

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

1 Einleitung	1
1.1 Problemstellung	1
1.2 Zielsetzung	1
1.3 Struktur	2
2 Grundlagen des Airborne Laserscanings	3
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
3 Technische Realisierung	11
3.1 Verwendete Gerätschaften	11
3.1.1 Velodyne VLP-16	11
3.1.2 Inertiale Messeinheit und GNSS-Empfänger iMAR iNAT-M200-FLAT	12
3.1.3 Raspberry Pi 3 Typ B	13
3.1.4 Multikopter Copterproject CineStar 6HL	14
3.1.5 Gimbal Freefly MöVI M5	16
3.2 Auswahl des Datenverarbeitungssystems	16
3.3 Stromversorgung	17
3.4 Anbindung des Raspberry Pi an den Laserscanner	18
3.5 Verbindung des GNSS-Moduls zum Laserscanner	18
3.6 Steuerung im Betrieb	19
3.7 Platinenentwurf und -realisierung	21
4 Theoretische Datenverarbeitung	24
4.1 Verwendete Programmiersprachen	24

4.2	Datenlieferung vom Laserscanner	25
4.3	Geplantes Datenmodell	26
4.4	Weiterverarbeitung der Daten zu Koordinaten	26
4.5	Anforderungen an das Skript	28
5	Entwicklung des Skriptes	30
5.1	Klassenentwurf	30
5.2	Evaluation einzelner Methoden	30
5.3	Multikern-Verarbeitung der Daten	32
5.4	Auslagerung der Transformation in C++	33
5.5	Klassen	34
5.5.1	VdAutoStart (Python)	34
5.5.2	VdInterface (Python)	34
5.5.3	VdHardware (Python)	34
5.5.4	VdPoint (Python / C++)	35
5.5.5	VdDataset (Python / C++)	35
5.5.6	VdFile und Subklassen (Python / C++)	35
5.5.7	VdBuffer	36
5.5.8	VdTransformer	36
5.6	Steuerung des Skriptes	36
6	Konfiguration des Raspberry Pi	37
6.1	Installation von Raspbian	37
6.2	Befehle mit Root-Rechten	38
6.3	IP-Adressen-Konfiguration	38
6.4	Konfiguration als WLAN-Access-Point	39
6.5	Autostart des Skriptes	40
6.6	Einrichtung als Proxy	40
7	Systemüberprüfungen	42
7.1	Untersuchung der Gleichzeitigkeit von PPS-Signalen von verschiedenen GNSS-Empfängern	42
7.2	Messgenauigkeit des Laserscanners im Vergleich	42
7.2.1	Vergleichsmessung an einer Fassadenfront / im geodätischen Labor	44
7.2.2	Vergleichsmessung in der Tiefgarage	45
8	Ausblick	47
Literaturverzeichnis		48
Abbildungsverzeichnis		51

Tabellenverzeichnis	53
Anhang	54
A Python-Skripte	55
A.1 vdAutoStart.py	55
A.2 vdBuffer.py	63
A.3 vdTransformer.py	65
A.4 vdInterface.py	67
A.5 vdGNSSTime.py	69
A.6 vdHardware.py	71
A.7 vdFile.py	74
A.8 vdASCIIFile.py	76
A.9 vdTxtFile.py	77
A.10 vdObjFile.py	78
A.11 vdXYZFile.py	78
A.12 vdSQLite.py	79
A.13 vdDataset.py	80
A.14 vdPoint.py	83
A.15 config.ini	86
A.16 convTxt2Obj.py	87
A.17 convBin2Obj.py	88
B C++-Quelltexte	90
B.1 main.cpp	90
B.2 VdFile.h	92
B.3 VdFile.cpp	93
B.4 VdASCIIFile.h	94
B.5 VdASCIIFile.cpp	95
B.6 VdObjFile.h	95
B.7 VdObjFile.cpp	96
B.8 VdTxtFile.h	97
B.9 VdTxtFile.cpp	97
B.10 VdXYZFile.h	98
B.11 VdXYZFile.cpp	98
B.12 VdSQLite.h	99
B.13 VdSQLite.cpp	100
B.14 VdDataset.h	101
B.15 VdDataset.cpp	102
B.16 VdPoint.h	105

B.17	VdPoint.cpp	106
B.18	VdXYZ.h	108
B.19	VdXYZ.cpp	108
C	Arduino-Quelltext	111
C.1	ppsVergleich.ino	111
D	Beispieldateien	113
D.1	Rohdaten vom Scanner	113
D.2	Dateiformat für Datenspeicherung als Text	113
D.3	Dateiformat für Datenspeicherung als OBJ	114

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. (Acevedo Pardo et al., 2015)

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. (Acevedo Pardo et al., 2015)

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.

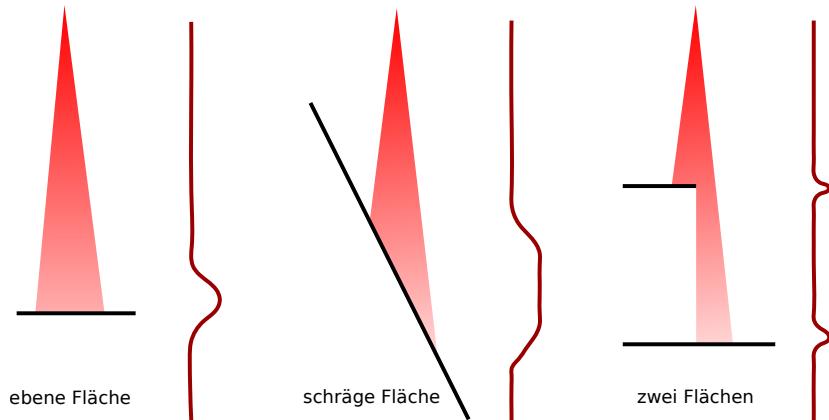


Abbildung 2.1: Reflektiertes Signal nach Oberfläche, nach Beraldin et al. (2010, S. 28)

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

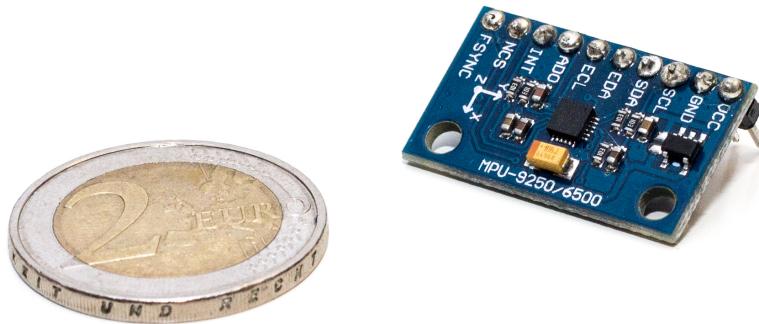


Abbildung 2.2: MPU-9250 - Low-Cost-MEMS-IMU-Modul wie es in vielen Consumer-Geräten und Multikoptern verwendet wird (schwarzes Bauteil mittig auf der Platine, eigene Aufnahme)

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 Gimbalen werden diese Sensoren eingesetzt. Für geodätische An-

wendungsfälle werden jedoch meist noch mechanische Systeme genutzt, da deren Genauigkeit und ihre zeitliche Abweichung geringer sind. Höherpreisige MEMS-Sensoren erreichen bereits gute Genauigkeiten.

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 Zeitstempel zu versehen.



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

Messgenau

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)

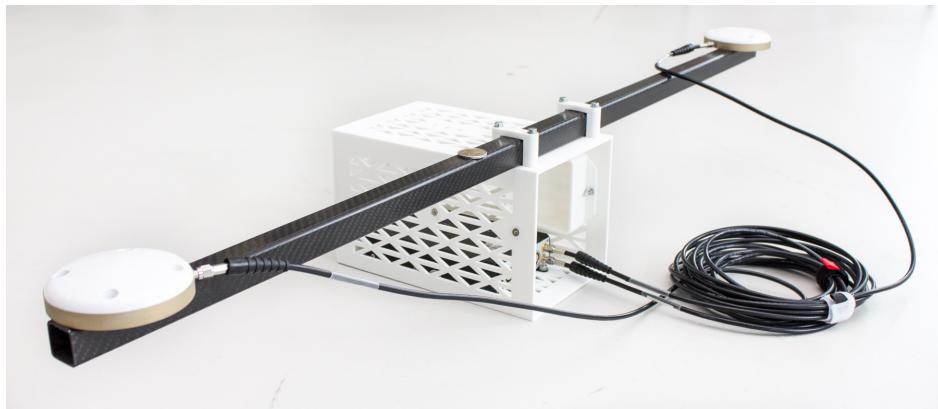


Abbildung 3.3: GNSS-Antennen des (links und rechts) iMAR iNAT-M200-Flat an Prototypen des modularen Gehäuses (eigene Aufnahme)

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). Im Betrieb ohne Peripherie wird dieser Stromverbrauch bei weitem nicht erreicht, hier liegt der Stromverbrauch bei etwa .

Messung

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. Sie wird eigentlich zur Stabilisierung von Aufnahmen mit digitalen Spiegelreflexkameras entwickelt. Sie hat eine Tragkraft von 2,3 Kilogramm. Außer den erwähnten Beschleunigungsmessern verwendet diese auch GNSS um das Driften der integrierten IMU zu vermindern. (Freefly, 2017)

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 Netzwerkschnitt-

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)

stelle und die Speicherung deutlich komplizierter und die Hardware nicht so leistungsfähig. Bei der Alternative, dem Raspberry-Pi übernimmt das Betriebssystem die grundlegenden Steuerungen, so dass nur noch die Daten selbst verarbeitet werden müssen. Außerdem bietet er mit der festverbauten Netzwerkschnittstelle und dem MicroSD-Karten- und der USB-Schnittstelle auch die komplett benötigte Hardware, die so nicht einzeln zusammengestellt und -gebaut werden muss.

3.3 Stromversorgung

Die Stromversorgung des Raspberry Pi an dem Multikopter 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 5 Volt stabilisierter Spannung abgeben werden können. Gegebenenfalls sind hierfür auch zwei komplett unabhängige Spannungsquellen zu nutzen.

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

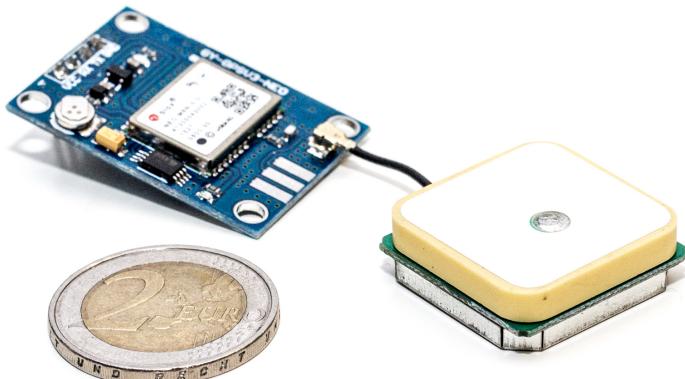


Abbildung 3.6: uBlox NEO-M8N, das Vorgängermodell NEO-6M mit PPS-Ausgang wurde verwendet (eigene Aufnahme)

3.4 Anbindung des Raspberry Pi an den Laserscanner

Durch seine vielseitigen Anschlussmöglichkeiten bildet der Raspberry Pi den Sternpunkt der Schnittstellen. Der Laserscanner wird mit einem RJ45-Kabel an der Netzwerkschnittstelle angeschlossen. Die inertiale Messeinheit zeichnet die Daten selbstständig auf, kann aber auch mittels der als serieller Schnittstelle nutzbaren GPIO-Pins an den Raspberry Pi angeschlossen werden. Außerdem kann an diesem Port auch ein GNSS-Modul angeschlossen werden. Dieses GNSS-Modul kann im Folgenden dem Raspberry Pi zu einer genauen Uhrzeit verhelfen, die für die Verarbeitung der Daten benötigt wird. Alternativ kann auch ein an den Laserscanner angeschlossenes GNSS-Modul sein Zeitstempel per Netzwerk an den Raspberry Pi liefern. Diese Methode soll hier verwendet werden.

3.5 Verbindung des GNSS-Modules zum Laserscanner

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

Die Übertragung der Daten des GNSS-Moduls 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 übertragen wird. Der direkte Anschluss eines uBlox GNSS-Modules vom Typ NEO-6M brachte zunächst keinen Erfolg. Messungen mit einem Arduino (siehe



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

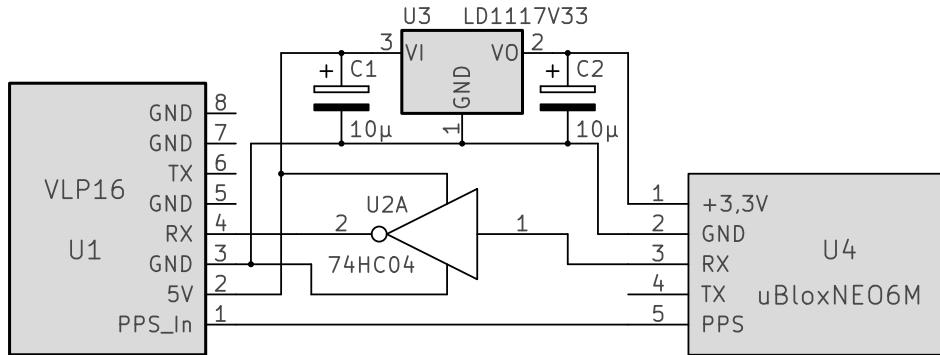


Abbildung 3.8: Entwurf der Schaltung zum Anschluss des GNSS-Modules an den Laserscanner

Abbildung 3.7) zeigten, dass das Signal des verwendeten GNSS-Moduls nicht dem im Datenblatt von Velodyne Lidar (2017a, S. 3) entsprach. Es zeigte sich, dass das Signal gedreht werden musste, da die Definition der Signalspannung verschieden war: Der Laserscanner benötigte ein Signal, bei dem Logisch 1 mit einer Spannung von über 3 Volt (Velodyne Lidar, 2017a, S. 3) codiert ist (HIGH), beim GNSS-Modul entspricht die höhere Spannung Logisch 0.

