Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fakultät Maschinenbau und Energietechnik
Bachelor-Studiengang Maschinenbau

Navigationsdatenverarbeitung in Python

Praktikumsarbeit
verfasst am Institut für Methodik der Fernerkundung des Deutschen Zentrums für Luft- und Raumfahrt

von

Arthur Schletter

geb. am 27.01.2001

in Görlitz

Matrikelnr. 78591

Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. Guido Reuther

Betreuer im Praktikumsbetrieb: Dr. Claas Köhler, Stefan Plattner

Oberpfaffenhofen, April - Juli 2023

E r k l ä r u n g

Ich versichere wahrheitsgemäß, die Praktikumsarbeit selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten Anderer unverändert oder mit Abänderungen entnommen wurde.

Leipzig, den 01.08.2023

Arthur Schletter

INHALT

Liste der verwendeten Abkürzungen und Formelzeichen

DLR	Deutsches Zentrum für Luft- und Raumfahrt
IMF	Institut für Methodik der Fernerkundung
RPi	Raspberry Pi
UDP	User Datagram Protocol
WGS84	World Geodetic System 1984
EGM2008	Earth Gravitational Model 2008
OpAiRS	Optical Airborne Remote Sensing & Calibration Homebase
NGA	US-amerikanische National Geospatial Intelligence Agency
GNSS	Global Navigation Satellite System

1 Einleitung

Ich habe das Pflichtpraktikum meines Maschinenbau-Bachelorstudiums am Deutschen Zentrum für Luft- und Raumfahrt (DLR), dort am Institut für Methodik der Fernerkundung (IMF) absolviert. In den 14 Wochen hatte ich die Möglichkeit, in der Abteilung Experimentelle Verfahren zwei Projekte zum Thema Navigationsdatenverarbeitung in der Programmiersprache Python zu bearbeiten.

Zu den Aufgaben dieser Abteilung gehört in der Gruppe „Validierung“ unter anderem die „Optimierung von Sensorkalibrierung und Messaufbauten für spezifische Aufgabenstellungen, u.a. zur Unterstützung der Methodenentwicklung für die Gewässerfernerkundung“ /1/. Im Zuge dieser Gewässerfernerkundung sollen Daten über Binnengewässer, insbesondere Seen, mittels optischer Satellitenaufnahmen im multi- und hyperspektralen Bereich gesammelt werden. Um aus den Aufnahmen nutzbare Daten zu gewinnen, müssen diverse von der Atmosphäre verursachte Einflüsse mittels Algorithmen entfernt werden.

Zur Überprüfung der Arbeit dieser Algorithmen werden die Ergebnisse mit Messungen direkt an der Wasseroberfläche verglichen. Um diese möglichst exakt durchführen zu können, wurde ein ferngesteuertes Messboot (LimnoVIS) gebaut, welches mit vier Motoren im 90°-Winkel ideal darauf ausgelegt ist, die durch Wind und Strömungen bedingte Drift des Bootes auszugleichen.

Auf dem Messboot ist neben umfangreicher optischer Sensorik auch ein Echolot und ein GPS-Empfänger installiert. Ein Flight Controller übernimmt sowohl die Steuerung des Bootes als auch den Driftausgleich. Zur Steuerung der Sensorik und zur Aufzeichnung der Messdaten ist ein Embedded Computer vom Typ „Raspberry Pi“ (RPi) eingebaut.

In den ersten Wochen meines Praktikums hatte ich die Aufgabe, ein Programm in der Programmiersprache Python zu schreiben, welches später auf diesen RPi übertragen werden kann. Es sollte einen Datenstrom vom GPS (weitergeleitet über den Flight Controller) und einen zweiten direkt vom Echolot miteinander verknüpfen, sodass am Ende für jeden Datensatz des Echolots die zugehörigen GPS-Daten in einer gemeinsamen Datei gespeichert sind. Gleichzeitig sollten die Echolotdaten auch noch unverändert an einen weiteren Computer weitergeleitet werden. Das Programm sollte so funktionieren, dass es entweder direkt auf dem im Boot eingebauten RPi laufen kann oder auf einem externen Computer, der die Echolot- und GPS-Daten mittels User Datagram Protocol (UDP) empfängt.

