disconnected!")

```

## A.3 vdTransformer.py

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

```

2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.20
7  """
8  import os
9  import signal
10 from multiprocessing import Process
11 from queue import Empty
12
13 from velodyneInterface.vdDataset import VdDataset
14
15 from velodyneInterface.vdFile import VdTxtFile
16
17
18 class VdTransformer(Process):
19
20     """ Process for transforming data from Velodyne VLP-16 """
21
22     def __init__(self, number, master):
23         """
24             Constructor
25             :param number: number of process
26             :type number: int
27             :param master: instance of VdAutoStart
28             :type master: VdAutoStart
29         """
30
31     # constructor of super class
32     Process.__init__(self)
33
34     self.__queue = master.queue
35     self.__number = number
36     self.__admin = master.admin
37     self.__go_on_transform = master.go_on_transform
38     self.__conf = master.conf
39
40     @staticmethod
41     def __signal_handler(sig_no, frame):
42         """
43             handles SIGINT-signal
44             :param sig_no: signal number
45             :type sig_no: int
46             :param frame: execution frame
47             :type frame: frame
48         """
49         del sig_no, frame
50         # self.master.end()
51         print("SIGINT vdTransformer")
52
53     def run(self):
54         """
55             starts transforming process
56         """
57         signal.signal(signal.SIGINT, self.__signal_handler)
58
59         if self.__admin:
60             os.nice(-15)
61
62         old_folder = ""

```

```

62     try:
63         while self.__go_on_transform.value:
64             try:
65                 # get file name from queue
66                 filename = self.__queue.get(True, 2)
67                 folder = os.path.dirname(filename)
68                 if dir != old_folder:
69                     vd_file = VdTxtFile(
70                         self.__conf,
71                         folder + "/obj_file" + str(self.__number))
72                     old_folder = folder
73
74             f = open(filename, "rb")
75
76             # count number of datasets
77             file_size = os.path.getsize(f.name)
78             dataset_cnt = int(file_size / 1206)
79
80             for i in range(dataset_cnt):
81                 # read next
82                 vd_data = VdDataset(self.__conf, f.read(1206))
83
84                 # convert data
85                 vd_data.convert_data()
86
87                 # add them on writing queue
88                 vd_file.add_dataset(vd_data)
89
90                 # write file
91                 vd_file.write()
92                 # close file
93                 f.close()
94                 # delete binary file
95                 if self.__conf.get("file", "deleteBin") == "True":
96                     os.remove(f.name)
97             except Empty:
98                 print("Queue empty!")
99                 continue
100            except BrokenPipeError:
101                print("vdTransformer-Pipe broken")

```

## A.4 vdInterface.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.19
7  """
8
9  import socket
10 import sys
11
12
13 class VdInterface(object):

```

```

15     """ interface to velodyne scanner """
16
17     @staticmethod
18     def get_data_stream(conf):
19         """
20             Creates socket to scanner data stream
21             :param conf: configuration file
22             :type conf: configparser.ConfigParser
23             :return: socket to scanner
24             :rtype: socket.socket
25         """
26
27         return VdInterface.get_stream(conf.get("network", "UDP_IP"),
28                                       int(conf.get("network", "UDP_PORT_DATA")))
28
29     @staticmethod
30     def get_gnss_stream(conf):
31         """
32             Creates socket to scanner gnss stream
33             :param conf: configuration file
34             :type conf: configparser.ConfigParser
35             :return: socket to scanner
36             :rtype: socket.socket
37         """
38
39         return VdInterface.get_stream(conf.get("network", "UDP_IP"),
40                                       int(conf.get("network", "UDP_PORT_GNSS")))
40
41     @staticmethod
42     def get_stream(ip, port):
43         """
44             Creates socket to scanner stream
45             :param ip: ip address of scanner
46             :type ip: str
47             :param port: port of scanner
48             :type port: int
49             :return: socket to scanner
50             :rtype: socket.socket
51         """
52
53         # Create Datagram Socket (UDP)
54         try:
55             # IPv4 UDP
56             sock = socket.socket(type=socket.SOCK_DGRAM)
57             print('Socket created!')
58         except socket.error:
59             print('Could not create socket!')
60             sys.exit()
61
62         # Sockets Options
63         sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
64         sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
65         # Allows broadcast UDP packets to be sent and received.
66
67         # Bind socket to local host and port
68         try:
69             sock.bind((ip, port))
70         except socket.error:
71             print('Bind failed.')
72
73         print('Socket connected!')