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

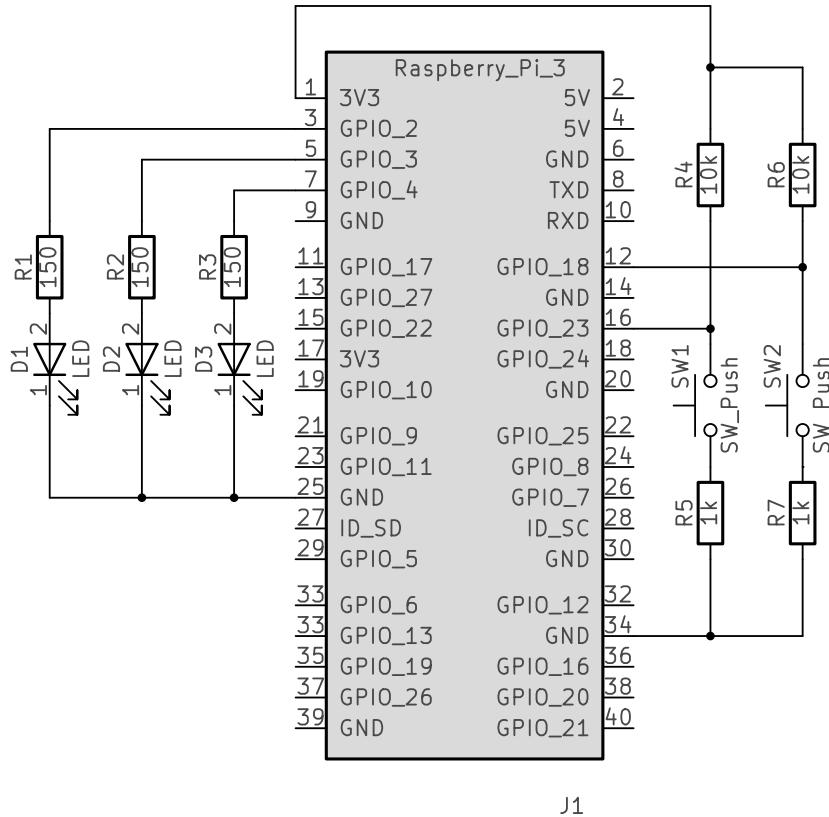


Abbildung 3.9: Entwurf der Schaltung für die Steuerung des Raspberry Pi

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 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 R4 und R6 auf ein High-Level gezogen. Durch Drücken des Tasters wird die Spannung über die Widerstände R5 und R7 auf ein Low-Level gezogen, welches durch das Python-Skript zur Laufzeit ausgelesen werden kann. Die Widerstände R5 und R7 dienen außerdem 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 dieses 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 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 verlötzten 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

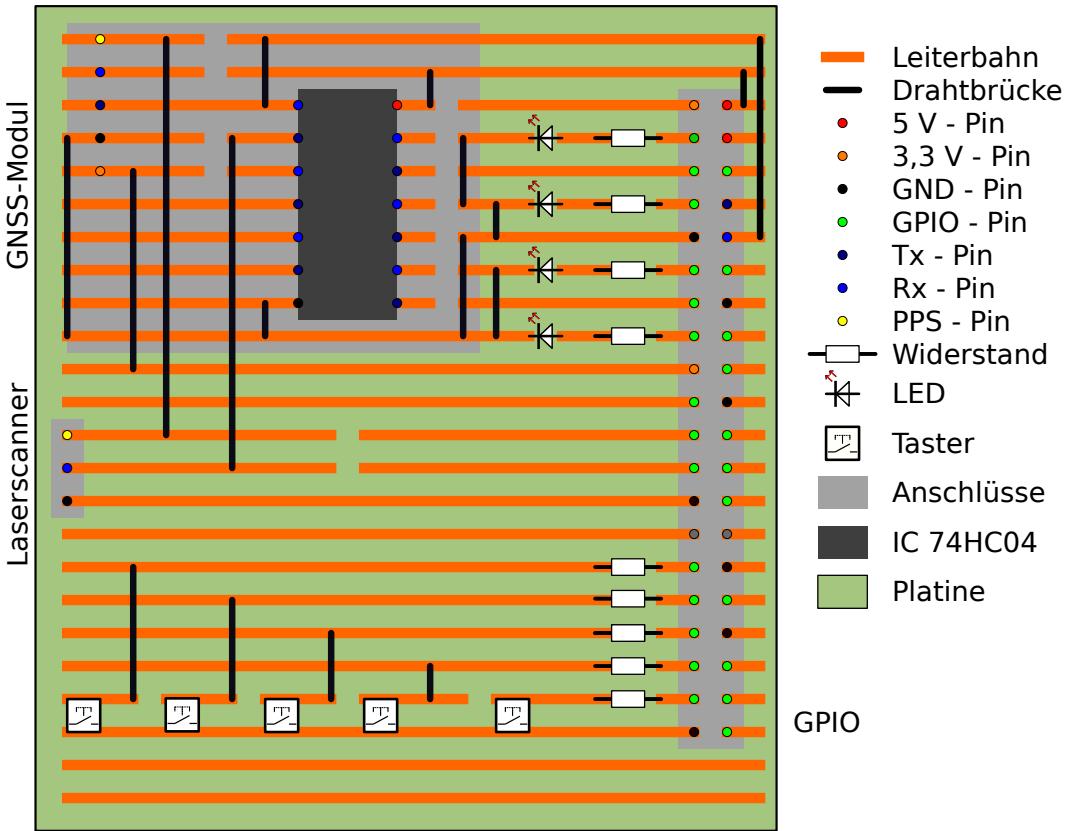


Abbildung 3.10: Layout der Lochstreifenplatine

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 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 anstatt hierfür einen zusätzlichen Spannungswandler zu verwenden. Die Platinengröße wurde so gewählt, dass die Platine direkt auf den Raspberry Pi 2 oder 3 aufgesteckt werden kann. 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. Es wurde Platz für weitere LEDs und Taster freigelassen.

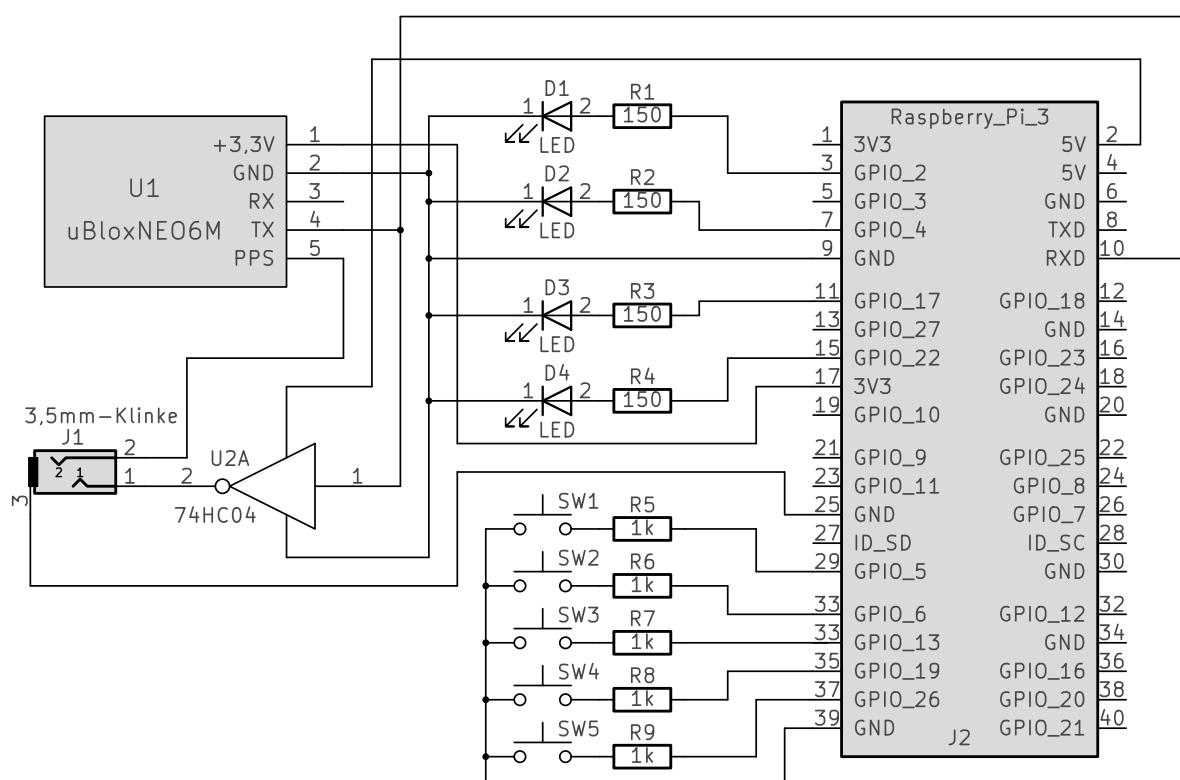


Abbildung 3.11: Endgültiger Schaltplan

4 Theoretische Datenverarbeitung

4.1 Verwendete Programmiersprachen

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 Programms 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. Außerdem sind die Skripte größtenteils plattformunabhängig.

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. Da sich viele Funktionen von der Version 2 zur 3 geändert haben, wurde auf die zuerst geplante Kompatibilität zu Python 2 verzichtet. 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.

C++ Ein großer Nachteil von Python ist die Ausführungsgeschwindigkeit. Daher wurden einzelne Teile des Quelltextes im Verlauf der Entwicklung in C++ umgeschrieben. Bei C++ handelt es sich um eine objektorientierte Erweiterung der Programmiersprache C. C wurde 1972 von Dennis Ritchie entwickelt. Anfang der 1980er Jahre erweiterte Bjarne Stroustrup C zu C++. C++ hat zwar eine höhere Ausführungsgeschwindigkeit, dafür ist C++ jedoch nicht so leicht zu erlernen. (Küveler & Schwoch, 2017, S. 4f)

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)

4.2 Datenlieferung vom Laserscanner

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

Ein Datenpaket (siehe Tabelle 4.1) besteht jeweils aus einem Header von 42 Bytes, gefolgt von 12 Datenblöcken mit jeweils 32 Messungen, abgeschlossen von 4 Bytes, die den Zeitstempel angeben und 2 Bytes, die den eingestellten Scan-Modus zurückliefern. Jeder Datenblock enthält die aktuelle horizontale Ausrichtung des rotierenden Lasers und darauf folgend die Messwerte von zwei Messungen der 16 Laserstrahlen. Die genaue Horizontalrichtung zum Zeitpunkt der Messung muss aus den Horizontalrichtungen aus zwei auf einander folgenden Messungen interpoliert werden.

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 Neigungs-

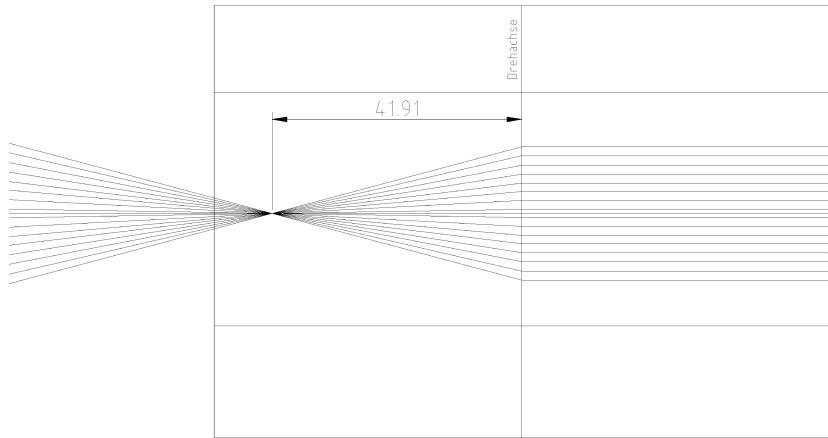


Abbildung 4.1: Strahlengang im Laserscanner VLP-16, Werte in Millimetern, nach Velodyne Lidar (2014)

daten 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.

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.

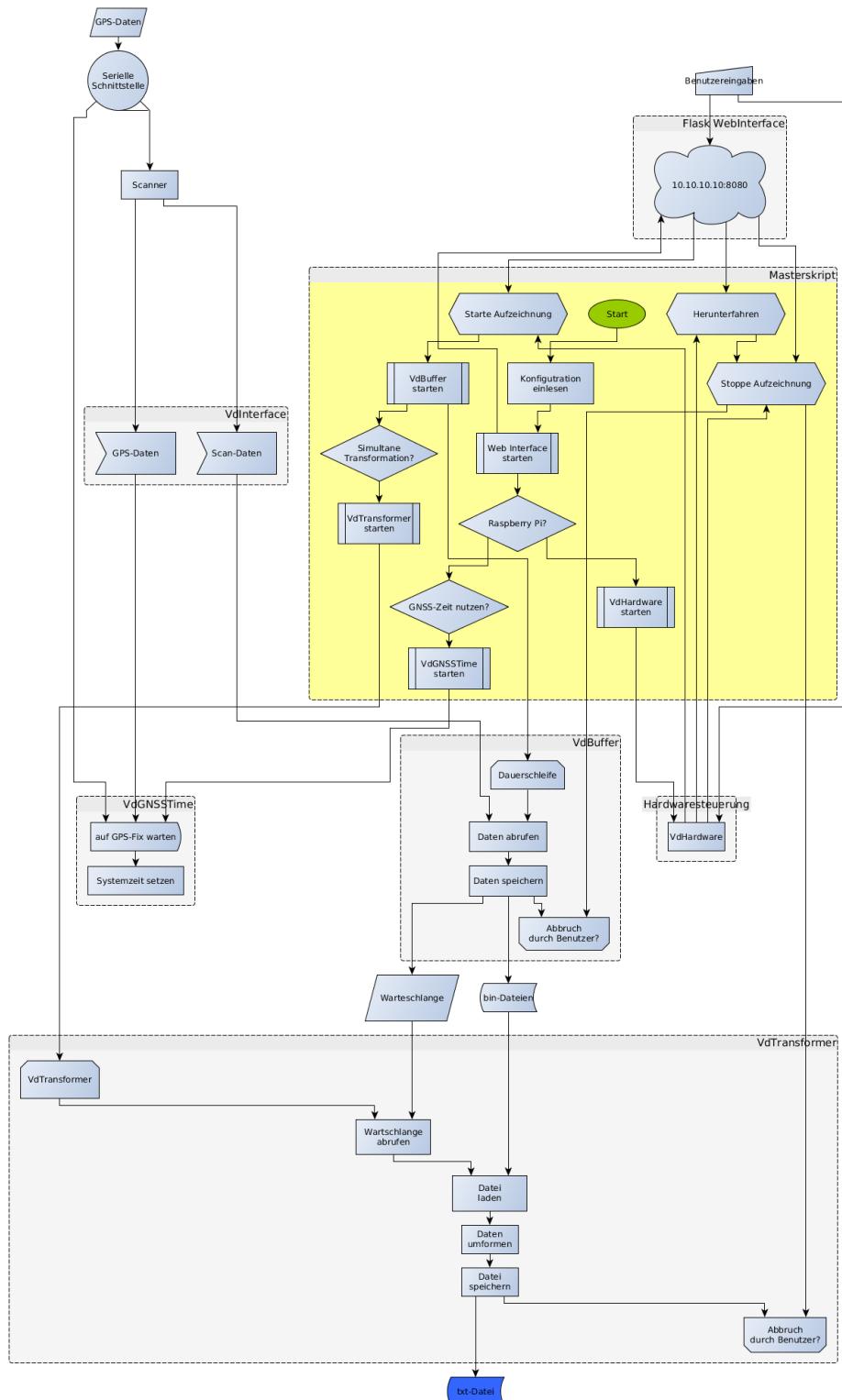


Abbildung 4.2: Vereinfachter Ablaufplan des Skriptes

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 5.5 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