Im Folgenden (Kapitel 2) beschreibe ich die verwendete Software der Messgeräte, die Grundlagen der Datenprotokolle der Messgeräte und die Vorgehensweise bei der Entstehung des Programms, aufgetretene Probleme und ihre Lösungen.

Die zweite Hälfte meines Praktikums habe ich in der Gruppe „Kalibrierung optischer Systeme“ verbracht. Diese beschäftigt sich unter anderem mit der Hyperspektraltechnologie. Der Einsatz

entsprechender Sensoren für Erdbeobachtungsaufgaben erfolgt z. B. vom Flugzeug aus. Während des Fluges werden Daten gesammelt, die später ausgewertet werden.
Für die Flugplanung ist die Höhe, in der geflogen wird, selbstverständlich relevant. Das Problem ist nun aber, dass die Höhe, die ein GPS-Gerät ermittelt, sich an einem mathematisch leicht beschreibbaren Modell der Erdkugel orientiert – dem Ellipsoid –, welches an einigen Stellen nicht unwesentlich von der tatsächlichen Erdform – dem Geoid – abweicht. Die Flughöhe, in der gemessen werden soll, angegeben in auf tausender gerundeten Fuß, z. B. 7000 ft MSL (mit MSL = mean sea level, also Höhe über dem mittleren Meeresspiegel), muss vor dem Flug festgelegt werden. Diese MSL-Höhe wird von den Wissenschaftlern vor Flugbeginn bestimmt und dann an die Piloten weitergegeben. Diese Angabe erfolgt einfacher als Höhe über dem Geoid, da diese eher den barometrischen Druckwerten entspricht, mit denen die Messgeräte im Flugzeug arbeiten. Deshalb benötigt man eine Umrechnung zwischen Höhen relativ zum Ellipsoid und Höhen relativ zum Geoid. Diese ist in einem weltweiten Standard, dem WGS 84 festgelegt /2/. Meine Aufgabe war es, ein Programm zu schreiben, wiederum in Python, welches für jeden gegebenen Punkt auf der Erdoberfläche, mit Breiten- und Längengrad, den Abstand zwischen Ellipsoid und Geoid, die sog. Geoidundulation, berechnet. In Kapitel 3 dieser Arbeit erläutere ich die mathematischen Hintergründe und wiederum bei der Programmierung aufgetreten Probleme und ihre Lösungen sowie eine Auswertung der Berechnung bezüglich ihrer Genauigkeit.

2 Zusammenführung von Echolot- und GPS-Daten eines Messboots

2.1 Ausgangslage und Ziel des Projekts

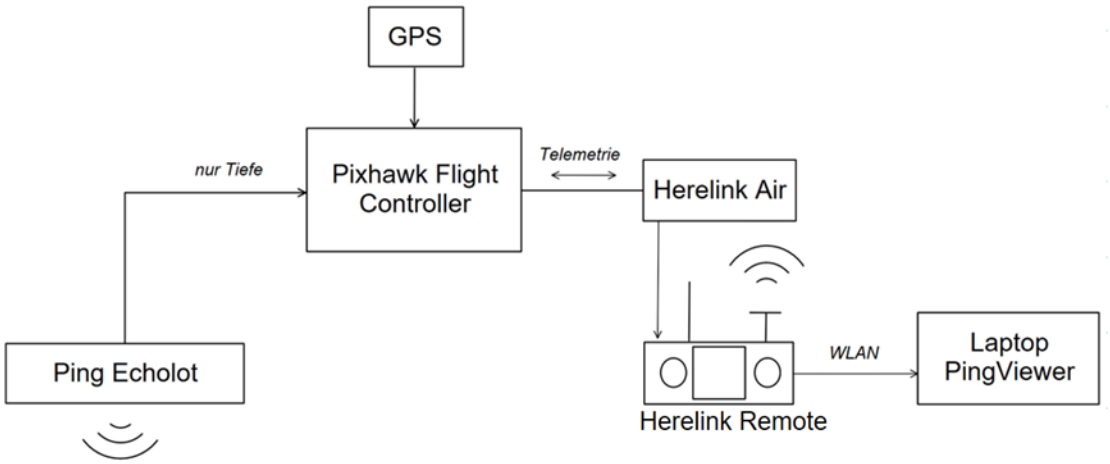


Abb. 2.1: Ursprünglicher Aufbau der relevanten Funktionselemente im Messboot (schematisch)

Das Messboot (LimnoVIS) war bereits ohne das neue RPi-Programm im Einsatz. Der Aufbau wird schematisch durch Abb. 2.1 verdeutlicht. Gesteuert wird es über einen Flight Controller des Typs Pixhawk 4 (Hersteller: Holybro), auf dem die Open-Source-Software Ardupilot läuft. Dieser liest über eine serielle Schnittstelle die Daten des GPS-Empfängers ein.