```

```

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

78     return sock

```

## A.5 vdGNSSTime.py

```

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

4  """
5  @author: Florian Timm
6  @version: 2017.11.20
7  """

9  import os
10 import socket
11 from datetime import datetime
12 from threading import Thread

14 import serial

16 from velodyneInterface.vdInterface import VdInterface

19 class VdGNSSTime(Thread):
21     """ system time by gnss data """

23     def __init__(self, master):
24         """
25             Constructor
26             :param master: instance of VdAutoStart
27             :type master: VdAutoStart
28         """
29         Thread.__init__(self)
30         self.__tScanner = None
31         self.__tSerial = None
32         self.__master = master
33         self.__conf = master.conf
34         self.__time_corrected = False

36     def run(self):
37         """
38             starts threads for time detection
39         """
40         # get data from serial port
41         self.__tSerial = Thread(target=self.__get_gnss_time_from_serial())
42         self.__tSerial.start()

44         # get data from scanner
45         self.__tScanner = Thread(target=self.__get_gnss_time_from_scanner())
46         self.__tScanner.start()

48         self.__master.gnss_status = "Connecting..."
50     def __get_gnss_time_from_scanner(self):

```

```

51     """ gets data by scanner network stream """
52     sock = VdInterface.get_gnss_stream(self.__conf)
53     sock.settimeout(1)
54     self.__master.gnss_status = "Wait for fix..."
55     while not self.__time_corrected:
56         try:
57             data = sock.recvfrom(2048)[0] # buffer size is 2048 bytes
58             message = data[206:278].decode('utf-8', 'replace')
59             if self.__get_gnss_time_from_string(message):
60                 break
61         except socket.timeout:
62             continue
63         # else:
64         #     print(message)
65         if data == 'QUIT':
66             break
67     sock.close()

69     # noinspection PyArgumentList
70     def __get_gnss_time_from_serial(self):
71         """ get data by serial port """
72         ser = None
73         try:
74             port = self.__conf.get("serial", "GNSSport")
75             ser = serial.Serial(port, timeout=1)
76             self.__master.gnss_status = "Wait for fix..."
77             while not self.__time_corrected:
78                 line = ser.readline()
79                 message = line.decode('utf-8', 'replace')
80                 if self.__get_gnss_time_from_string(message):
81                     break
82                 # else:
83                 #     print(message)
84         except serial.SerialTimeoutException:
85             pass
86         except serial.serialutil.SerialException:
87             print("Could not open serial port!")
88         finally:
89             if ser is not None:
90                 ser.close()

92     def __get_gnss_time_from_string(self, message):
93         if message[0:6] == "$GPRMC":
94             p = message.split(",")
95             if p[2] == "A":
96                 print("GNSS-Fix")
97                 timestamp = datetime.strptime(p[1] + "D" + p[9],
98                                              '%H%M%S.000%d%m%y')
99                 self.__set_system_time(timestamp)
100                self.__time_corrected = True
101                self.__master.gnss_status = "Got time!"
102                return True
103            return False

105    def __set_system_time(self, timestamp):
106        """
107            sets system time
108            :param timestamp: current timestamp
109            :type timestamp: datetime

```

```

110         :return:
111         :rtype:
112         """
113         os.system("timedatectl set-ntp 0")
114         os.system("timedatectl set-time \"\" +
115                     timestamp.strftime("%Y-%m-%d %H:%M:%S") + "\"")
116         os.system("timedatectl set-ntp 1")
117         self.__master.set_gnss_status("System time set")
118
119     def stop(self):
120         """ stops all threads """
121         self.__master.gnss_status = "Stopped"
122         self.__time_corrected = True

```

## A.6 vdHardware.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.20
7  """
8
9  import RPi.GPIO as GPIO
10 import time
11 from threading import Thread
12 import threading
13
14
15 class VdHardware(Thread):
16
17     """ controls hardware control, extends Thread """
18
19     def __init__(self, master):
20         """
21             Constructor
22             :param master: instance of VdAutoStart
23             :type master: VdAutoStart
24         """
25         Thread.__init__(self)
26
27         GPIO.setmode(GPIO.BCM)
28
29         self.__taster1 = 18 # start / stop
30         self.__taster2 = 25 # shutdown
31
32         # led-pins:
33         # 0: receiving
34         # 1: queue
35         # 2: recording
36         self.__led = [10, 9, 11]
37         self.__receiving = False
38         self.__queue = False
39         self.__recording = False
40
41         self.__master = master

```