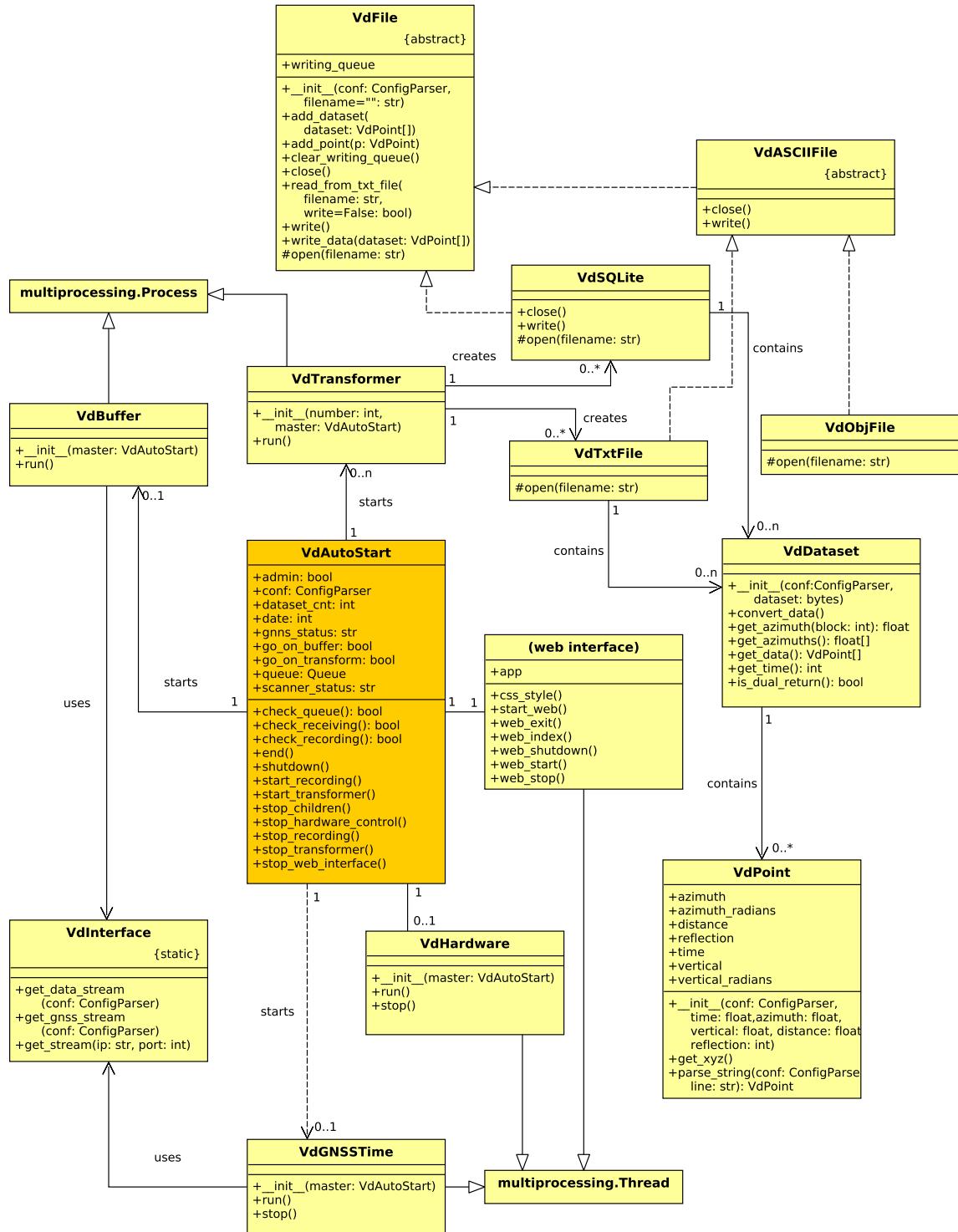


Abbildung 5.1: UML-Klassendiagramm

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 die Weboberfläche (Methode `startWeb()` in der 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. Sobald eine in der Konfiguration festgelegte Anzahl von Datensätzen zwischengespeichert wurde, 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 (abschaltbar). Damit die Umformer-Prozesse beim Schreiben nicht auf einander warten müssen, schreibt jeder Prozess in eine andere Ergebnisdatei. Diese können nach der Messung einfach zusammengeführt werden. Falls nun zum Beispiel die Stromversorgung unterbrochen wird, sind nun nur die Daten der letzten Sekunden verloren. Daten, die älter sind, sind entweder auch trotz Spannungsverlust als binäre Daten oder als Textdatei gespeichert. Durch ein Skript ist es möglich, den Umformerprozesses alleine zu starten und die restlichen, noch nicht gewandelten Daten umzuformen.

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 Auslagerung der Transformation in C++

Da auch bei Nutzung der kompletten Prozessorleistung des Raspberry Pi es nicht möglich war, die Daten zeitgleich zu transformieren und lange Nachbearbeitungszeiten auftraten, wurden weitere Ansätze geprüft. Am erfolgsversprechendsten war die Nutzung von C++ für diesen Teil. Die Programmiersprachen C und C++ sind bekannt dafür, dass sie sehr schnelle Ausführungszeiten bieten. Die Klassen, welche für die Transformation zuständig waren, wurden daher in C++ nachgebaut. Die Geschwindigkeit für die Umformung eines Testdatensatzes erfolgte in etwa der zehnfachen Geschwindigkeit. Diese Steigerung reicht aus, um bei Nutzung der vier Kerne des Raspberry Pi die Daten zeitgleich umzuformen. Der C++-Code wurde als eigenständiges Kommandozeilenprogramm erstellt. Somit ist es unabhängig vom Python-Code und kann

auch selbständig verwendet werden, zum Beispiel zum späteren Umformen von Daten. Die Pythonklasse `VdTransformer` wurde umschaltbar durch die Konfigurationsdatei um die Aufrufe des zusätzlichen Programmes erweitert. So kann je nach verwendeter Umgebung umgeschaltet werden zwischen der Umformung mittels Python (plattformunabhängig) und für verschiedene Plattformen kompilierter C++-Umformungen (ARM 32bit, Linux 64bit).

5.5 Klassen

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

5.5.1 VdAutoStart (Python)

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.

5.5.2 VdInterface (Python)

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

5.5.3 VdHardware (Python)

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.

5.5.4 VdPoint (Python / C++)

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

5.5.5 VdDataset (Python / C++)

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 Implementierungen der Klasse **VdFile** können diese dann zum Beispiel als Datei gespeichert werden.

5.5.6 VdFile und Subklassen (Python / C++)

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 sowie VdTxtFile, VdObjFile und 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 Punktwolke 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.

5.5.7 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.

5.5.8 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).

5.6 Steuerung des Skriptes

Die grundlegenden Einstellungen erfolgen über die Konfigurationsdatei `config.ini`. Hier können Einstellungen zum Verhalten des Skript durchgeführt werden, aber auch die Parameter des Laserscanners angepasst werden. Die weitere Steuerung erfolgt über die Weboberfläche des Skriptes und des Laserscanner.

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 neu gestartet 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 Administrator 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. Um zukünftige Updates einfach zu ermöglichen, müssen die statischen IP-Einstellungen (Zeile 8-11) im Normalfall abgeschaltet werden und die DHCP Einstellungen (Zeile 13) aktiviert werden. Raspberry

Listing 6.2: Konfiguration der /etc/network/interfaces

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

5 #Ethernet
6 auto eth0
7 ## for Velodyne
8 iface eth0 inet static
9   address 192.168.1.110
10  netmask 255.255.255.0
11  gateway 192.168.1.110
12 ## for Internet etc. (DHCP)
13 # iface eth0 inet dhcp