An einer zweiten seriellen Schnittstelle ist das Echolot angeschlossen. Hier wird ein Sonar vom Typ „Ping1D“ der Firma Bluerobotics genutzt, das mit einer Frequenz von 10 Hz die Wassertiefe misst. Der Flight Controller gibt die gemessene Wassertiefe zusammen mit den aktuellen GPS-Positionsdaten innerhalb des Telemetrie-Datenstroms an die Fernsteuerung zurück, wo sie während der Messung visualisiert und auch aufgezeichnet werden können, so dass nach Abschluss der Messfahrt vom befahrenen Gebiet eine Tiefenkarte erstellt werden kann. Zusätzlich werden die Tiefen- und GPS-Daten auch auf einer SD-Karte im Flight Controller mitgeloggt.

Das Sonar kann allerdings nicht nur den einen Wert der Wassertiefe ausgeben. Eine von ihm ausgesendete Schallwelle gibt zahlreiche Echos zurück, zum Beispiel von Fischen, Pflanzen und auch abhängig davon, ob der Untergrund hart oder eher weich ist. Dieses Echogramm kann als sog. Wasserfalldiagramm über die Software Ping Viewer dargestellt werden (vgl. Abb. 2.2). Dabei ist an der rechten Seite das aktuelle Echogramm zu sehen, die Stärke des Ausschlags von der Mittellinie verweist dabei auf die Stärke des Echos und somit auf die Härte der Oberfläche, die das Echo ausgelöst hat.

Im zeitlichen Verlauf bewegen sich die einzelnen Echogramme stetig nach links, wo die Stärke des Echos im Folgenden durch verschiedene Farben repräsentiert werden: rot zeigt ein starkes Echo, blau ein schwaches bis keins.

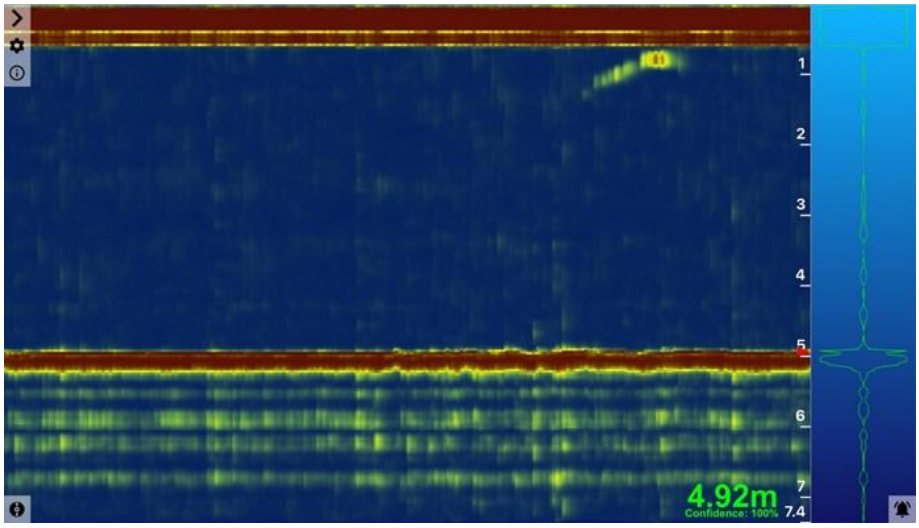


Abb. 2.2: Beispiel eines Wasserfalldiagramms im Ping Viewer, entnommen aus /3/

Diese Echogrammdaten liefern interessante weitere Informationen, zum Beispiel ob der Grund des Gewässers aus hartem Gestein oder Kies (starkes Echo) oder Sediment (weich, liefert ein eher verwaschenes Echo) besteht. Schließt man das Sonar an einen Computer an, auf dem der Ping Viewer installiert ist, kann so während der Messfahrt der zeitliche Verlauf des Echogramms beobachtet und auch aufgezeichnet werden. In diesem Fall laufen die Sonardaten jedoch nicht über den Flight Controller und werden von ihm dementsprechend auch nicht mit einem GPS-Marker versehen. Es ist also nach Abschluss der Messungen bei der Auswertung der Daten nicht mehr möglich, den einzelnen Echogramm-Datensätzen genaue Koordinaten zuzuweisen und die dadurch gewonnenen Informationen in die Kartierung einfließen zu lassen.