```

43         # activate input pins
44         # recording start/stop
45         GPIO.setup(self.__taster1, GPIO.IN, pull_up_down=GPIO.PUD_UP)
46         # shutdown
47         GPIO.setup(self.__taster2, GPIO.IN, pull_up_down=GPIO.PUD_UP)

48         # activate outputs
49         for l in self.__led:
50             GPIO.setup(l, GPIO.OUT)    # GPS-Fix
51             GPIO.output(l, GPIO.LOW)

52         self.__go_on = True

53
54     def run(self):
55         """ run thread and start hardware control """
56         GPIO.add_event_detect(
57             self.__taster1,
58             GPIO.FALLING,
59             self.__button1_pressed)
60         GPIO.add_event_detect(
61             self.__taster2,
62             GPIO.FALLING,
63             self.__button1_pressed)

64         self.__timer_check_leds()

65
66     def __timer_check_leds(self):
67         """ checks LEDs every second """
68         self.__check_leds()
69         if self.__go_on:
70             t = threading.Timer(1, self.__timer_check_leds)
71             t.start()

72
73     def __check_leds(self):
74         """ check LEDs """
75         self.__set_recording(self.__master.check_recording())
76         self.__set_receiving(self.__master.check_receiving())
77         self.__set_queue(self.__master.check_queue())

78
79     def __button1_pressed(self):
80         """ raised when button 1 is pressed """
81         time.sleep(0.1)  # contact bounce

82         # > 2 seconds
83         wait = GPIO.wait_for_edge(self.__taster1, GPIO.RISING, timeout=1900)

84
85         if wait is None:
86             # no rising edge = pressed
87             if self.__master.go_on_buffer.value:
88                 self.__master.stop_recording()
89             else:
90                 self.__master.start_recording()

91
92     def __button2_pressed(self):
93         """ raised when button 1 is pressed """
94         time.sleep(0.1)  # contact bounce

95         # > 2 seconds
96         wait = GPIO.wait_for_edge(self.__taster2, GPIO.RISING, timeout=1900)

```