15 allow-hotplug wlan0
16 iface wlan0 inet static
17   address 10.10.10.10
18   netmask 255.255.255.0
19   network 10.10.10.0
```

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

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

Listing 6.3: Konfiguration der /etc/dnsmasq.conf

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

6.4 Konfiguration als WLAN-Access-Point

Um einen Zugriff auf die Python-Weboberfläche des Skriptes und die Konfiguration des Laserscanners zu ermöglichen, soll der Raspberry Pi selbst als WLAN-Access-Point fungieren. Hierzu 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. Durch die Nutzung des Befehls `su` wird der Befehl als Nutzer `pi`, also ohne Administratorrechte gestartet. Durch Weglassen von `su pi -c` kann dieser auch als Administrator gestartet werden. Dies hat jedoch sicherheitstechnische Nachteile, dafür können dann Prioritäten der Befehle gesetzt werden.

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.

6.6 Einrichtung als Proxy

Zusätzlich ist es sinnvoll, den Raspberry Pi auch als Proxy einzurichten. So lässt sich dann über die WLAN-Schnittstelle des Raspberry Pi auf die Konfigurationswebseite des Laserscanners zugreifen, um Einstellungen des Scanners mobil zu ändern.

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 (siehe Anhang C.1) 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 Zoller+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 (siehe Abbildung 7.2). 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.

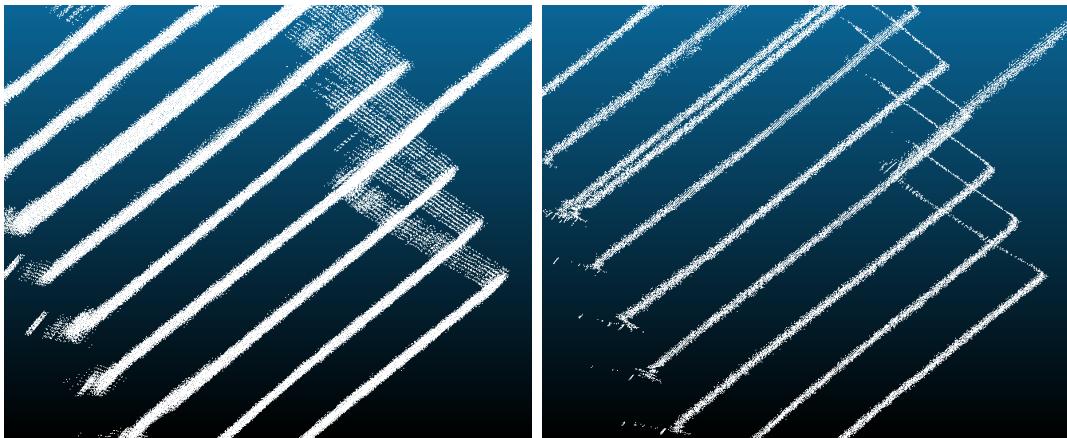


Abbildung 7.2: Punktfolke des Velodyne VLP-16 vor (links) und nach (rechts) der Korrektur der Horizontalrichtung am Beispiel eines Pfeilers in der Tiefgarage

7.2.1 Vergleichsmessung an einer Fassadenfront / im geodätischen Labor

Die erste Messung zur Überprüfung der Entfernungsmessung fand vor einer weißen Fassade der HafenCity Universität statt. Durch einen Fehler im Auswerteskript, der erst bei großen Strecken auffiel, waren die Messdaten jedoch leider nicht zu verwenden. Es wurde daher eine ähnliche Messung im geodätischen Labor nachgeholt (siehe Abbildung 7.3). Hier wurde der Velodyne VLP-16 mittig im Raum aufgestellt. Zwei Stellwände wurden zur Überprüfung der Entfernungsmessung verwendet. Eine blieb hierbei immer an der gleichen Stelle, etwa Meter vom Velodyne entfernt stehen, die zweite wurde in verschiedenen Entfernung aufgestellt (zwischen Meter). Die Messszenarios wurden jeweils mit dem Imager 5010 von Zoller+Fröhlich dokumentiert. Bei Messungen auf weiße Oberflächen liegt die Streckenmessgenauigkeit des Imager 5010 unter einem Millimeter (GmbH, 2017). Zum Vergleich mit dem Velodyne VLP-16, der eine angegebene Genauigkeit von 3 Zentimetern hat (Velodyne Lidar, 2017b), wird die Messung des Imagers daher als wahrer Wert angenommen.

Nachbearbeitung der Messdaten Die Punktfolken des Imager 5010 wurden in der Software Z+F LaserControl des Herstellers über angebrachten Targets zusammengeführt. Anschließend wurden die Punktfolken 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 Punktfolke automatisch eingepasst wurde. Hieraus konnte dann durch Bestimmung der Mittelachse und der Daten des Fokuspunktes der Nullpunkt der Punktfolken des VLP-16 bestimmt werden. In den Laserpunktfolken der beiden Scanner wurden in Geomagic Wrap die Wände, Decke, Fußboden und die

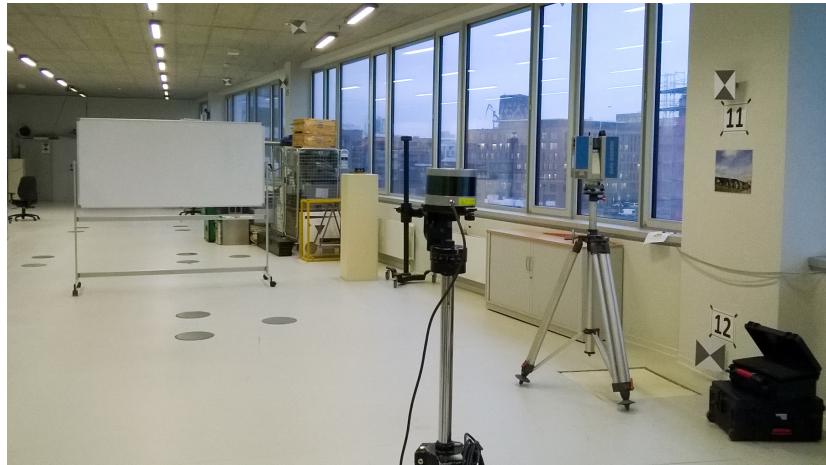


Abbildung 7.3: Messung im geodätischen Labor

Stellwände modelliert. Hierfür wurden die entsprechenden Punkte ausgewählt und mittels Best-Fit-Anpassung Ebenen in den Punkten ausgeglichen. Über die Wände und Decken wurden die Punktwolken nochmals genauer zu einander ausgerichtet.

Präzision der Streckenmessung Beim Modellieren der Wände bestand die Möglichkeit, sich die Genauigkeit der Ausgleichung in den Punkten anzuzeigen. Wie vom Hersteller des Velodyne angegeben, betrug die Standardabweichung dieser Punkte zur durch Ausgleichung bestimmten Ebene bei maximal 3 Zentimetern.

Richtigkeit der Streckenmessung Um zu Überprüfen, ob nicht nur die Wiederholungsgenauigkeit des Velodyne VLP-16 im angegeben Bereich liegt, wurden die Positionen der modellierten Stellwände aus den Datensätzen des Velodyne mit denen des Imager 5010 verglichen. Hierzu wurden die Abstände zwischen den Flächen aus beiden Messverfahren jeweils einer Stellwand in CAD bestimmt.

weiter...

7.2.2 Vergleichsmessung in der Tiefgarage

Hauptsächlich zur Überprüfung der Genauigkeit der Winkelmessung, aber auch zur Überprüfung der Streckenmessung auf maximaler Entfernung wurde eine weitere Messung in der Tiefgarage durchgeführt. Durch die vielen klaren, rechtwinkligen Kanten, sind hier viele Möglichkeiten um Kanten zu detektieren gegeben. Zum Vergleich wurde die Tiefgarage mit einem Trimble S7 mit einer Winkelgenauigkeit von und einer prismafreien Streckengenauigkeit von 2 mm + 2 ppm verwendet (Inc., 2017).

Genauigke

Auswertung Aus den Daten der Tachymettermessung wurden die Wände und Pfeiler der Tiefgarage modelliert. Durch Anpassung der Laserscandaten an diesen Grundriss wurde die Orientierung des Velodyne VLP-16 bestimmt. Die Neigung wurde wiederum

durch Ausgleichung der Fußbodenebene durchgeführt. Auch aus den Laserpunktswolken des Velodyne Vlp-16 wurden die Ecken modelliert. Hierfür würden jeweils Ebenen in die Punkte der Pfeiler- und Wandoberflächen in Geomagic Wrap mittels Ausgleichung eingepasst.

weiter...

8 Ausblick

Weiterverarbeitung der Daten Die Umformung und Speicherung der Daten war nur ein erster Schritt zur Entwicklung des Airborne Laserscanning Systems. Es ist so zwar möglich, die Daten vom Laserscanner, der inertialen Messeinheit und des GNSS zu speichern, jedoch müssen diese Daten weiter prozessiert werden. Bisher wurde der Scanner nur stationär zur Messung eingesetzt. Um ihn auch kinematisch nutzen zu können, müssen aus den Daten der IMU die Bewegungen rekonstruiert werden. Hiermit können dann die Messungen in ein System überführt werden. Diese Verarbeitung wird in einer Folgearbeit behandelt werden.

Anwendbarkeit des Messsystems Da das fertige Messsystem keine Abhängigkeiten vom Multikopter hat, kann es nicht nur fliegend, sondern auch terrestrisch verwendet werden. Einzige Bedingung der Nutzung ist die freie Sicht zum Himmel für den GNSS-Empfang. So wäre auch eine Montage des Systems, gegebenenfalls auch unter Nutzung des Gimbal, an andere, bodengebundene Fahrzeuge denkbar. So ließen sich zum Beispiel zur Digitalisierung von Straßenzügen für 3D-Stadtmodelle zuerst die Straßen und Wege mit Autos, Fahrrädern oder Bollerwagen bodengebunden aufnehmen und mit dem gleichen Messsystem anschließend mit dem Multikopter eine Messung von oben durchführen. So wäre mit einem Messesystem die nahtlose Erfassung von Gebäudehüllen möglich.

Erweiterungs- und Optimierungsmöglichkeiten In der Programmierung des Raspberry Pi sind natürlich noch Verbesserungen und Optimierungen möglich. Eine sinnvolle Erweiterung ist zum Beispiel in der Konfiguration des Raspberry Pis die Einrichtung als Proxyserver für den Laserscanner, um auch hier Einstellungen mobil durchführen zu können. Programmiertechnisch könnten die restlichen Skripte noch in ihrer Geschwindigkeit der Ausführung optimiert werden.

Erweiteren

Literaturverzeichnis

- Acevedo Pardo, Carlos; Farjas Abadía, Mercedes; Sternberg, Harald (2015): Design and development of a low-cost modular Aerial Mobile Mapping System. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, Band XL-1/W4:S. 195–199.
- Bachfeld, Daniel (März 2013): Quadrokopter-Know-how. *c't Hacks*, (03/2013):S. 42 – 53.
- 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.
- Freefly (2017): MöVI M5. <http://freeflysystems.com/movi-m5>. (Aufruf: 02. Dez. 2017).
- GmbH, Zoller + Fröhlich (2017): Z+F IMAGER 5010 Datenblatt. http://www.zf-laser.com/fileadmin/editor/Datenblaetter/Z_F_IMAGER_5010_Datenblatt_D.pdf. (Aufruf: 02. Dez. 2017).
- Heise Online (2017): Quadrocopter - Drohnen & Multikopter. <https://www.heise.de/thema/Quadrocopter>. (Aufruf: 27. Okt. 2017).
- iMAR Navigation GmbH (2015): iNAT-M200-FLAT. (Datenblatt).
- Inc., Trimble (2017): Datasheet Trimble S7 Total Station.

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.

Küveler, Gerd; Schwoch, Dietrich (2017): C/C++ für Studium und Beruf - Eine Einführung mit vielen Beispielen, Aufgaben und Lösungen. Springer-Verlag, Berlin Heidelberg New York, ISBN 978-3-658-18581-7.

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. Okt. 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. Okt. 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. Okt. 2017).

Schulz, Jasper (2016): Aufbau und Betrieb eines Zeilenlaserscanners an einem Multikopter. (Masterthesis, 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. Okt. 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. (Aufruf: 22. Okt. 2017).

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

Wilken, Mathias (2017): Untersuchung der RTK-Performance des INS/GNSS iNAT M200 Systems. (Bachelorthesis, 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)	12
3.2	iMAR iNAT-M200-Flat im Prototypen des modularen Gehäuses, Leitungen führen zu den GNSS-Antennen (eigene Aufnahme)	13
3.3	GNSS-Antennen des (links und rechts) iMAR iNAT-M200-Flat an Prototypen des modularen Gehäuses (eigene Aufnahme)	13
3.4	Raspberry Pi 3 (eigene Aufnahme)	15
3.5	Multikopter Copterproject CineStar 6HL mit Gimbal Freefly MöVI M5 (eigene Aufnahme)	16
3.6	uBlox NEO-M8N, das Vorgängermodell NEO-6M mit PPS-Ausgang wurde verwendet (eigene Aufnahme)	18
3.7	Messung des Signals am uBlox NEO-6M (grün: Ausgangssignal; rot: Signal nach Nutzung eines Pegelwandler; 1000 Punkte entsprechen 5 Volt)	19
3.8	Entwurf der Schaltung zum Anschluss des GNSS-Modules an den Laser-scanner	19
3.9	Entwurf der Schaltung für die Steuerung des Raspberry Pi	20
3.10	Layout der Lochstreifenplatine	22
3.11	Endgültiger Schaltplan	23
4.1	Strahlengang im Laserscanner VLP-16, Werte in Millimetern, nach Velodyne Lidar (2014)	27
4.2	Vereinfachter Ablaufplan des Skriptes	29
5.1	UML-Klassendiagramm	31
7.1	Vergleichsmessungen mit dem Velodyne VLP-16, Zoller+Fröhlich Imager 5010 und dem Trimble S7	43
7.2	Punktwolke des Velodyne VLP-16 vor (links) und nach (rechts) der Korrektur der Horizontalrichtung am Beispiel eines Pfeilers in der Tiefgarage	44

7.3 Messung im geodätischen Labor	45
---	----

Tabellenverzeichnis

3.1	Spannungs- und Strombedarf der einzelnen Module (Velodyne Lidar, 2017b; iMAR Navigation GmbH, 2015; RS Components Limited, 2015)	17
4.1	Aufbau der Daten des Netzwerkpaketes, nach Velodyne Lidar (2016) . . .	25
6.1	IP-Adressen-Verteilung	38

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 vdBuffer import VdBuffer
21 from vdTransformer import VdTransformer

23 from vdGNSSTime import VdGNSSTime

26 class VdAutoStart(object):

28     """ main script for automatic start """

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

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

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

```

46         self.__pTransformer = None
47
48         # pipes for child processes
49         manager = Manager()
50         self.__gnss_status = "(unknown)"
51         # self.__gnssReady = manager.Value('gnssReady',False)
52         self.__go_on_buffer = manager.Value('_go_on_buffer', False)
53         self.__go_on_transform = manager.Value('_go_on_transform', False)
54         self.__scanner_status = manager.Value('__scanner_status', "(unknown)")
55         self.__dataset_cnt = manager.Value('__dataset_cnt', 0)
56         self.__date = manager.Value('__date', None)
57
58         # queue for transformer
59         self.__queue = Queue()
60
61         # attribute for web interface
62         self.__web_interface = web_interface
63
64         # check admin
65         try:
66             os.rename('/etc/foo', '/etc/bar')
67             self.__admin = True
68         except IOError:
69             self.__admin = False
70
71         # check raspberry pi
72         try:
73             import RPi.GPIO
74             self.__raspberry = True
75         except ModuleNotFoundError:
76             self.__raspberry = False
77
78         self.__gnss = None
79
80     def run(self):
81         """ start script """
82
83         # handle SIGINT
84         signal.signal(signal.SIGINT, self.__signal_handler)
85
86         # use hardware control on raspberry pi
87         if self.__raspberry:
88             print("Raspberry Pi was detected")
89             from velodyneInterface.vdHardware import VdHardware
90             self.__vd.hardware = VdHardware(self)
91             self.__vd.hardware.start()
92         else:
93             print("Raspberry Pi could not be detected")
94             print("Hardware control deactivated")
95
96         # set time by using gnss
97         if self.__conf.get("functions", "use_gnss_time") == "True":
98             self.__gnss = VdGNSSTime(self)
99             self.__gnss.start()
100
101    def start_transformer(self):
102        """ Starts transformer processes"""
103        print("Start transformer...")
104        # number of transformer according number of processor cores

```

```

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

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

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

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

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

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

```

```

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

```

```

223         if x - y > 0:
224             return True
225         return False
226
227     # getter/setter methods
228     def __get_conf(self):
229         """
230             Gets config file
231             :return: config file
232             :rtype: configparser.ConfigParser
233         """
234
235         return self.__conf
236     conf = property(__get_conf)
237
238     def __get_gnss_status(self):
239         """
240             Gets GNSS status
241             :return: GNSS status
242             :rtype: Manager
243         """
244
245         return self.__gnss_status
246
247     def __set_gnss_status(self, gnss_status):
248         """
249             Sets GNSS status
250             :param gnss_status: gnss status
251             :type gnss_status: str
252         """
253
254         self.__gnss_status = gnss_status
255     gnss_status = property(__get_gnss_status, __set_gnss_status)
256
257     def __get_go_on_buffer(self):
258         """
259             Should Buffer buffer data?
260             :return: go on buffering
261             :rtype: Manager
262         """
263
264         return self.__go_on_buffer
265     go_on_buffer = property(__get_go_on_buffer)
266
267     def __get_go_on_transform(self):
268         """
269             Should Transformer transform data?
270             :return: go on transforming
271             :rtype: Manager
272         """
273
274         return self.__go_on_transform
275     go_on_transform = property(__get_go_on_transform)
276
277     def __get_scanner_status(self):
278         """
279             Gets scanner status
280             :return:
281             :rtype: Manager
282         """
283
284         return self.__scanner_status
285     scanner_status = property(__get_scanner_status)
286
287     def __get_dataset_cnt(self):

```

```

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

290     def __get_date(self):
291         """
292             Gets recording start time
293             :return: timestamp starting recording
294             :rtype: Manager
295         """
296         return self.__date

298     def __set_date(self, date):
299         """
300             Sets recording start time
301             :param date: timestamp starting recording
302             :type date: datetime
303         """
304         self.__date = date
305         #: recording start time
306 date = property(__get_date, __set_date)

308     def __is_admin(self):
309         """
310             Admin?
311             :return: Admin?
312             :rtype: bool
313         """
314         return self.__admin
315 admin = property(__is_admin)

317     def __get_queue(self):
318         """
319             Gets transformer queue
320             :return: transformer queue
321             :rtype: Queue
322         """
323         return self.__queue
324 queue = property(__get_queue)

327 # web control
328 app = Flask(__name__)

331 @app.route("/")
332 def web_index():
333     """ index page of web control """
334     runtime = "(inactive)"
335     pps = "(inactive)"
336     if ms.date.value is not None:
337         time_diff = datetime.now() - ms.date.value
338         td_sec = time_diff.seconds + \
339             (int(time_diff.microseconds / 1000) / 1000.)
340         seconds = td_sec % 60

```

```

341     minutes = int((td_sec // 60) % 60)
342     hours = int(td_sec // 3600)

344     runtime = '{:02d}:{:02d}:{:06.3f}'.format(hours, minutes, seconds)

346     pps = '{:.0f}'.format(ms.dataset_cnt.value / td_sec)

348     elif ms.go_on_buffer.value:
349         runtime = "(no data)"

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     else:
381         output += """<a href="/start" id="start">
382             Start recording</a><br />"""
383     output += """
384         <a href="/exit" id="exit">Terminate script<br />
385             (control by SSH available only)</a></td></tr><br />
386         <a href="/shutdown" id="shutdown">Shutdown Raspberry Pi</a>
387     </content>
388     </body>
389     </html>"""

390     return output

393 @app.route("/style.css")
394 def css_style():
395     """ css file of web control """
396     return """
397     body, html, content {
398         text-align: center;

```

```

399      }
400
401      content {
402          max-width: 15cm;
403          display: block;
404          margin: auto;
405      }
406
407      table {
408          border-collapse: collapse;
409          width: 90%;
410          margin: auto;
411      }
412
413      td {
414          border: 1px solid black;
415          padding: 1px 2px;
416      }
417
418      td#column1 {
419          width: 30%;
420      }
421
422      a {
423          display: block;
424          width: 90%;
425          padding: 0.5em 0;
426          text-align: center;
427          margin: auto;
428          color: #fff;
429      }
430
431      a#stop {
432          background-color: #e90;
433      }
434
435      a#shutdown {
436          background-color: #b00;
437      }
438
439      a#start {
440          background-color: #1a1;
441      }
442
443      a#exit {
444          background-color: #f44;
445      }
446      """
447
448
449  @app.route("/shutdown")
450  def web_shutdown():
451      """ web control: shutdown """
452      ms.shutdown()
453      return """
454      <meta http-equiv="refresh" content="3; URL=/">
455      Shutdown..."""

```