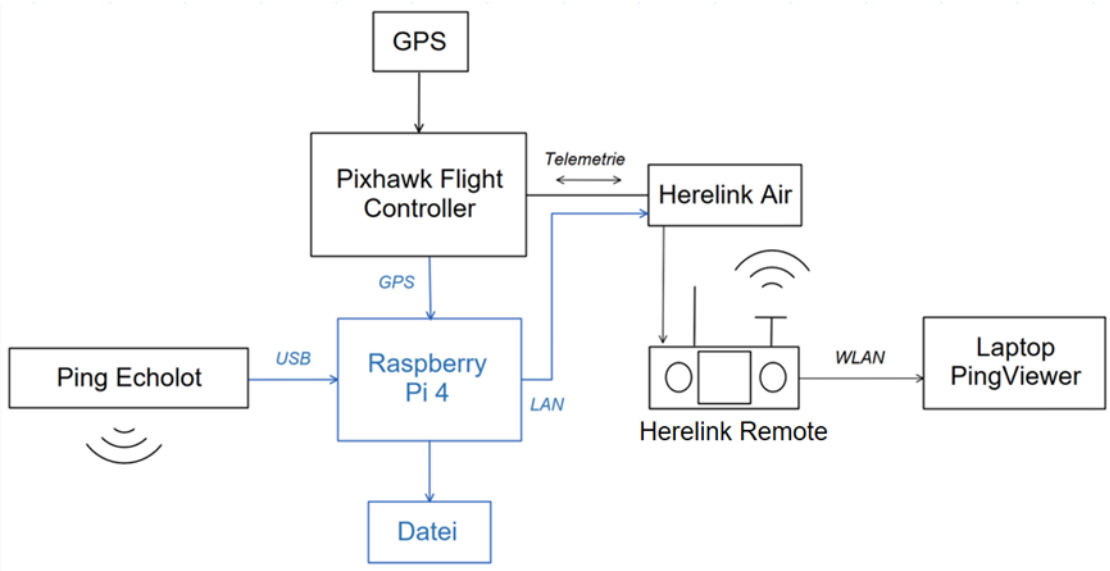


Abb. 2.3: Neuer Aufbau der Funktionselemente mit Raspberry Pi (Veränderungen zum ursprünglichen Aufbau in blau)

Dies soll mit dem neuen Programm verbessert werden (schematischer Aufbau in Abb. 2.3). Auf dem ins Messboot eingebauten Raspberry Pi 4 soll ein Programm laufen, welches die

GPS-Daten und Echogramme für jeden Zeitpunkt der Messungen verknüpft und gemeinsam in eine Datei schreibt. Gleichzeitig soll die Weiterleitung der reinen Echogrammdaten an einen Computer mit installiertem PingViewer über das Netzwerk (UDP) erfolgen, sodass eine Live-Verfolgung nach wie vor möglich ist.
Zur Verknüpfung der Daten werden die GPS-Daten aus dem vom Flight Controller kommenden seriellen Datenstrom (Mavlink-Protokoll) eingelesen. Für diesen Zweck gibt es eine Mavlink-Bibliothek für Python (pymavlink, /4/), über die das Parsen und das Filtern der gewünschten Informationen erfolgen kann. Ähnliches gilt für die Daten vom Echolot; auch für das Einlesen dieser Daten über eine serielle Schnittstelle (USB) gibt es eine Python-Bibliothek, die einfach importiert werden kann (brping, /5/) und die dann genutzt werden kann, um eine Verbindung zum Echolot aufzubauen und die Nachrichten einzeln aus dem Datenstrom zu extrahieren.
Im Programm werden die Datensätze, die jeweils gleichzeitig entstanden sind, verknüpft und in eine Binärdatei geschrieben, die später wieder ausgewertet werden kann. Zusätzlich soll das Echogramm unverändert über eine LAN-Verbindung an den Herelink Air zurückgegeben werden, sodass es von dort aus über die Fernsteuerung an den externen Laptop weitergegeben und dort mit dem Ping Viewer visualisiert werden kann