```

103     if wait is None:
104         # no rising edge = pressed
105         self._master.shutdown()
106
107     def __switch_led(self, led, yesno):
108         """
109             switch led
110             :param led: pin of led
111             :type led: int
112             :param yesno: True = on
113             :type yesno: bool
114         """
115         if yesno:
116             GPIO.output(self._led[led], GPIO.HIGH)
117         else:
118             GPIO.output(self._led[led], GPIO.LOW)
119
120     def __update_leds(self):
121         """ switch all LEDs to right status """
122         self.__switch_led(0, self.__receiving)
123         self.__switch_led(1, self.__queue)
124         self.__switch_led(2, self.__recording)
125
126     def __set_receiving(self, yesno):
127         """
128             set receiving variable and led
129             :param yesno: True = on
130             :type yesno: bool
131         """
132         if self.__receiving != yesno:
133             self.__receiving = yesno
134             self.__update_leds()
135
136     def __set_queue(self, yesno):
137         """
138             set queue variable and led
139             :param yesno: True = on
140             :type yesno: bool
141         """
142         if self.__queue != yesno:
143             self.__queue = yesno
144             self.__update_leds()
145
146     def __set_recording(self, yesno):
147         """
148             set recording variable and led
149             :param yesno: True = on
150             :type yesno: bool
151         """
152         if self.__recording != yesno:
153             self.__recording = yesno
154             self.__update_leds()
155
156     def stop(self):
157         """ stops thread """
158         self.__go_on = False
159         GPIO.cleanup()

```

## A.7 vdFile.py

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

4  """
5  @author: Florian Timm
6  @version: 2017.11.22
7  """

9  import datetime
10 import sqlite3
11 from abc import abstractmethod, ABC

13 from velodyneInterface.vdPoint import VdPoint

16 class VdFile(ABC):

18     """ abstract class for saving data """

20     def __init__(self, conf, filename=""):
21         """
22             Creates a new ascii-file
23             :param conf: configuration file
24             :type conf: configparser.ConfigParser
25             :param filename: name and path to new file
26             :type filename: str
27         """
28         self.__conf = conf
29         # create file
30         self.__writing_queue = []
31         self._open(filename)

33     def __get_writing_queue(self):
34         """
35             Returns points in queue
36             :return: points in queue
37             :rtype: VdPoint []
38         """
39         return self.__writing_queue

41     writing_queue = property(__get_writing_queue)

43     def clear_writing_queue(self):
44         """
45             clears writing queue """
46         self.__writing_queue = []

47     def _make_filename(self, file_format, file_name=""):
48         """
49             generates a new file_name from timestamp
50             :param file_format: file suffix
51             :type file_format: str
52             :return: string with date and suffix
53             :rtype: str
54         """
55         if file_name == "":
56             name = self.__conf.get("file", "namePre")
57             name += datetime.datetime.now().strftime(
```

```

58         self.__conf.get("file", "timeFormat"))
59         name = "." + file_format
60         return name
61     elif not file_name.endswith("." + file_format):
62         return file_name + "." + file_format
63     return file_name
64
65     def write_data(self, data):
66         """
67             adds data and writes it to file
68             :param data: ascii data to write
69             :type data: VdPoint []
70         """
71         self.add_dataset(data)
72         self.write()
73
74     def add_point(self, p):
75         """
76             Adds a point to write queue
77             :param p: point
78             :type p: VdPoint
79         """
80         self.__writing_queue.append(p)
81
82     def add_dataset(self, dataset):
83         """
84             adds multiple points to write queue
85             :param dataset: multiple points
86             :type dataset: VdPoint []
87         """
88         self.__writing_queue.extend(dataset)
89
90     def read_from_txt_file(self, filename, write=False):
91         """
92             Parses data from txt file
93             :param filename: path and filename of txt file
94             :type filename: str
95             :param write: write data to new file while reading txt
96             :type write: bool
97         """
98         txt = open(filename)
99
100        for no, line in enumerate(txt):
101            try:
102                p = VdPoint.parse_string(self.__conf, line)
103                self.writing_queue.append(p)
104                # print("Line {0} was parsed".format(no + 1))
105            except ValueError as e:
106                print("Error in line {0}: {1}".format(no + 1, e))
107
108            if write and len(self.writing_queue) > 50000:
109                self.write()
110
111        if write:
112            self.write()
113
114    @abstractmethod
115    def write(self):
116        """ abstract method: should write data from writing_queue """
117        pass

```

```

118     @abstractmethod
119     def _open(self, filename):
120         """
121             abstract method: should open file for writing
122             :param filename: file name
123             :type filename: str
124         """
125         pass
126
127     @abstractmethod
128     def close(self):
129         """ abstract method: should close file """
130         pass
131
132
133 class VdASCIIFile(VdFile):
134     """ abstract class for writing ascii files """
135
136     def _open_ascii(self, filename, file_format):
137         """
138             opens ascii file for writing
139             :param filename: file name
140             :type filename: str
141         """
142         filename = self._make_filename(file_format, filename)
143         self.__file = open(filename, 'a')
144
145     def _write2file(self, data):
146         """
147             writes ascii data to file
148             :param data: data to write
149             :type data: str
150         """
151         self.__file.write(data)
152
153     def write(self):
154         """ writes data to file """
155         txt = ""
156         for d in self.writing_queue:
157             txt += self._format(d)
158         self._write2file(txt)
159         self.clear_writing_queue()
160
161     @abstractmethod
162     def _format(self, p):
163         """
164             abstract method: should convert point data to string for ascii file
165             :param p: Point
166             :type p: VdPoint
167             :return: point data as string
168             :rtype: str
169         """
170         pass
171
172     def close(self):
173         """ close file """
174         self.__file.close()