```

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

```

```

18  class VdBuffer(Process):
19
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
102                     datasets_in_buffer += 1
103
104                     self.__datasets.value += measurements_per_dataset
105
106                     # safe data to file every 1500 datasets
107                     # (about 5 or 10 seconds)
108                     if (datasets_in_buffer >= 1500) or \
109                         (not self.__go_on_buffering.value):
110
111                         # write file
112                         f = open(
113                             self.__folder + "/" + str(self.__file_no) + ".bin",
114                             "wb")
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                         if data == 'QUIT':
130                             break
131
132             except socket.timeout:
133                 print("No data")
134                 continue
135
136             sock.close()
137             self.__scanner_status.value = "recording stopped"
138             print("Disconnected!")

```

A.3 vdTransformer.py

```

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

4  """
5  @author: Florian Timm
6  @version: 2017.11.20
7  """
8  import os
9  import signal
10 import subprocess
11 from multiprocessing import Process
12 from queue import Empty

14 from vdDataset import VdDataset

16 from vdTxtFile import VdTxtFile

19 class VdTransformer(Process):
21     """ Process for transforming data from Velodyne VLP-16 """
23     def __init__(self, number, master):
24         """
25             Constructor
26             :param number: number of process
27             :type number: int
28             :param master: instance of VdAutoStart
29             :type master: VdAutoStart
30         """
32     # constructor of super class
33     Process.__init__(self)

35     self.__queue = master.queue
36     self.__number = number
37     self.__admin = master.admin
38     self.__go_on_transform = master.go_on_transform
39     self.__conf = master.conf

41     @staticmethod
42     def __signal_handler(sig_no, frame):
43         """
44             handles SIGINT-signal
45             :param sig_no: signal number
46             :type sig_no: int
47             :param frame: execution frame
48             :type frame: frame
49         """
50     del sig_no, frame
51     # self.master.end()
52     print("SIGINT vdTransformer")

54     def run(self):
55         """ starts transforming process """
56         signal.signal(signal.SIGINT, self.__signal_handler)

58     if self.__admin:
59         os.nice(-15)

```

```

61     old_folder = ""
62
63     print ("Transformer started!")
64     while self.__go_on_transform.value:
65         try:
66             # get file name from queue
67             filename = self.__queue.get(True, 2)
68             folder = os.path.dirname(filename)
69             trans = self.__conf.get("file", "transformer")
70             if trans == "python":
71                 if dir != old_folder:
72                     vd_file = VdTxtFile(
73                         self.__conf,
74                         folder + "/obj_file" + str(self.__number))
75                     old_folder = folder
76
77             f = open(filename, "rb")
78
79             # count number of datasets
80             file_size = os.path.getsize(f.name)
81             dataset_cnt = int(file_size / 1206)
82
83             for i in range(dataset_cnt):
84                 # read next
85                 vd_data = VdDataset(self.__conf, f.read(1206))
86
87                 # convert data
88                 vd_data.convert_data()
89
90                 # add them on writing queue
91                 vd_file.add_dataset(vd_data.get_data())
92
93                 # write file
94                 vd_file.write()
95                 # close file
96                 f.close()
97                 break
98             elif trans == "linux64":
99                 result = subprocess.run(['./vdTrans_linux64', "bin", filename,
100                                         "txt", new_file], stdout=subprocess.PIPE)
101                 print(result.stdout.decode('utf-8'))
102             elif trans == "arm320":
103                 result = subprocess.run(['./vdTrans_arm32', "bin", filename,
104                                         "txt", new_file], stdout=subprocess.PIPE)
105                 print(result.stdout.decode('utf-8'))
106             pass
107
108             # delete binary file
109             if self.__conf.get("file", "deleteBin") == "True":
110                 os.remove(f.name)
111             except Empty:
112                 print("Queue empty!")
113                 continue

```

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):
14
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         return VdInterface.get_stream(conf.get("network", "UDP_IP"),
27                                       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         return VdInterface.get_stream(conf.get("network", "UDP_IP"),
39                                       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()

```

```

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!')
74
75         # now keep talking with the client
76         print('Listening on: ' + ip + ':' + str(port))
77
78         return sock

```

A.5 vdGNSSTime.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 os
10 import socket
11 from datetime import datetime
12 from threading import Thread
13
14 import serial
15 from vdInterface import VdInterface
16
17
18 class VdGNSSTime(Thread):
19
20     """ system time by gnss data """
21
22     def __init__(self, master):
23         """
24             Constructor
25             :param master: instance of VdAutoStart
26             :type master: VdAutoStart
27         """
28         Thread.__init__(self)
29         self.__tScanner = None
30         self.__tSerial = None
31         self.__master = master
32         self.__conf = master.conf
33         self.__time_corrected = False
34
35     def run(self):
36         """
37             starts threads for time detection

```

```

38     """
39     # get data from serial port
40     self.__tSerial = Thread(target=self.__get_gnss_time_from_serial())
41     self.__tSerial.start()
42
43     # get data from scanner
44     self.__tScanner = Thread(target=self.__get_gnss_time_from_scanner())
45     self.__tScanner.start()
46
47     self.__master.gnss_status = "Connecting..."
48
49     def __get_gnss_time_from_scanner(self):
50         """ gets data by scanner network stream """
51         sock = VdInterface.get_gnss_stream(self.__conf)
52         sock.settimeout(1)
53         self.__master.gnss_status = "Wait for fix..."
54         while not self.__time_corrected:
55             try:
56                 data = sock.recvfrom(2048)[0] # buffer size is 2048 bytes
57                 message = data[206:278].decode('utf-8', 'replace')
58                 if self.__get_gnss_time_from_string(message):
59                     break
60             except socket.timeout:
61                 continue
62             # else:
63             #     print(message)
64             if data == 'QUIT':
65                 break
66         sock.close()
67
68     # noinspection PyArgumentList
69     def __get_gnss_time_from_serial(self):
70         """ get data by serial port """
71         ser = None
72         try:
73             port = self.__conf.get("serial", "GNSSport")
74             ser = serial.Serial(port, timeout=1)
75             self.__master.gnss_status = "Wait for fix..."
76             while not self.__time_corrected:
77                 line = ser.readline()
78                 message = line.decode('utf-8', 'replace')
79                 if self.__get_gnss_time_from_string(message):
80                     break
81                 # else:
82                 #     print(message)
83             except serial.SerialTimeoutException:
84                 pass
85             except serial.serialutil.SerialException:
86                 print("Could not open serial port!")
87         finally:
88             if ser is not None:
89                 ser.close()
90
91     def __get_gnss_time_from_string(self, message):
92         if message[0:6] == "$GPRMC":
93             p = message.split(",")
94             if p[2] == "A":
95                 print("GNSS-Fix")
96                 timestamp = datetime.strptime(p[1] + "D" + p[9],

```

```

97                                     '%H%M%S.000%d%m%y')
98             self.__set_system_time(timestamp)
99             self.__time_corrected = True
100            self.__master.gnss_status = "Got time!"
101            return True
102        return False

104    def __set_system_time(self, timestamp):
105        """
106            sets system time
107            :param timestamp: current timestamp
108            :type timestamp: datetime
109            :return:
110            :rtype:
111        """
112        os.system("timedatectl set-ntp 0")
113        os.system("timedatectl set-time \"{}\" +".format(
114            timestamp.strftime("%Y-%m-%d %H:%M:%S") + "\")")
115        os.system("timedatectl set-ntp 1")
116        self.__master.set_gnss_status("System time set")

118    def stop(self):
119        """
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 -*-

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

9  import RPi.GPIO as GPIO
10 import time
11 from threading import Thread
12 import threading

15 class VdHardware(Thread):

17     """ controls hardware control, extends Thread """

19     def __init__(self, master):
20         """
21             Constructor
22             :param master: instance of VdAutoStart
23             :type master: VdAutoStart
24         """
25         Thread.__init__(self)

27         GPIO.setmode(GPIO.BCM)

29         self.__taster1 = 18 # start / stop
30         self.__taster2 = 25 # shutdown

```

```

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
42
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
49         # activate outputs
50         for l in self.__led:
51             GPIO.setup(l, GPIO.OUT)    # GPS-Fix
52             GPIO.output(l, GPIO.LOW)
53
54         self.__go_on = True
55
56     def run(self):
57         """ run thread and start hardware control """
58         GPIO.add_event_detect(
59             self.__taster1,
60             GPIO.FALLING,
61             self.__button1_pressed)
62         GPIO.add_event_detect(
63             self.__taster2,
64             GPIO.FALLING,
65             self.__button1_pressed)
66
67         self.__timer_check_leds()
68
69     def __timer_check_leds(self):
70         """ checks LEDs every second """
71         self.__check_leds()
72         if self.__go_on:
73             t = threading.Timer(1, self.__timer_check_leds)
74             t.start()
75
76     def __check_leds(self):
77         """ check LEDs """
78         self.__set_recording(self.__master.check_recording())
79         self.__set_receiving(self.__master.check_receiving())
80         self.__set_queue(self.__master.check_queue())
81
82     def __button1_pressed(self):
83         """ raised when button 1 is pressed """
84         time.sleep(0.1)    # contact bounce
85
86         # > 2 seconds
87         wait = GPIO.wait_for_edge(self.__taster1, GPIO.RISING, timeout=1900)
88
89         if wait is None:

```

```

90         # no rising edge = pressed
91         if self.__master.go_on_buffer.value:
92             self.__master.stop_recording()
93         else:
94             self.__master.start_recording()

96     def __button2_pressed(self):
97         """ raised when button 1 is pressed """
98         time.sleep(0.1) # contact bounce

100        # > 2 seconds
101        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()

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)

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)

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()

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()

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 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.27
7  """
8
9  import datetime
10 from abc import abstractmethod, ABC
11
12 from vdPoint import VdPoint
13
14
15 class VdFile(ABC):
16
17     """ abstract class for saving data """
18
19     def __init__(self, conf, filename=""):
20         """
21             Creates a new ascii-file
22             :param conf: configuration file
23             :type conf: configparser.ConfigParser
24             :param filename: name and path to new file
25             :type filename: str
26         """
27         self.__conf = conf
28         # create file
29         self.__writing_queue = []
30         self._open(filename)
31
32     def __get_writing_queue(self):
33         """
34             Returns points in queue
35             :return: points in queue
36             :rtype: VdPoint[]
37         """
38         return self.__writing_queue
39
40     writing_queue = property(__get_writing_queue)
41
42     def clear_writing_queue(self):
43         """
44             Clears writing queue """
45         self.__writing_queue = []

```

```

46     def _make_filename(self, file_format, file_name=""):
47         """
48             generates a new file_name from timestamp
49             :param file_format: file suffix
50             :type file_format: str
51             :return: string with date and suffix
52             :rtype: str
53             """
54         if file_name == "":
55             name = self._conf.get("file", "namePre")
56             name += datetime.datetime.now().strftime(
57                 self._conf.get("file", "timeFormat"))
58             name = "." + file_format
59             return name
60         elif not file_name.endswith("." + file_format):
61             return file_name + "." + file_format
62         return file_name
63
64     def write_data(self, data):
65         """
66             adds data and writes it to file
67             :param data: ascii data to write
68             :type data: VdPoint []
69             """
70         self.add_dataset(data)
71         self.write()
72
73     def add_point(self, p):
74         """
75             Adds a point to write queue
76             :param p: point
77             :type p: VdPoint
78             """
79         self._writing_queue.append(p)
80
81     def add_dataset(self, dataset):
82         """
83             adds multiple points to write queue
84             :param dataset: multiple points
85             :type dataset: VdPoint []
86             """
87         self._writing_queue.extend(dataset)
88
89     def read_from_txt_file(self, filename, write=False):
90         """
91             Parses data from txt file
92             :param filename: path and filename of txt file
93             :type filename: str
94             :param write: write data to new file while reading txt
95             :type write: bool
96             """
97         txt = open(filename)
98
99         for no, line in enumerate(txt):
100             try:
101                 p = VdPoint.parse_string(self._conf, line)
102                 self.writing_queue.append(p)
103                 # print("Line {0} was parsed".format(no + 1))

```

```

104         except ValueError as e:
105             print("Error in line {0}: {1}".format(no + 1, e))
106
107             if write and len(self.writing_queue) > 50000:
108                 self.write()
109
110             if write:
111                 self.write()
112
113     @abstractmethod
114     def write(self):
115         """ abstract method: should write data from writing_queue """
116         pass
117
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

```

A.8 vdASCIIFile.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.27
7  """
8
9  from vdFile import VdFile
10 from vdPoint import VdPoint
11 from abc import abstractmethod, ABC
12
13 class VdASCIIFile(VdFile):
14     """ abstract class for writing ascii files """
15
16     def _open_ascii(self, filename, file_format):
17         """
18             opens ascii file for writing
19             :param filename: file name
20             :type filename: str
21         """
22         filename = self._make_filename(file_format, filename)
23         self.__file = open(filename, 'a')
24
25     def _write2file(self, data):
26         """
27             writes ascii data to file
28             :param data: data to write
29             :type data: str

```

```

30         """
31         self._file.write(data)
32
33     def write(self):
34         """ writes data to file """
35         txt = ""
36         for d in self.writing_queue:
37             txt += self._format(d)
38         self._write2file(txt)
39         self.clear_writing_queue()
40
41     @abstractmethod
42     def _format(self, p):
43         """
44             abstract method: should convert point data to string for ascii file
45             :param p: Point
46             :type p: VdPoint
47             :return: point data as string
48             :rtype: str
49         """
50         pass
51
52     def close(self):
53         """ close file """
54         self._file.close()

```

A.9 vdTxtFile.py

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.27
7  """
8
9  from vdASCIIFile import VdASCIIFile
10 from vdPoint import VdPoint
11
12 class VdTxtFile(VdASCIIFile):
13
14     """ creates and fills an txt-file """
15
16     def _open(self, filename=""):
17         """
18             opens a txt file for writing
19             :param filename: name and path to new file
20             :type filename: str
21         """
22         VdASCIIFile._open_ascii(self, filename, "txt")
23
24     def _format(self, p):
25         """
26             Formats point for TXT
27             :param p: VdPoint
28             :type p: VdPoint
29             :return: txt point string
30             :rtype: str

```

```

31      """
32      format_string = '{:012.1f}\t{:07.3f}\t{: 03.0f}\t{:06.3f}\t{:03.0f}\n'
33      return format_string.format(p.time,
34                                  p.azimuth,
35                                  p.vertical,
36                                  p.distance,
37                                  p.reflection)

```

A.10 vdObjFile.py

```

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

4  """
5  @author: Florian Timm
6  @version: 2017.11.27
7  """

9  from vdASCIIFile import VdASCIIFile
10 from vdPoint import VdPoint

12 class VdObjFile(VdASCIIFile):

14     """ creates and fills an obj-file """
15
16     def _open(self, filename=""):
17         """
18             opens a txt file for writing
19             :param filename: name and path to new file
20             :type filename: str
21         """
22         VdASCIIFile._open_ascii(self, filename, "obj")