2.2 Genereller Aufbau des Python-Programms

Das vollständige Programm arbeitet, aufgeteilt in einzelne Funktionen, folgende Schritte ab:

1. Vorbereiten der Binärdatei (Datei erstellen oder leeren und Header schreiben)
2. Parsen der über die Kommandozeile eingegebenen Parameter
3. Verbindung zum Echolot und GPS-Gerät aufnehmen
4. Daten einlesen und ggf. für die Weiterverarbeitung formatieren
5. Echolotdaten unverändert über eine UDP-Verbindung an ein Gerät mit Ping Viewer schicken
6. Zusammengehörige Daten von GPS und Echolot gemeinsam in die Datei schreiben

Schritt 4 bis 6 wiederholen sich im Folgenden in einer While-True-Endlosschleife, bis das Programm manuell abgebrochen wird.
Im ersten Schritt wird die Binärdatei erstellt oder, falls sie bereits vorhanden ist, geleert. Dann wird der Header erzeugt. Dieser ist notwendig, damit auch die auf dem RPi gespeicherte Datei am Ende vom Ping Viewer wieder visualisiert werden kann.
Generell kann der Ping Viewer entweder einen hereinkommenden Datenstrom direkt darstellen oder eine Logdatei von einem vorher aufgezeichneten Datenstrom erneut abspielen. Idealerweise hat deshalb die von unserem Programm geschriebene Datei das gleiche Format wie das Logfile. Fürs Schreiben des Headers werden Open-Source-Codes aus der Dokumentation für den Ping Viewer verwendet, die auch bei einer gewöhnlichen Aufnahme von Echolotdaten

den Header der Logdatei schreiben. Sie sind zu finden unter /6/. Die Daten, die der Header enthält, sind in der Dokumentation vom Ping Viewer zu finden (/7/) und beinhalten unter anderem Informationen über das Programm, wie Name und Version des Betriebssystems, aber auch Informationen über den Sensor. Da der Header nicht vom Programm selbst generiert wird, werden diese Daten manuell eingegeben. Dieser Vorgang ist Teil der write_header()-Funktion ab Zeile 27 im Code.
Eine besondere Herausforderung war die korrekte Formatierung des Headers, wie es der Ping Viewer automatisch tut, sodass die Daten später korrekt abgespielt werden können. Das Programm formatiert die Zeichen des Logfiles als 16-bit-Gruppen, während der von der Funktion formatierte Header als 8-bit-Gruppen in die Datei geschrieben wurde. Um dieses Problem zu umgehen und die Datei für den Ping Viewer später lesbar zu machen, wird der gesamte Header über eine for-Schleife Byte für Byte entsprechend angepasst (vgl. Abb. 2.4):

```
header_list = list(header_packed)
    for i in range(0, len(header_list)):
        if header_list[i] != 0x00:
            byte = header_list[i].to_bytes(2, 'little')
        else:
            byte = header_list[i].to_bytes(1, 'little')
        storage.write(byte)
    storage.close()
```

Abb. 2.4: Formatierung des 8-bit-Headers als 16-bit Logfile

2.3 Durchführung der Programmierung, aufgetretene Probleme und Lösungen

Neben der Möglichkeit, GPS- und Echolotdaten über serielle Schnittstellen einzulesen, gibt es eine zweite Möglichkeit, bei der beide Datenströme über UDP eingelesen werden. Die erste Version dient für den oben beschriebenen Einsatz, bei dem das Programm direkt auf dem im Messboot eingebauten RPi läuft. Bei einer zweiten Variante wird das Programm von einem externen Rechner aus gestartet. Auf dem Messboot läuft nur ein kleines Programm, welches die seriellen Schnittstellen ausliest und als Server beide Datenströme über UDP weiterschickt.