```

```

177  class VdObjFile(VdASCIIFile):
178
179      """ creates and fills an obj-file """
180
181      def _open(self, filename=""):
182          """
183              opens a txt file for writing
184              :param filename: name and path to new file
185              :type filename: str
186          """
187          VdASCIIFile._open_ascii(self, filename, "obj")
188
189      def _format(self, p):
190          """
191              Formats point for OBJ
192              :param p: VdPoint
193              :type p: VdPoint
194              :return: obj point string
195              :rtype: str
196          """
197          x, y, z = p.get_xyz()
198          format_string = 'v {:.3f} {:.3f} {:.3f}\n'
199          return format_string.format(x, y, z)
200
201
202  class VdXYZFile(VdASCIIFile):
203
204      """ creates and fills an xyz-file """
205
206      def _open(self, filename=""):
207          """
208              opens a txt file for writing
209              :param filename: name and path to new file
210              :type filename: str
211          """
212          VdASCIIFile._open_ascii(self, filename, "xyz")
213
214      def _format(self, p):
215          """
216              Formats point for OBJ
217              :param p: VdPoint
218              :type p: VdPoint
219              :return: obj point string
220              :rtype: str
221          """
222          x, y, z = p.get_xyz()
223          format_string = '{:.3f} {:.3f} {:.3f}\n'
224          return format_string.format(x, y, z)
225
226
227  class VdTxtFile(VdASCIIFile):
228
229      """ creates and fills an txt-file """
230
231      def _open(self, filename=""):
232          """
233              opens a txt file for writing
234              :param filename: name and path to new file

```

```

235         :type filename: str
236         """
237         VdASCIIFile._open_ascii(self, filename, "txt")
238
239     def _format(self, p):
240         """
241             Formats point for TXT
242             :param p: VdPoint
243             :type p: VdPoint
244             :return: txt point string
245             :rtype: str
246             """
247             format_string = '{:012.1f}\t{:07.3f}\t{: 03.0f}\t{:06.3f}\t{:03.0f}\n'
248             return format_string.format(p.time,
249                                         p.azimuth,
250                                         p.vertical,
251                                         p.distance,
252                                         p.reflection)
253
254
255     class VdSQLite(VdFile):
256         """ class for writing data to sqlite database """
257
258         def _open(self, filename):
259             """
260                 opens a new db file
261                 :param filename: name of db file
262                 :type filename: str
263             """
264             filename = self._make_filename("db", filename)
265             print(filename)
266             self.__db = sqlite3.connect(filename)
267             self.__cursor = self.__db.cursor()
268             self.__cursor.execute("CREATE TABLE raw_data (
269                                 id INTEGER PRIMARY KEY AUTOINCREMENT ,
270                                 time FLOAT ,
271                                 azimuth FLOAT ,
272                                 vertical FLOAT ,
273                                 distance FLOAT ,
274                                 reflection INTEGER )")
275             self.__db.commit()
276
277         def write(self):
278             """ writes data to database """
279             insert = []
280             for p in self.writing_queue:
281                 insert.append((p.time, p.azimuth, p.vertical,
282                               p.distance, p.reflection))
283
284             self.__cursor.executemany("INSERT INTO raw_data (
285                                     time, azimuth, vertical, "
286                                     "distance, reflection) "
287                                     "VALUES (?, ?, ?, ?, ?)", insert)
288             self.__db.commit()
289             self.clear_writing_queue()
290
291         def close(self):
292             """ close database """
293             self.__db.close()

```

## A.8 vdDataset.py

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

4  """
5  @author: Florian Timm
6  @version: 2017.11.20
7  """

9  import json

11 from velodyneInterface.vdPoint import VdPoint

14 class VdDataset(object):

16     """ representation of one dataset of velodyne vlp-16 """

18     def __init__(self, conf, dataset):
19         """
20             Constructor
21             :param conf: config-file
22             :type conf: configparser.ConfigParser
23             :param dataset: binary dataset
24             :type dataset: bytes
25         """

27         self.__dataset = dataset
28         self.__conf = conf

30         self.__vertical_angle = json.loads(
31             self.__conf.get("device", "verticalAngle"))
32         self.__offset = json.loads(self.__conf.get("device", "offset"))
33         self.__data = []

35     def get_azimuth(self, block):
36         """
37             gets azimuth of a data block
38             :param block: number of data block
39             :type block: int
40             :return: azimuth
41             :rtype: float
42         """

44         offset = self.__offset[block]
45         # change byte order
46         azi = ord(self.__dataset[offset + 2:offset + 3]) + \
47             (ord(self.__dataset[offset + 3:offset + 4]) << 8)
48         azi /= 100.0
49         # print(azi)
50         return azi

52     def get_time(self):
53         """
54             gets timestamp of dataset
55             :return: timestamp of dataset
56             :rtype: int
57         """
```