24     def _format(self, p):
25         """
26             Formats point for OBJ
27             :param p: VdPoint
28             :type p: VdPoint
29             :return: obj point string
30             :rtype: str
31         """
32         x, y, z = p.get_xyz()
33         format_string = 'v {:.3f} {:.3f} {:.3f}\n'
34         return format_string.format(x, y, z)

```

A.11 vdXYZFile.py

```

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

4  """
5  @author: Florian Timm
6  @version: 2017.11.27
7  """

9  from vdASCIIFile import VdASCIIFile
10 from vdPoint import VdPoint

```

```
12     class VdXYZFile(VdASCIIFile):
13
14         """ creates and fills an xyz-file """
15
16         def _open(self, filename=""):
17             """
18                 opens a txt file for writing
19                 :param filename: name and path to new file
20                 :type filename: str
21             """
22             VdASCIIFile._open_ascii(self, filename, "xyz")
23
24         def _format(self, p):
25             """
26                 Formats point for OBJ
27                 :param p: VdPoint
28                 :type p: VdPoint
29                 :return: obj point string
30                 :rtype: str
31             """
32             x, y, z = p.get_xyz()
33             format_string = '{:.3f} {:.3f} {:.3f}\n'
34             return format_string.format(x, y, z)
```

A.12 vdSQLite.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.27
7  """
8
9  import sqlite3
10 from vdFile import VdFile
11 from vdPoint import VdPoint
12
13
14 class VdSQLite(VdFile):
15     """ class for writing data to sqlite database """
16
17     def _open(self, filename):
18         """
19             opens a new db file
20             :param filename: name of db file
21             :type filename: str
22         """
23
24         filename = self._make_filename("db", filename)
25         print(filename)
26         self.__db = sqlite3.connect(filename)
27         self.__cursor = self.__db.cursor()
28         self.__cursor.execute("CREATE TABLE IF NOT EXISTS raw_data ("
29                             "id INTEGER PRIMARY KEY AUTOINCREMENT,"
30                             "time FLOAT,"
31                             "azimuth FLOAT,"
32                             "vertical FLOAT,"
```

```

32             "distance FLOAT ,"
33                     "reflection INTEGER )")
34         self.__db.commit()

36     def write(self):
37         """ writes data to database """
38         insert = []
39         for p in self.writing_queue:
40             insert.append((p.time, p.azimuth, p.vertical,
41                           p.distance, p.reflection))

43         self.__cursor.executemany("INSERT INTO raw_data ("
44                               "time, azimuth, vertical, "
45                               "distance, reflection) "
46                               "VALUES (?, ?, ?, ?, ?)", insert)
47         self.__db.commit()
48         self.clear_writing_queue()

50     def close(self):
51         """ close database """
52         self.__db.close()

```

A.13 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 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         """
43
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         return azi
50
51     def get_time(self):
52         """
53             gets timestamp of dataset
54             :return: timestamp of dataset
55             :rtype: int
56         """
57
58         time = ord(self.__dataset[1200:1201]) + \
59             (ord(self.__dataset[1201:1202]) << 8) + \
60             (ord(self.__dataset[1202:1203]) << 16) + \
61             (ord(self.__dataset[1203:1204]) << 24)
62         # print(time)
63         return time
64
65     def is_dual_return(self):
66         """
67             checks whether dual return is activated
68             :return: dual return active?
69             :rtype: bool
70         """
71
72         mode = ord(self.__dataset[1204:1205])
73         if mode == 57:
74             return True
75         return False
76
77     def get_azimuths(self):
78         """
79             get all azimuths and rotation angles from dataset
80             :return: azimuths and rotation angles
81             :rtype: list, list
82         """
83
84         # create empty lists
85         azimuths = [0.] * 24
86         rotation = [0.] * 12
87
88         # read existing azimuth values
89         for j in range(0, 24, 2):
90             a = self.get_azimuth(j // 2)
91             azimuths[j] = a
92
93         #: rotation angle

```

```

94         d = 0

96         # DualReturn active?
97         if self.is_dual_return():
98             for j in range(0, 19, 4):
99                 d2 = azimuths[j + 4] - azimuths[j]
100                if d2 < 0:
101                    d2 += 360.0
102                d = d2 / 2.0
103                a = azimuths[j] + d
104                azimuths[j + 1] = a
105                azimuths[j + 3] = a
106                rotation[j // 2] = d
107                rotation[j // 2 + 1] = d

109            rotation[10] = d
110            azimuths[21] = azimuths[20] + d

112        # Strongest / Last-Return
113    else:
114        for j in range(0, 22, 2):
115            d2 = azimuths[j + 2] - azimuths[j]
116            if d2 < 0:
117                d2 += 360.0
118            d = d2 / 2.0
119            a = azimuths[j] + d
120            azimuths[j + 1] = a
121            rotation[j // 2] = d

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

127        # >360 -> -360
128        for j in range(24):
129            if azimuths[j] > 360.0:
130                azimuths[j] -= 360.0

132        # print (azimuths)
133        # print (rotation)
134        return azimuths, rotation

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

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

```

```

153         times[i] = time
154         time += t_2repeat

156     t_between_laser = float(self._conf.get("device", "tInterBeams"))
157     t_recharge = float(self._conf.get("device", "tRecharge"))
158     part_rotation = float(self._conf.get("device", "ratioRotation"))

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

173             reflection = ord(self._dataset[6 + offset:7 + offset])

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

179             # create point
180             p = VdPoint(
181                 self._conf, round(
182                     time, 1), a, self._vertical_angle[k],
183                     dist, reflection)
184             self._data.append(p)

186             time += t_between_laser

188             # offset for next loop
189             offset += 3
190             time += t_recharge - t_between_laser

192     def get_data(self):
193         """
194             get all point data
195             :return: list of VdPoints
196             :rtype: list
197         """
198         return self._data

```

A.14 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):
13
14      """ Represents a point """
15
16      __dRho = math.pi / 180.0
17
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
35          self.__time = time
36          self.__azimuth = azimuth
37          self.__vertical = vertical
38          self.__reflection = reflection
39          self.__distance = distance
40          self.__conf = conf
41
42      @staticmethod
43      def parse_string(conf, line):
44          """
45              Parses string to VdPoint
46              :param conf: config file
47              :type conf: configparser.ConfigParser
48              :param line: Point as TXT
49              :type line: str
50              :return: Point
51              :rtype: VdPoint
52              :raise ValueError: malformed string
53          """
54
55          d = line.split()
56          if len(d) > 4:
57              time = float(d[0])
58              azimuth = float(d[1])
59              vertical = float(d[2])
60              distance = float(d[3])
61              reflection = int(d[4])
62              return VdPoint(conf, time, azimuth,
63                             vertical, distance, reflection)
64          else:
65              raise ValueError('Malformed string')
66
67      def __deg2rad(self, degree):
68          """
69              converts degree to radians
70              :param degree: degrees

```

```

69         :type degree: float
70         :return: radians
71         :rtype: float
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
91
92         # horizontal distance
93         s = d * math.cos(v) + beam_center
94
95         x = s * math.sin(a)
96         y = s * math.cos(a)
97         z = d * math.sin(v)
98
99         return x, y, z
100
101    def __get_time(self):
102        """
103        Gets recording time
104        :return: recording time in microseconds
105        :rtype: float
106        """
107        return self._time
108
109    def __get_azimuth(self):
110        """
111        Gets azimuth direction
112        :return: azimuth direction in degrees
113        :rtype: float
114        """
115        return self._azimuth
116
117    def __get_azimuth_radians(self):
118        """
119        Gets azimuth in radians
120        :return: azimuth direction in radians
121        :rtype: float
122        """
123        return self._deg2rad(self.azimuth)
124
125    def __get_vertical(self):
126        """
127        Gets vertical angle in degrees

```

```

128         :return: vertical angle in degrees
129         :rtype: float
130         """
131         return self._vertical
132
133     def __get_vertical_radians(self):
134         """
135             Gets vertical angle in radians
136             :return: vertical angle in radians
137             :rtype: float
138             """
139         return self._deg2rad(self.vertical)
140
141     def __get_reflection(self):
142         """
143             Gets reflection
144             :return: reflection between 0 and 255
145             :rtype: int
146             """
147         return self._reflection
148
149     def __get_distance(self):
150         """
151             Gets distance
152             :return: distance in metres
153             :rtype: float
154             """
155         return self._distance
156
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.15 config.ini

```

1 [network]
2 UDP_IP = 0.0.0.0
3 UDP_PORT_DATA = 2368
4 UDP_PORT_GNSS = 8308
5
6 [serial]
7 # Serieller Port
8 #Raspberry
9 GNSSport = /dev/ttyAMA0
10 #Ubuntu
11 #GNSSport = /dev/ttyUSB0
12
13 [functions]
14 # Zeitgleiche Transformation zu txt aktivieren
15 activateTransformer = True
16
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 timeFormat = '%Y-%m-%dT%H:%M:%S'

32 # Dateipraefix der zu speichernden Datei
33 namePre = data

35 # zu verwendender Transformer
36 transformer = python
37 # transformer = linux64
38 # transformer = arm32

41 [device]
42 # Zeit zwischen den Messungen der Einzelstrahlen
43 tInterBeams = 2.304

45 # Zeit zwischen zwei Aussendungen des gleichen Messlasers
46 tRepeat = 55.296

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

51 # Anteil der Zeit zwischen Einzellasern an Wiederholungszeit,
52 # fuer Interpolation des Horizontalwinkels
53 ratioRotation = 0.041666666666666664
54 #tZwischenStrahl / tRepeat

56 # Zeit nach letztem Strahl bis zum naechsten
57 tRecharge = 20.736
58 #tRepeat - 15 * tZwischenStrahl

60 # Abstand des Strahlenzentrums von der Drehachse
61 beamCenter = 0.04191

63 valuesPerDataset = 384
64 #12*32

66 # Bytes pro Messdatenblock
67 offsetBlock = 100
68 # 3 * 32 + 4

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

```

A.16 convTxt2Obj.py

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

```

2  # -*- coding: utf-8 -*-
3
4  """
5  @author: Florian Timm
6  @version: 2017.11.20
7  """
8  import configparser
9
10 from vdObjFile import VdObjFile
11
12 fileName = "BeispielDateien/test.txt"
13
14 conf = configparser.ConfigParser()
15 conf.read("config.ini")
16
17 f = VdObjFile(conf, fileName)
18 f.read_from_txt_file(fileName, True)

```

A.17 convBin2Obj.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 configparser
10 import os
11 from glob import glob
12 from time import *
13
14 from vdDataset import VdDataset
15
16 from vdObjFile import VdObjFile
17
18 # load config file
19 conf = configparser.ConfigParser()
20 conf.read("config.ini")
21
22 #fs = glob(
23 #    "../BeispielDateien/*.bin")
24
25 fs = ["/ssd/daten/ThesisMessung2/data2017-11-27T13:51:16/4.bin"]
26
27 t1 = clock()
28 if len(fs) > 0:
29     folder = os.path.dirname(fs[0])
30     obj = VdObjFile(
31         conf,
32         folder + "/file")
33
34     for filename in fs:
35         print(filename)
36
37         bin_file = open(filename, "rb")

```

```
39      # Calculate number of datasets
40      fileSize = os.path.getsize(bin_file.name)
41      print ("FileSize: "+str(fileSize))
42      cntDatasets = fileSize // 1206
43      print ("Datasets: "+str(cntDatasets))

45      for i in range(cntDatasets):
46          vdData = VdDataset(conf, bin_file.read(1206))
47          vdData.convert_data()
48          obj.write_data(vdData.get_data())
49          bin_file.close()
50      obj.close()

52  t2 = clock()

54  t = t2 - t1
55  print ("Laufzeit: " + str(t))
```

B C++-Quelltexte

B.1 main.cpp

```
1  /*!
2   * \brief      main method
3   *
4   * \author    Florian Timm
5   * \version   2017.11.29
6   * \copyright MIT License
7   */
8
9  #include <iostream>
10 #include <stdio.h>
11 #include <list>
12 #include <iterator>
13 #include <string>
14 using namespace std;
15 #include "VdXYZ.h"
16 #include "VdPoint.h"
17 #include "VdDataset.h"
18 #include "VdFile.h"
19 #include "VdXYZFile.h"
20 #include "VdTxtFile.h"
21 #include "VdObjFile.h"
22 #include "VdSQLite.h"
23 extern "C" {
24 #include "iniparser/iniparser.h"
25 }
26
27 dictionary * ini = NULL;
28
29 void loadini() {
30     ini = iniparser_load("config.ini");
31 }
32
33 void transformBin2x(std::string binFile, VdFile* newFile) {
34     /**
35      * transforms bin files to several
36      * file formats
37      * @param binFile bin file
38      * @param newFile file object to write
39      */
40
41     ifstream file(binFile.c_str(), ios::binary | ios::in);
42
43     if (file.is_open()) {
44         file.seekg(0, ios::end);
45         int length = file.tellg();
```

```

46     file.seekg(0, ios::beg);

48     cout << "FileSize: " << length << endl;
49     int cntDatasets = (int) (length / 1206);
50     cout << "DataSets: " << cntDatasets << endl;

52     for (int i = 0; i < cntDatasets; i++) {

54         //while (!file.eof()) {
55         char dataset[1206];
56         file.read(dataset, 1206);
57         VdDataset vd = VdDataset(ini, dataset);
58         vd.convertData();

59         std::list<VdPoint> p = vd.getData();
60         newFile->addDataset(&p);
61     }
62 }
63 newFile->write();
64 }
65 file.close();
66 }

68 void start(string old_format, string filename_old, string new_format,
69             string filename_new) {
70 /**
71 * transforms files
72 * @param old_format format of input file (bin, txt)
73 * @param filename_old name of input file
74 * @param new_format format of output file (txt, xyz, obj, sql)
75 * @param filename_new name of output file
76 */
77 if (old_format == "bin") {
78     cout << "Input: bin" << endl;
79     cout << "Output: ";
80     if (new_format == "txt") {
81         cout << "txt" << endl;
82         VdTxtFile txt(filename_new);
83         transformBin2x(filename_old, &txt);
84     } else if (new_format == "xyz") {
85         cout << "xyz" << endl;
86         VdXYZFile xyz = VdXYZFile(filename_new);
87         transformBin2x(filename_old, &xyz);
88     } else if (new_format == "obj") {
89         cout << "obj" << endl;
90         VdObjFile obj = VdObjFile(filename_new);
91         transformBin2x(filename_old, &obj);
92     } else if (new_format == "sql") {
93         cout << "sql" << endl;
94         VdSQLite sql = VdSQLite(filename_new);
95         transformBin2x(filename_old, &sql);
96     } else {
97         cout << "unknown file format" << endl;
98     }
99
100 } else if (old_format == "txt") {
101     cout << "not implemented!" << endl;
102 } else {
103     cout << "unknown file format" << endl;
104 }