2.3.1 Input und Trennung der Versionen

Die für den Betrieb notwendigen Parameter werden beim Programmstart über die Kommandozeile eingegeben (s. Abb. 2.5). Zuerst wird die Version des Programms eingestellt (connection serial oder udp), im Folgenden für Version 1 (serielle Schnittstellen) jeweils die COM-Ports, an denen die Geräte (Ping und GPS-Gerät bzw Flight Controller) per USB angeschlossen sind und die entsprechende Baudrate (standardmäßig ist es 115200) sowie den Output-Port, über den die unveränderten Echogramm-Daten weitergeschickt werden sollen. Wird die zweite Version verwendet, werden statt der COM-Ports und Baudrates die IP-Adresse des

Servers, der die Ping- und GPS-Daten übergibt – also des Computers auf dem Messboot – sowie die entsprechenden Ports beider UDP-Verbindungen benötigt. Auch hier muss der Output-Port angegeben werden.

```
Programm im serial-Modus starten:
>>> gps-echolot.py --connection serial --ping_device COM3 --ping_baudrate 115200 --gps_device COM4 --
gps_baudrate 115200 --output_port 5002

Programm im UDP-Modus starten:
>>> gps-echolot.py --connection udp --host 127.0.0.1 --gps_port 5000 --gps_port 5001 --output_port 5002
```

Abb. 2.5: Beispiel für das Starten des Programms in beiden Modi

Das Erste, was das Programm nach dem Parsen der Eingabeparameter tut, ist die Version zu prüfen. Über eine If-Verzweigung wird der Input unter connection ausgewertet. Ist dort etwas Anderes als udp oder serial eingetragen worden, bricht es sofort ab und fordert den Anwender auf, eine der beiden Möglichkeiten zu wählen. Ansonsten startet die jeweilige Version.

2.3.2 Version 1 – Input über serielle Schnittstellen

Das Programm wurde ursprünglich geschrieben, um ausschließlich diese Verwendung zuzulassen, deshalb ist ein Großteil der Arbeitszeit in die Programmierung diesen Codes geflossen. Durch die zunehmende Erfahrung während des Entwicklungsprozesses war die Anpassung auf die Verwendung von UDP als Input im Späteren ein weniger großer Aufwand.
Zunächst ging es darum, die Daten vom Echolot auf eine Weise ins Programm einzulesen, die eine weitere Verwendung ermöglichte. Dazu gibt es generell zwei Möglichkeiten: Klassisch existiert das oben genannte Ping Protocol, welches über die Python-Bibliothek brping Daten vom Gerät anfordern kann.

```python
from brping import Ping1D


#Einlesen der Daten vom Echolot:
def readEcholot():
    myPing = Ping1D()
    myPing.connect_serial("COM3", 115200)
    myPing.initialize()
    data = myPing.get_profile()
    return data


data = readEcholot()            # Daten vom Echolot werden eingelesen
```

Abb. 2.6: Code zum Anfordern des Echogramms vom Ping Sonar

Die zurückgegebenen Daten vom get_profile-Befehl (data in Abb. 2.6), sind ein Dictionary mit Informationen wie der Tiefe (als ein einzelner Wert), der Confidence des Programms (also der Sicherheit bezüglich der Korrektheit der Daten, in %) und den Profildaten als Bytes in hexadezimaler Schreibweise (vgl. /8/).

Tabelle 1: Message-Format einer Nachricht zwischen Ping Sonar und Ping Viewer nach /8/

Byte	Type	Name	Description
0	u8	start1	Start frame identifier, ASCII 'B'
1	u8	start2	Start frame identifier, ASCII 'R'
2-3	u16	payload_length	Number of bytes in payload.
4-5	u16	message_id	The message id.
6	u8	src_device_id	The device ID of the device sending the message.
7	u8	dst_device_id	The device ID of the intended recipient of the message.
8-n	u8[]	payload	The message payload.
(n+1)-(n+2)	u16	checksum	The message checksum. The checksum is calculated as the sum of all the non-checksum bytes in the message.