```

59         time = ord(self._dataset[1200:1201]) + \
60             (ord(self._dataset[1201:1202]) << 8) + \
61             (ord(self._dataset[1202:1203]) << 16) + \
62             (ord(self._dataset[1203:1204]) << 24)
63         # print(time)
64         return time
65
66     def is_dual_return(self):
67         """
68             checks whether dual return is activated
69             :return: dual return active?
70             :rtype: bool
71         """
72
73         mode = ord(self._dataset[1204:1205])
74         if mode == 57:
75             return True
76         else:
77             return False
78
79     def get_azimuths(self):
80         """
81             get all azimuths and rotation angles from dataset
82             :return: azimuths and rotation angles
83             :rtype: list, list
84         """
85
86         # create empty lists
87         azimuths = [0.] * 24
88         rotation = [0.] * 12
89
90         # read existing azimuth values
91         for j in range(0, 24, 2):
92             a = self.get_azimuth(j // 2)
93             azimuths[j] = a
94
95         #: rotation angle
96         d = 0
97
98         # DualReturn active?
99         if self.is_dual_return():
100             for j in range(0, 19, 4):
101                 d2 = azimuths[j + 4] - azimuths[j]
102                 if d2 < 0:
103                     d2 += 360.0
104                 d = d2 / 2.0
105                 a = azimuths[j] + d
106                 azimuths[j + 1] = a
107                 azimuths[j + 3] = a
108                 rotation[j // 2] = d
109                 rotation[j // 2 + 1] = d
110
111             rotation[10] = d
112             azimuths[21] = azimuths[20] + d
113
114         # Strongest / Last-Return
115     else:
116         for j in range(0, 22, 2):

```

```

117         d2 = azimuths[j + 2] - azimuths[j]
118         if d2 < 0:
119             d2 += 360.0
120         d = d2 / 2.0
121         a = azimuths[j] + d
122         azimuths[j + 1] = a
123         rotation[j // 2] = d

125     # last rotation angle from angle before
126     rotation[11] = d
127     azimuths[23] = azimuths[22] + d

129     # >360 -> -360
130     for j in range(24):
131         if azimuths[j] > 360.0:
132             azimuths[j] -= 360.0

134     # print (azimuths)
135     # print (rotation)
136     return azimuths, rotation

138 def convert_data(self):
139     """ converts binary data to objects """
140
141     azimuth, rotation = self.get_azimuths()
142     dual_return = self.is_dual_return()

144     # timestamp from dataset
145     time = self.get_time()
146     times = [0.] * 12
147     t_2repeat = 2 * float(self.__conf.get("device", "tRepeat"))
148     if dual_return:
149         for i in range(0,12,2):
150             times[i] = time
151             times[i+1] = time
152             time += t_2repeat
153     else:
154         for i in range(12):
155             time[i] = time
156             time += t_2repeat

158     t_between_laser = float(self.__conf.get("device", "tInterBeams"))
159     t_recharge = float(self.__conf.get("device", "tRecharge"))
160     part_rotation = float(self.__conf.get("device", "ratioRotation"))

162     # data package has 12 blocks with 32 measurements
163     for i in range(12):
164         offset = self.__offset[i]
165         time = times[i]
166         for j in range(2):
167             azi_block = azimuth[i*2 + j]
168             for k in range(16):
169                 # get distance
170                 dist = ord(self.__dataset[4 + offset:5 + offset]) \
171                     + (ord(self.__dataset[5 + offset:6 + offset]) << 8)
172                 if dist > 0:
173                     dist /= 500.0

175             reflection = ord(self.__dataset[6 + offset:7 + offset])

```

```

177             # interpolate azimuth
178             a = azi_block + rotation[i] * k * part_rotation
179             # print(a)

181             # create point
182             p = VdPoint(
183                 self.__conf, round(
184                     time, 1), a, self.__vertical_angle[k],
185                     dist, reflection)
186             self.__data.append(p)

188             time += t_between_laser

190             # offset for next loop
191             offset += 3
192             time += t_recharge - t_between_laser

194     def get_data(self):
195         """
196             get all point data
197             :return: list of VdPoints
198             :rtype: list
199             """
200         return self.__data

```

## A.9 vdPoint.py

```

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

4  """
5  @author: Florian Timm
6  @version: 2017.11.19
7  """

9  import math

12 class VdPoint(object):

14     """ Represents a point """