```

```

105  }

107 int main(int argc, char* argv[]) {
108     /**
109      * main method, starts program
110      * @param argc number of arguments
111      * @param argv arguments
112      * @return 0
113      */
114

115     loadini();

116
117     if (argc == 3) {
118         string filename_old = argv[1];
119         string filename_new = argv[2];
120         string old_end = filename_old.substr(filename_old.length() - 3);
121         string new_end = filename_new.substr(filename_new.length() - 3);
122         //transformBin2Obj(argv[1], argv[2]);
123         start(old_end, filename_old, new_end, filename_new);
124
125     } else if (argc == 5) {
126         start(argv[1], argv[2], argv[3], argv[4]);
127     } else {
128         cout << "No parameters!" << endl;
129         cout << "usage: veloTrans [bin|txt] old_file [txt|xyz|obj|sql] new_file"
130             << endl;
131     }
132     return (0);
133 }

```

B.2 VdFile.h

```

1  /*!
2   * \brief      class for writing data
3   *
4   * \author    Florian Timm
5   * \version   2017.11.29
6   * \copyright MIT License
7   */

9 #ifndef VDFILE_H_
10 #define VDFILE_H_

12 #include <list>
13 #include <string>
14 #include "VdFile.h"
15 #include "VdPoint.h"

17 class VdFile {
18 public:
19     std::list<VdPoint>& getWritingQueue();
20     void clearWritingQueue();
21     std::string makeFilename (std::string file_format, std::string file_name=std::
22                             string(""));
22     void addPoint(VdPoint point);
23     void addDataset(std::list<VdPoint>* dataset);
24     void writeData(std::list<VdPoint>* data);
25     virtual void write() = 0;

```

```

27 protected:
28     virtual void open(std::string filename = "") = 0;
29     virtual void close() = 0;
30     std::list<VdPoint> writingQueue;
31 };

34 #endif /* VDFILE_H_ */

```

B.3 VdFile.cpp

```

1  /*
2   * \brief      class for writing data
3   *
4   * \author    Florian Timm
5   * \version   2017.11.29
6   * \copyright MIT License
7   */
8 #include <string>
9 #include <iostream>
10 #include <list>
11 using namespace std;
12 #include "VdFile.h"

14 list<VdPoint>& VdFile::getWritingQueue() {
15     /**
16      * Returns points in queue
17      * @return points in queue
18     */
19     return this->writingQueue;
20 }

22 void VdFile::clearWritingQueue() {
23     /** clears writing queue */
24     this->writingQueue.clear();
25 }

27 inline bool ends_with(string const & value, string const & ending) {
28     if (ending.size() > value.size())
29         return (false);
30     return (equal(ending.rbegin(), ending.rend(), value.rbegin()));
31 }

33 string VdFile::makeFilename(string file_format, string file_name) {
34     /**
35      * generates a new file_name from timestamp
36      * @param file_format file suffix
37      * @param file_name file name
38      * @return string with date and suffix
39     */
40     string fileEnding = string(".") + file_format;
41     if (file_name.empty()) {
42         return (string("noname") + fileEnding);
43     } else if (ends_with(file_name, fileEnding)) {
44         return (file_name);
45     }
46     return (file_name + fileEnding);

```

```

47  }

49 void VdFile::writeData(list<VdPoint>* dataset) {
50     /**
51      * adds data and writes it to file
52      * @param data ascii data to write
53      */
54     this->addDataset(dataset);
55     this->write();
56 }

58 void VdFile::addPoint(VdPoint point) {
59     /**
60      * Adds a point to write queue
61      * @param p point
62      */
63     this->writingQueue.push_back(point);
64 }

66 void VdFile::addDataset(list<VdPoint>* dataset) {
67     /**
68      * adds multiple points to write queue
69      * @param dataset multiple points
70      */
71     this->writingQueue.insert(this->writingQueue.begin(), dataset->begin(), dataset->
72         end());

```

B.4 VdASCIIFile.h

```

1  /*!
2   * \brief      class for writing data to ascii file
3   *
4   * \author    Florian Timm
5   * \version   2017.11.29
6   * \copyright MIT License
7   */

9 #ifndef VDASCIIFILE_H_
10 #define VDASCIIFILE_H_

12 #include "VdFile.h"
13 #include <fstream>

15 class VdASCIIFile : public VdFile {
16 public:
17     void write();
18 protected:
19     void openASCII(std::string filename, std::string file_format);
20     void write2file(std::string data);
21     void close();
22     virtual std::string format(VdPoint point) = 0;

24     FILE* file;
25 };

27 #endif /* VDASCIIFILE_H_ */

```

B.5 VdASCIIFile.cpp

```
1  /*!
2   * \brief      class for writing data to ascii file
3   *
4   * \author     Florian Timm
5   * \version    2017.11.29
6   * \copyright  MIT License
7   */
8
9  #include <string>
10 #include <stdio.h>
11 #include <vector>
12 #include <fstream>
13 #include <iostream>
14 using namespace std;
15 #include "VdPoint.h"
16 #include "VdASCIIFile.h"
17 #include "iniparser/iniparser.h"
18
19 void VdASCIIFile::openASCII(string filename, string file_format) {
20     /**
21      * opens ascii file for writing
22      * @param filename file name
23      */
24     filename = this->makeFilename(file_format, filename);
25     this->file = fopen(filename.c_str(), "at");
26 }
27
28 void VdASCIIFile::write2file(string data) {
29     /**
30      * writes ascii data to file
31      * @param data data to write
32      */
33     if (this->file!=NULL) {
34         fputs (data.c_str(),this->file);
35     }
36 }
37
38 void VdASCIIFile::write() {
39     /** writes data to file */
40     string txt;
41     for (VdPoint p : this->writingQueue)
42         txt += this->format(p);
43     this->write2file(txt);
44     this->clearWritingQueue();
45 }
46
47 void VdASCIIFile::close() {
48     /** close file */
49     fclose(this->file);
50 }
```

B.6 VdObjFile.h

```
1  /*!
2   * \brief      represents a file for writing data
3   *
```

```

4   * \author Florian Timm
5   * \version 2017.11.30
6   * \copyright MIT License
7   */

9 #ifndef VDOBJFILE_H_
10 #define VDOBJFILE_H_

12 #include "VdASCIIFile.h"

14 class VdObjFile: public VdASCIIFile {
15 public:
16     VdObjFile(std::string filename = "");

18 protected:
19     void open(std::string filename = "");
20     string format(VdPoint point);
21 };

23 #endif /* VDOBJFILE_H_ */

```

B.7 VdObjFile.cpp

```

1  /*
2   * \brief represents a file for writing data
3   *
4   * \author Florian Timm
5   * \version 2017.11.30
6   * \copyright MIT License
7   */

9 #include <string>
10 #include <vector>
11 #include <fstream>
12 using namespace std;
13 #include "VdXYZ.h"
14 #include "VdObjFile.h"

16 VdObjFile::VdObjFile(string filename) {
17     // TODO Auto-generated constructor stub
18     this->open(filename);
19 }

21 void VdObjFile::open(std::string filename) {
22     /**
23      * opens a txt file for writing
24      * @param filename name and path to new file
25      */
26     this->openASCII(filename, string("obj"));
27 }

29 string VdObjFile::format(VdPoint point) {
30     /**
31      * Formats point for OBJ
32      * @param point VdPoint
33      * @return obj point string
34      */
35     VdXYZ xyz = point.getXYZ();

```

```

36     char result[50];
37     sprintf (result, "v %.3f %.3f %.3f\n", xyz.getX(), xyz.getY(), xyz.getZ());
38     return (string(result));
39 }

```

B.8 VdTxtFile.h

```

1  /*!
2   * \brief      class for writing raw data to txt file
3   *
4   * \author    Florian Timm
5   * \version   2017.11.29
6   * \copyright MIT License
7   */
8
9 #ifndef VDTXTFILE_H_
10 #define VDTXTFILE_H_
11
12 #include <string>
13 #include "VdASCIIFile.h"
14 #include "VdPoint.h"
15 using namespace std;
16
17 class VdTxtFile: public VdASCIIFile {
18 public:
19     VdTxtFile(string filename = "");
20
21 protected:
22     void open (string filename = "");
23     string format(VdPoint);
24 };
25
26 #endif /* VDTXTFILE_H_ */

```

B.9 VdTxtFile.cpp

```

1  /*
2   * \brief      class for writing raw data to txt file
3   *
4   * \author    Florian Timm
5   * \version   2017.11.29
6   * \copyright MIT License
7   */
8
9 #include "VdTxtFile.h"
10 #include <string>
11 using namespace std;
12
13 VdTxtFile::VdTxtFile(string filename) {
14     /**
15      * constructor
16      * @param filename: name and path to new file
17      */
18     this->open(filename);
19 }
20
21 void VdTxtFile::open(std::string filename) {

```

```

22     /**
23      * opens a txt file for writing
24      * @param filename: name and path to new file
25      */
26     this->openASCII(filename, string("txt"));
27 }

29 string VdTxtFile::format(VdPoint point) {
30     /**
31      * Formats point for Txt
32      * @param point: VdPoint
33      * @return txt point string
34      */
35     VdXYZ xyz = point.getXYZ();
36     char result[40];
37     sprintf(result, "%012.1f\t%07.3f\t%03d\t%06.3f\t%03d\n",
38             point.getAzimuth(), point.getVertical(), point.getDistance(),
39             point.getReflection());
40     return (string(result));
41 }

```

B.10 VdXYZFile.h

```

1  /* !
2   * \brief      class for writing data to xyz file
3   *
4   * \author    Florian Timm
5   * \version   2017.11.30
6   * \copyright MIT License
7   */

9 #ifndef VDXYZFILE_H_
10 #define VDXYZFILE_H_

12 #include <string>
13 #include "VdASCIIFile.h"
14 using namespace std;

16 class VdXYZFile: public VdASCIIFile {
17 public:
18     VdXYZFile(string filename = "");

20 protected:
21     void open(string filename);
22     string format(VdPoint point);
23 };

25 #endif /* VDXYZFILE_H_ */

```

B.11 VdXYZFile.cpp

```

1  /* !
2   * \brief      class for writing data to xyz file
3   *
4   * \author    Florian Timm
5   * \version   2017.11.30
6   * \copyright MIT License

```

```

7     */

9 #include "VdXYZFile.h"
10 #include <string>
11 #include <iostream>
12 using namespace std;

14 VdXYZFile::VdXYZFile(string filename) {
15     /**
16      * konstruktor
17      * @param filename name for new file
18      */
19     this->open(filename);
20     this->writingQueue = list<VdPoint>();
21 }

23 void VdXYZFile::open(string filename = "") {
24     /**
25      * opens a txt file for writing
26      * @param filename name and path to new file
27      */
28     openASCII(filename, "xyz");
29 }

31 string VdXYZFile::format(VdPoint point) {
32     /**
33      * Formats point for OBJ
34      * @param point VdPoint
35      * @return obj point string
36      */
37     VdXYZ xyz = point.getXYZ();
38     // '{:.3f} {:.3f} {:.3f}\n'
39     char result[50];
40     sprintf(result, "%.3f %.3f %.3f\n", xyz.getX(), xyz.getY(), xyz.getZ());
41     //cout << result;
42     return (string(result));
43 }

```

B.12 VdSQLLite.h

```

1 /*!
2  * \brief      class for writing data to sqlite database
3  *
4  * \author    Florian Timm
5  * \version   2017.11.29
6  * \copyright MIT License
7  */

9 #ifndef VDSQLITE_H_
10 #define VDSQLITE_H_

12 #include "VdFile.h"
13 #include <sqlite3.h>
14 #include <string>
15 using namespace std;

17 class VdSQLLite : public VdFile{
18 public:

```

```

19     VdSQLite(string filename = "");
20     void write();
21 protected:
22     void open(string filename = "");
23     void close();
24 private:
25     sqlite3 *db;
26     sqlite3_stmt *stmt;
27 };
28 #endif /* VDSQLITE_H_ */

```

B.13 VdSQLite.cpp

```

1  /*!
2   * \brief      class for writing data to sqlite database
3   *
4   * \author    Florian Timm
5   * \version   2017.11.29
6   * \copyright MIT License
7   */
8
9 #include "VdSQLite.h"
10 #include <stdio.h>
11 #include <iostream>
12 #include <sqlite3.h>
13 using namespace std;
14
15 VdSQLite::VdSQLite(string filename) {
16     /**
17      * constructor
18      * @param filename name of db file
19      */
20     this->open(filename);
21     db;
22 }
23
24 void VdSQLite::open(string filename) {
25     /**
26      * opens a new db file
27      * @param filename name of db file
28      */
29     char *zErrMsg = 0;
30     filename = this->makeFilename(string("db"), filename);
31     sqlite3_open(filename.c_str(), &this->db);
32
33     string sql =
34         "CREATE TABLE IF NOT EXISTS raw_data (id INTEGER PRIMARY KEY AUTOINCREMENT,
35             time FLOAT, azimuth FLOAT, vertical FLOAT, distance FLOAT, reflection
36             INTEGER)";
37     sqlite3_exec(this->db, sql.c_str(), 0, 0, &zErrMsg);
38     sqlite3_prepare_v2(db,
39         "INSERT INTO raw_data (time, azimuth, vertical, distance, reflection) VALUES
40             (?, ?, ?, ?, ?)",
41         -1, &stmt, 0);
42 }
43
44 void VdSQLite::write() {

```

```

42     /** writes data to database */
43     char *zErrMsg = 0;
44     sqlite3_exec(this->db, "BEGIN TRANSACTION;", 0, 0, &zErrMsg);
45     for (VdPoint p : this->writingQueue) {
46         sqlite3_bind_double(stmt, 1, p.getTime());
47         sqlite3_bind_double(stmt, 2, p.getAzimuth());
48         sqlite3_bind_double(stmt, 3, p.getVertical());
49         sqlite3_bind_double(stmt, 4, p.getDistance());
50         sqlite3_bind_int(stmt, 5, p.getReflection());
51         sqlite3_step(stmt);
52         sqlite3_reset(stmt);
53     }
54     sqlite3_exec(this->db, "END TRANSACTION;", 0, 0, &zErrMsg);
55     this->clearWritingQueue();
56 }
57 void VdSQLite::close() {
58     /* close database */
59     sqlite3_close(db);
60 }
```