Leider führt die Übergabe dieser Datensätze an den PingViewer nicht zu einer Ausgabe dort. Betrachtet man sich das Protokoll für das Message-Format des Ping Viewers (Tabelle 1), fällt auch schnell auf, warum: Der get_profile-Befehl gibt nur die Daten selbst (message payload) zurück, ohne Header (Byte 0 bis 7) und Checksum (die letzten beiden Bytes) zu generieren.
Man hätte an dieser Stelle eine Funktion schreiben können, um dies zu ergänzen. Allerdings gibt es im GitHub einen fertigen Code, den sogenannten Ping Proxy, bei dem mehrere UDP-Clients mit einer seriellen Schnittstelle verbunden werden können. Der Import der Klassen PingProxy und PingClient aus diesem Programm (pingproxy.py, s. /9/) vereinfacht diese Aufgabe immens. Durch die Methode run() der Klasse PingProxy werden die Daten vom Ping Sonar eingelesen und über den UDP-Output-Port an einen Ping Viewer lesbar weitergegeben. Gleichzeitig ermöglicht es die Extraktion der hexadezimalen Profildaten durch eine simple Ergänzung einer return device_data-Funktion im originalen Code. Die hex-Daten können einfach in ein File geschrieben werden.
Als Nächstes müssen die GPS-Daten eingelesen werden. Dies geschieht über eine readGPS()-Funktion, die aus dem eingehenden String Längen- und Breitengrad sowie Höhe über dem Referenzellipsoid (vgl. Kapitel 3) extrahiert und als Float speichert. Wie oben bereits beschrieben gibt es auch dafür eine Python-Bibliothek (pymavlink). Diese Informationen können als Status des sekundären AHRS-Filters (AHRS2) angefordert werden (vgl. Abb. 2.7). Er enthält neben der Höhe in Metern als Float und Längen- und Breitengrad in Grad * 10^7 als Integer auch noch Roll-, Nick- und Gierwinkel (englisch roll, pitch und yaw), die zur Beschreibung der Ausrichtung des Bootes dienen (vgl. /10, #178 im Abschnitt MAVLink Messages/). Letztere haben in unserem Fall eine geringere Relevanz und kommen vor allem daher, dass die gesamte ArduPilot- und Mavlink-Software eigentlich auf Drohnen ausgerichtet ist.

Die einzelnen Werte für Längen- und Breitengrad sowie Höhe werden aus der Nachricht extrahiert und im Falle der ersten beiden in einen Float_Wert umgewandelt und durch 10^7 geteilt, um wieder den eigentlichen Wert in Dezimalgrad zu erhalten.

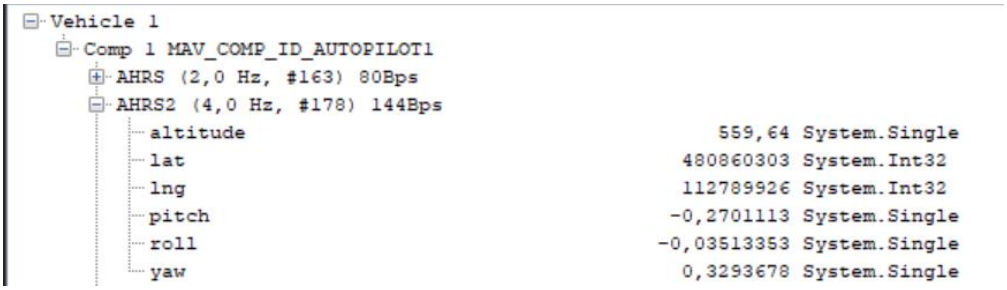


Abb. 2.7: Beispielhafter Status des AHRS2-Filters, entnommen aus dem MAVlink Inspector der Mission-Planner-Software

Alle drei Werte werden zusammen mit einem Zeitstempel (Computerzeit in Sekunden) aus der Funktion wieder ans Hauptprogramm übergeben und im Folgenden in Bytes umgewandelt und ebenfalls ins File geschrieben.

Ein Problem mit dem Einlesen der GPS-Daten war, dass die über Mavlink übermittelten Daten in der Frequenz nicht mit den Ping-Daten mithalten können. Die Ping-Daten kommen mit einer Frequenz von 10 Hz (realisiert dadurch, dass am Ende der While-Schleife eine Wartezeit von 0,1 Sekunden eingebaut ist). Damit hat das Programm die gleiche Frequenz, wie wenn der Ping Viewer normal läuft. Die Frequenz der eingehenden GPS-Daten ist etwas geringer. Sie schwankt, liegt aber zumeist um 4 Hz (vgl. Abb. 2.7). Dies führt dazu, dass nicht bei jedem eintreffenden Ping-Paket neue GPS-Daten vorliegen. Um Lücken in den Daten zu vermeiden, wurden die prev-Werte (lat_prev, long_prev, alt_prev) eingeführt. Sie werden in jedem Vorgang mit den aktuellen Daten aktualisiert. Falls im nächsten Durchlauf keine neuen GPS-Daten vorhanden sind, können die Echolotdaten zwar nicht exakt, aber zumindest mit dem letzten bekannten Ort gekennzeichnet werden. Bei einer Ping-Frequenz von 10 Hz und einer Geschwindigkeit des Bootes von 1 m/s wächst der Fehler bei der Lokalisierung um 10 cm pro Echopuls, bis eine aktualisierte GPS-Position vorliegt. Dies geschieht durchschnittlich alle 2,5 Echopulse, die maximale Abweichung von der realen Position liegt also bei ca. 25cm.