16     __dRho = math.pi / 180.0

18     def __init__(self, conf, time, azimuth, vertical, distance, reflection):
19         """
20             Constructor
21             :param conf: config file
22             :type conf: configparser.ConfigParser
23             :param time: recording time in microseconds
24             :type time: float
25             :param azimuth: Azimuth direction in degrees
26             :type azimuth: float
27             :param vertical: Vertical angle in degrees
28             :type vertical: float
29             :param distance: distance in metres
30             :type distance: float

```

```

31         :param reflection: reflection 0-255
32         :type reflection: int
33         """
34         self.__time = time
35         self.__azimuth = azimuth
36         self.__vertical = vertical
37         self.__reflection = reflection
38         self.__distance = distance
39         self.__conf = conf
40
41     @staticmethod
42     def parse_string(conf, line):
43         """
44             Parses string to VdPoint
45             :param conf: config file
46             :type conf: configparser.ConfigParser
47             :param line: Point as TXT
48             :type line: str
49             :return: Point
50             :rtype: VdPoint
51             :raise ValueError: malformed string
52         """
53         d = line.split()
54         if len(d) > 4:
55             time = float(d[0])
56             azimuth = float(d[1])
57             vertical = float(d[2])
58             distance = float(d[3])
59             reflection = int(d[4])
60             return VdPoint(conf, time, azimuth,
61                           vertical, distance, reflection)
61         else:
62             raise ValueError('Malformed string')
63
64     def __deg2rad(self, degree):
65         """
66             converts degree to radians
67             :param degree: degrees
68             :type degree: float
69             :return: radians
70             :rtype: float
71         """
72
73         return degree * self.__dRho
74
75     def get_xyz(self):
76         """
77             Gets local coordinates
78             :return: local coordinates x, y, z in metres
79             :rtype: float, float, float
80         """
81         beam_center = float(self.__conf.get("device", "beamCenter"))
82
83         # slope distance to beam center
84         d = self.distance - beam_center
85
86         # vertical angle in radians
87         v = self.vertical.radians
88
89         # azimuth in radians

```

```

90         a = self.azimuth_radians
92
93         # horizontal distance
94         s = d * math.cos(v) + beam_center
95
96         x = s * math.sin(a)
97         y = s * math.cos(a)
98         z = d * math.sin(v)
99
100        return x, y, z
101
102    def __get_time(self):
103        """
104            Gets recording time
105            :return: recording time in microseconds
106            :rtype: float
107        """
108        return self.__time
109
110    def __get_azimuth(self):
111        """
112            Gets azimuth direction
113            :return: azimuth direction in degrees
114            :rtype: float
115        """
116        return self.__azimuth
117
118    def __get_azimuth_radians(self):
119        """
120            Gets azimuth in radians
121            :return: azimuth direction in radians
122            :rtype: float
123        """
124        return self.__deg2rad(self.azimuth)
125
126    def __get_vertical(self):
127        """
128            Gets vertical angle in degrees
129            :return: vertical angle in degrees
130            :rtype: float
131        """
132        return self.__vertical
133
134    def __get_vertical_radians(self):
135        """
136            Gets vertical angle in radians
137            :return: vertical angle in radians
138            :rtype: float
139        """
140        return self.__deg2rad(self.vertical)
141
142    def __get_reflection(self):
143        """
144            Gets reflection
145            :return: reflection between 0 and 255
146            :rtype: int
147        """
148        return self.__reflection

```

```

149     def __get_distance(self):
150         """
151             Gets distance
152             :return: distance in metres
153             :rtype: float
154         """
155         return self.__distance

157     # properties
158     time = property(__get_time)
159     azimuth = property(__get_azimuth)
160     azimuth_radians = property(__get_azimuth_radians)
161     vertical = property(__get_vertical)
162     vertical_radians = property(__get_vertical_radians)
163     reflection = property(__get_reflection)
164     distance = property(__get_distance)

```

## A.10 config.ini

```

1 [network]
2 UDP_IP = 0.0.0.0
3 UDP_PORT_DATA = 2368
4 UDP_PORT_GNSS = 8308

6 [serial]
7 # Serieller Port
8 #Rasberry
9 GNSSport = /dev/ttyAMA0
10 #Ubuntu
11 #GNSSport = /dev/ttyUSBO

13 [functions]
14 # Zeitgleiche Transformation zu txt aktivieren
15 activateTransformer = True

17 # GNSS-Zeit verwenden
18 use_gnss_time = True


22 [file]
23 # Binaere Dateien nach deren Transformation loeschen
24 deleteBin = True