B.14 VdDataset.h

```

1  /*!
2   * \brief      represents a dataset
3   *
4   * \author    Florian Timm
5   * \version   2017.11.30
6   * \copyright MIT License
7   */
8
9 #ifndef VDDATASET_H_
10 #define VDDATASET_H_
11
12 #include "VdPoint.h"
13 #include <list>
14 #include "iniparser/iniparser.h"
15
16 class VdDataset {
17 public:
18     VdDataset(dictionary * conf, char * dataset);
19     void convertData();
20     const std::list<VdPoint>& getData() const;
21
22 private:
23     std::list<VdPoint> data;
24     double getAzimuth(int);
25     double getTime();
26     bool isDualReturn();
27     void getAzimuths(double[], double[]);
28
29     char * dataset;
30     dictionary * conf;
31     int verticalAngle[16] = { -15, 1, -13, -3, -11, 5, -9, 7, -7,
32                             9, -5, 11, -3, 13, -1, 15 };
33     double tRepeat;
34     int offsets[12] = { 0, 100, 200, 300, 400, 500, 600, 700, 800,
35                        900, 1000, 1100 };
```

```

37  };
38 #endif /* VDDATASET_H_ */

```

B.15 VdDataset.cpp

```

1  /*!
2   * \brief      represents a dataset
3   *
4   * \author    Florian Timm
5   * \version   2017.11.30
6   * \copyright MIT License
7   */
8
9  #include <iostream>
10 #include <fstream>
11 #include <list>
12 #include <iterator>
13 using namespace std;
14 #include "VdXYZ.h"
15 #include "VdPoint.h"
16 #include "VdDataset.h"
17 #include "iniparser/iniparser.h"
18
19
20 VdDataset::VdDataset(dictionary * conf, char * dataset) {
21     this->dataset = dataset;
22     this->conf = conf;
23     list<VdPoint> data;
24     tRepeat = iniparser_getdouble(conf, "device:trepeat", 55.296);
25 }
26
27 double VdDataset::getAzimuth(int block) {
28     /**
29      * gets azimuth of a data block
30      * @param block number of data block
31      * @return azimuth
32     */
33
34     int offset = this->offsets[block];
35     // change byte order
36     double azi = 0;
37     azi = (unsigned char) dataset[offset + 2];
38     azi += ((unsigned char) dataset[offset + 3]) << 8;
39     azi /= 100.0;
40     return azi;
41 }
42
43 double VdDataset::getTime() {
44     /**
45      * gets timestamp of dataset
46      * @return timestamp of dataset
47     */
48
49     double time = ((unsigned char) dataset[1200])
50         + (((unsigned char) dataset[1201]) << 8)
51         + (((unsigned char) dataset[1202]) << 16)
52         + (((unsigned char) dataset[1203]) << 24);
53     return time;
54 }

```

```

54 }

56 bool VdDataset::isDualReturn() {
57 /**
58 * checks whether dual return is activated
59 * @return dual return active?
60 */

62 int mode = (unsigned char) dataset[1204];
63 //cout << mode << endl;
64 if (mode == 57) {
65 //cout << "dr" << endl;
66 return true;
67 }
68 return false;
69 }

71 const std::list<VdPoint>& VdDataset::getData() const {
72 /**
73 * returns all point data
74 * @return point data
75 */
76 return data;
77 }

79 void VdDataset::getAzimuths(double azimuths[], double rotation[]) {
80 /**
81 * get all azimuths and rotation angles from dataset
82 * @return azimuths and rotation angles
83 */
84

85 // read existing azimuth values
86 for (int j = 0; j < 24; j += 2) {
87     double a = this->getAzimuth((int) j / 2);
88     azimuths[j] = a;
89 }

90 // rotation angle
91 double d = 0;
92 // DualReturn active?
93 if (this->isDualReturn()) {
94     for (int j = 0; j < 19; j += 4) {
95         double d2 = azimuths[j + 4] - azimuths[j];
96         if (d2 < 0) {
97             d2 += 360.0;
98         }
99         d = d2 / 2.0;
100        double a = azimuths[j] + d;
101        azimuths[j + 1] = a;
102        azimuths[j + 3] = a;
103        rotation[(int) j / 2] = d;
104        rotation[(int) j / 2 + 1] = d;
105    }
106 }
107 rotation[10] = d;
108 azimuths[21] = azimuths[20] + d;
109 }
110 // Strongest / Last-Return
111 else {
112     for (int j = 0; j < 22; j += 2) {

```

```

113     double d2 = azimuths[j + 2] - azimuths[j];
114     if (d2 < 0) {
115         d2 += 360.0;
116     }
117     d = d2 / 2.0;
118     double a = azimuths[j] + d;
119     azimuths[j + 1] = a;
120     rotation[(int) j / 2] = d;
121 }
122 }
123 // last rotation angle from angle before
124 rotation[11] = d;
125 azimuths[23] = azimuths[22] + d;

127 // >360 -> -360
128 for (int j = 0; j < 24; j++) {
129     while (azimuths[j] > 360.0) {
130         azimuths[j] -= 360.0;
131     }
132 }
134 }

136 void VdDataset::convertData() {
137     /** converts binary data to objects */
138     double t_between_laser = iniparser_getdouble(conf, "device:tinterbeams", 2.304);
139     double t_recharge = iniparser_getdouble(conf, "device:trecharge", 20.736);
140     double part_rotation = iniparser_getdouble(conf, "device:ratiорotation",
141         0.0416666666666664);

142     // create empty lists
143     double azimuth[24];
144     double rotation[12];
145     this->getAzimuths(azimuth, rotation);

147     bool dual_return = this->isDualReturn();

149 // timestamp from dataset
150     double time = this->getTime();
151     double times[12] = { };
152     double t_2repeat = 2 * tRepeat;
153     if (dual_return) {
154         for (int i = 0; i < 12; i += 2) {
155             times[i] = time;
156             times[i + 1] = time;
157             time += t_2repeat;
158         }
159     } else {
160         for (int i = 0; i < 12; i++) {
161             times[i] = time;
162             time += t_2repeat;
163         }
164     }
165     double azi_block, a;

167 // data package has 12 blocks with 32 measurements
168 for (int i = 0; i < 12; i++) {
169     int offset = this->offsets[i];
170     time = times[i];

```

```

171     for (int j = 0; j < 2; j++) {
172         azi_block = azimuth[(int) i * 2 + j];
173         for (int k = 0; k < 16; k++) {
174             // get distance
175             double dist = ((unsigned char) this->dataset[4 + offset])
176                     + (((unsigned char) this->dataset[5 + offset]) << 8);
177             if (dist > 0.0) {
178                 dist /= 500.0;
179
180                 int reflection = (unsigned char) this->dataset[offset + 6];
181
182                 // interpolate azimuth
183                 a = azi_block + rotation[i] * k * part_rotation;
184
185                 // create point
186                 VdPoint p(time, a, verticalAngle[k], dist, reflection);
187
188                 data.push_back(p);
189             }
190             time += t_between_laser;
191
192             // offset for next loop
193             offset += 3;
194         }
195         time += t_recharge - t_between_laser;
196     }
197 }
198 }
```

B.16 VdPoint.h

```

1  /*!
2  * \brief      represents a point
3  *
4  * \author    Florian Timm
5  * \version   2017.11.30
6  * \copyright MIT License
7  */
8
9 #ifndef VDPOINT_H_
10 #define VDPOINT_H_
11 #include "VdXYZ.h"
12
13 class VdPoint {
14 public:
15     VdPoint();
16     VdPoint(double, double, int, double, int);
17     virtual ~VdPoint();
18     static double deg2rad(double degree);
19     VdXYZ getXYZ();
20     double getAzimuth();
21     void setAzimuth(double azimuth);
22     double getDistance();
23     void setDistance(double distance);
24     int getReflection();
25     void setReflection(int reflection);
26     double getTime();
27     void setTime(double time);
```

```

28     int getVertical();
29     void setVertical(int vertical);
30     static double dRho;
31
32     private:
33     double time_;
34     double azimuth_;
35     int vertical_;
36     double distance_;
37     int reflection_;
38 };
39
40 #endif /* VDPOINT_H_ */

```

B.17 VdPoint.cpp

```

1  /*!
2   * \brief      represents a point
3   *
4   * \author    Florian Timm
5   * \version   2017.11.30
6   * \copyright MIT License
7   */
8
9  #include <math.h>
10 #include <vector>
11 #include <iostream>
12 #include "VdXYZ.h"
13 #include "VdPoint.h"
14 #define PI 3.141592653589793238463
15
16 VdPoint::VdPoint() {
17     time_ = 0;
18     azimuth_ = 0;
19     vertical_ = 0;
20     distance_ = 0;
21     reflection_ = 0;
22 }
23
24 VdPoint::VdPoint(double time, double azimuth, int verticalAngle,
25                   double distance, int reflection) {
26     /**
27      * Constructor
28      * @param time recording time in microseconds
29      * @param azimuth Azimuth direction in degrees
30      * @param vertical Vertical angle in degrees
31      * @param distance distance in metres
32      * @param reflection reflection 0-255
33      */
34     time_ = time;
35     azimuth_ = azimuth;
36     vertical_ = verticalAngle;
37     distance_ = distance;
38     reflection_ = reflection;
39 }
40
41 double VdPoint::dRho = PI / 180.0;

```

```

43  VdPoint::~VdPoint() {
44      /** destructor stub */
45  }

47  double VdPoint::deg2rad(double degree) {
48      /**
49       * converts degree to radians
50       * @param degree degrees
51       * @return radians
52       */
53      return degree * dRho;
54  }

56  VdXYZ VdPoint::getXYZ() {
57      /**
58       * Gets local coordinates
59       * @return local coordinates x, y, z in metres
60       */
61      double beam_center = 0.04191;

63      // slope distance to beam center
64      double d = distance_ - beam_center;

66      // vertical angle in radians
67      double v = deg2rad(vertical_);

69      // azimuth in radians
70      double a = deg2rad(azimuth_);

72      // horizontal distance
73      double s = d * cos(v) + beam_center;

75      double x = s * sin(a);
76      double y = s * cos(a);
77      double z = d * sin(v);

79      return VdXYZ(x, y, z);
80  }

82  double VdPoint::getAzimuth() {
83      /**
84       * returns azimuth
85       * @return azimuth in degrees
86       */
87      return (azimuth_);
88  }

90  double VdPoint::getDistance() {
91      /**
92       * returns the distance
93       * @return distance
94       */
95      return (distance_);
96  }

98  int VdPoint::getReflection() {
99      /**
100       * returns the reflection
101       * @return reflection between 0 - 255

```

```

102     */
103     return (reflection_);
104 }

106 double VdPoint::getTime() {
107     /**
108      * returns time
109      * @return time in microseconds
110      */
111     return (time_);
112 }

114 int VdPoint::getVertical() {
115     /**
116      * returns the vertical angle
117      * @return vertical angle in degrees
118      */
119     return (vertical_);
120 }

```

B.18 VdXYZ.h

```

1  /*!
2  * \brief      represents a coordinate
3  *
4  * \author    Florian Timm
5  * \version   2017.11.30
6  * \copyright MIT License
7 */

9 #ifndef VDXYZ_H_
10 #define VDXYZ_H_

12 class VdXYZ {
13 public:
14     VdXYZ();
15     VdXYZ(double x, double y, double z);
16     virtual ~VdXYZ();
17     double getX();
18     void setX(double x);
19     double getY();
20     void setY(double y);
21     double getZ();
22     void setZ(double z);

24 private:
25     double x_;
26     double y_;
27     double z_;
28 };

30 #endif /* VDXYZ_H_ */

```

B.19 VdXYZ.cpp

```

1  /*!
2  * \brief      represents a coordinate

```

```

3   *
4   * \author Florian Timm
5   * \version 2017.11.30
6   * \copyright MIT License
7   */

9 #include "VdXYZ.h"

11 VdXYZ::VdXYZ() {
12     /** constructor */
13     this->x_ = -999;
14     this->y_ = -999;
15     this->z_ = -999;
16 }

18 VdXYZ::VdXYZ(double x, double y, double z) {
19     /**
20      * constructor
21      * @param x x coordinate
22      * @param y y coordinate
23      * @param z z coordinate
24      */
25     this->x_ = x;
26     this->y_ = y;
27     this->z_ = z;
28 }

30 VdXYZ::~VdXYZ() {
31     /** destructor stub */
32 }

34 double VdXYZ::getX() {
35     /**
36      * returns x coordinate
37      * @return x coordinate
38      */
39     return (x_);
40 }

42 void VdXYZ::setX(double x) {
43     /**
44      * sets x coordinate
45      * @param x x coordinate
46      */
47     this->x_ = x;
48 }

50 double VdXYZ::getY() {
51     /**
52      * returns y coordinate
53      * @return y coordinate
54      */
55     return (y_);
56 }

58 void VdXYZ::setY(double y) {
59     /**
60      * sets y coordinate
61      * @param y y coordinate

```

```
62      */
63      this->y_ = y;
64  }

66  double VdXYZ::getZ() {
67  /**
68   * returns z coordinate
69   * @return z coordinate
70   */
71  return (z_);
72 }

74  void VdXYZ::setZ(double z) {
75  /**
76   * sets z coordinate
77   * @param z z coordinate
78   */
79  this->z_ = z;
80 }
```

C Arduino-Quelltext

C.1 ppsVergleich.ino

```
1 #include <LiquidCrystal.h>
2 LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

4 const int kanal1 = 31;
5 const int kanal2 = 33;

7 int state = LOW;
8 int laststate = LOW;
9 unsigned long time1 = 0, time2 = 0;

11 void setup() {
12     // Display 2 Zeilen a 16 Zeichen
13     lcd.begin(16, 2);
14     // Schreibposition setzen
15     lcd.setCursor(0, 0);
16     // Schreibt Text
17     lcd.print("Zeitdifferenz:");
18     lcd.setCursor(0, 1);

20     pinMode(kanal1, INPUT);
21     pinMode(kanal2, INPUT);

23     Serial.begin(9600);
24 }

26 void loop() {
27     state = digitalRead(kanal1);
28     time1 = micros();
29     // erste Flanke
30     if (laststate == LOW && state == HIGH) {
31         laststate = LOW;
32         state = LOW;
33         // solange, bis 2. Flanke
34         do {
35             laststate = state;
36             state = digitalRead(kanal2);
37             time2 = micros();
38         } while (!(laststate == LOW && state == HIGH));

40         // Zeitdiff berechnen und ausgeben
41         unsigned long diff = time2 - time1;
42         String wert = String(diff);
43         int pos = 16 - wert.length();
44         if (pos <= 0) {
45             pos = 0;
```

```
46     wert = "zu lang";
47 }
48 lcd.setCursor(pos, 1);
49 Serial.println(diff);
50 lcd.print(diff);
51 laststate = LOW;
52 state = LOW;
53 delay(50000);
54 lcd.clear();
55 lcd.setCursor(0, 0);
56 // Schreibt Text
57 lcd.print("Zeitdifferenz:");
58 lcd.setCursor(0, 1);
59 }
60 laststate = state;
61 }
```

D Beispieldateien

D.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
```

D.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

```

D.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