26 #Takt zur Speicherung Buffer -> HDD
27 takt = 5

29 # Format der Zeit am Dateinamen
30 dateFormat = %%Y-%%m-%%dT%%H:%%M:%%S

32 # Dateipraefix der zu speichernden Datei
33 namePre = data


36 [device]
37 # Zeit zwischen den Messungen der Einzelstrahlen
38 tInterBeams = 2.304

```

```

40 # Zeit zwischen zwei Aussendungen des gleichen Messlasers
41 tRepeat = 55.296

43 # Hoehenwinkel der 16 Messstrahlen
44 verticalAngle = [-15, 1, -13, -3, -11, 5, -9, 7, -7, 9, -5, 11, -3, 13, -1, 15]

46 # Anteil der Zeit zwischen Einzellasern an Wiederholungszeit,
47 # fuer Interpolation des Horizontalwinkels
48 ratioRotation = 0.04166666666666666
49 #tZwischenStrahl / tRepeat

51 # Zeit nach letztem Strahl bis zum naechsten
52 tRecharge = 20.736
53 #tRepeat - 15 * tZwischenStrahl

55 # Abstand des Strahlenzentrums von der Drehachse
56 beamCenter = 0.04191

58 valuesPerDataset = 384
59 #12*32

61 # Bytes pro Messdatenblock
62 offsetBlock = 100
63 # 3 * 32 + 4

65 # Versatz vom Start fuer jeden Messblock
66 offset = [0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100]
67 #list(range(0,1206,offsetBlock))[0:12]

```

## A.11 convTxt2Obj.py

```

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

4 """
5 @author: Florian Timm
6 @version: 2017.11.20
7 """
8 import configparser

10 from velodyneInterface.vdFile import VdObjFile

12 fileName = "Beispieldateien/test.txt"

14 conf = configparser.ConfigParser()
15 conf.read("velodyneInterface/config.ini")

17 f = VdObjFile(conf, fileName)
18 f.read_from_txt_file(fileName, True)

```

## A.12 convBin2Obj.py

```

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

4 """
5 @author: Florian Timm

```

```

6  @version: 2017.11.20
7  """
9  import configparser
10 import os
11 from glob import glob

13 from velodyneInterface.vdDataset import VdDataset

15 from velodyneInterface.vdFile import VdObjFile

17 # load config file
18 conf = configparser.ConfigParser()
19 conf.read("velodyneInterface/config.ini")

21 fs = glob(
22     "/ssd/daten/ThesisMessung/tief1_bin/*.bin")

24 if len(fs) > 0:
25     folder = os.path.dirname(fs[0])
26     obj = VdObjFile(
27         conf,
28         folder + "/file")

30 for filename in fs:
31     print(filename)

33     bin_file = open(filename, "rb")

35     # Calculate number of datasets
36     fileSize = os.path.getsize(bin_file.name)
37     cntDatasets = int(fileSize / 1206)

39     for i in range(cntDatasets):
40         vdData = VdDataset(conf, bin_file.read(1206))
41         vdData.convert_data()
42         obj.write_data(vdData.get_data())
43         bin_file.close()
44     obj.close()

```

# B Beispieldateien

## B.1 Rohdaten vom Scanner

Netzwerk-Header

Flag (FF EE) Horizontalrichtung

Strecke Reflektivität

Timestamp Return-Modus

```
0000      ff ff ff ff ff ff 60 76 88 00 00 00 00 08 00 45 00
0010      04 d2 00 00 40 00 ff 11 b4 aa c0 a8 01 c8 ff ff
0020      ff ff 09 40 09 40 04 be 00 00 ff ee 02 4d 00 00
0030      0f 00 00 0a 00 00 16 f0 01 04 00 00 0d 00 00 0a
0040      00 00 0d 00 00 06 00 00 0b 00 00 08 00 00 10 00
0050      00 05 92 01 05 00 00 04 00 00 0f 00 00 07 00 00
0060      0f 00 00 0a 00 00 16 e6 01 08 00 00 0d 00 00 0a
0070      00 00 0d 00 00 06 00 00 0b 00 00 08 00 00 10 00
0080      00 05 7e 01 05 00 00 04 00 00 0f 00 00 07 ff ee
0090      02 4d 00 00 0f 00 00 0a 00 00 16 f0 01 04 00 00
...
04d0      05 00 00 04 00 00 0f 00 00 07 8c 25 44 63 39 22
```

## 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