Durch die Abfolge des Programms besteht die erstellte Binärdatei immer aus abwechselnd dem Datensatz vom Echolot und dann den GPS-Daten mit Zeitstempel.

Das Auslesen der GPS-Daten durch den PingViewer ist natürlich in der originalen Programmierung nicht vorgesehen. Damit eine mögliche Anpassung in der Software (möglich dadurch, dass der Ping Viewer vollständig open source ist, aber nicht im Zuge dieses Projektes erfolgt) erleichtert wird, werden die GPS-Daten in ein Nachrichtenpaket im Format nach Tabelle 1 folgendermaßen verpackt:

Tabelle 2: Aufbau der Nachricht mit den GPS-Daten nach /8/

Byte	Typ	Inhalt	Für die GPS-Nachricht
0	u8	start1	Start-Kennzeichner, ASCII 'B'
1	u8	start2	Start-Kennzeichner, ASCII 'R'
2-3	u16	payload_length	Anzahl der Bytes in der Payload = 14
4-5	u16	message_id	Message ID (willkürlich gewählt aus nicht genutzten IDs) = 1301
6	u8	src_device_id	Geräte-ID des sendenden Geräts = 1
7	u8	dst_device_id	Geräte-ID des angestrebten Empfängers = 0
8-11	u32	payload	Breitengrad (latitude) in ddeg
12-15	u32		Längengrad (longitude) in ddeg
16-17	u16		Höhe (altitude) in m
18-21	u32		Computerzeit in s
22-23	u16	checksum	Checksum, berechnet als Summe aller Bytes der Nachricht

Die GPS-Daten sind also als Nachricht mit einer dem System unbekannten Message-ID verpackt, die aber grundsätzlich den Regeln des Nachrichtenaufbaus im Ping Protocol folgt. Eine Anpassung im Ping Viewer ist dementsprechend vermutlich nicht übermäßig schwierig, soll an dieser Stelle allerdings nicht mit ins Projekt eingeschlossen werden. Aktuell ist es ausreichend, jeweils die Ping- und GPS-Daten aufeinanderfolgend in der Datei zu finden und mit einem geeigneten Programm wieder auslesen und in Bezug zueinander auswerten zu können.

2.3.3 Version 2 – Input über UDP-Verbindungen

Für die Anpassung des Programms für einen Input durch UDP-Verbindungen musste zum Test des Programms zuerst ein zweites Programm geschrieben werden, welches die seriellen Datenströme von Echolot und GPS als UDP ausgibt. Dieses Programm müsste auch bei der Anwendung des Hauptprogramms auf einem externen Rechner auf dem RPi laufen, um zu gewährleisten, dass überhaupt UDP-Datenströme von beiden Datensätzen existieren, die vom Programm eingelesen werden können.
Das Hauptprogramm wird zunächst genauso gestartet, wie in der ersten Version: Es leert die Datei, schreibt den Header und parst die Parameter aus der Kommandozeile. Anders als oben stellt es aber nicht selbst eine Verbindung zu den Messgeräten her, sondern richtet lediglich an den aus den Parametern entnommenen Ports Sockets für eine UDP-Verbindung ein und wartet im Folgenden auf Daten vom Server (vgl. Abb. 2.8).
Das Hilfsprogramm auf dem RPi fungiert als dieser Server, es erfüllt Teile der Aufgaben, die in Version 1 das Hauptprogramm selbst übernommen hat: Es baut mit dem mavutil-Modul eine Verbindung zum GPS-Gerät auf, liest die Daten ein, extrahiert sie aus dem String und wandelt sie in Float-Werte der richtigen Einheiten um. Auch hier existieren die in jedem Durchgang

aktualisierten prev-Werte, die verwendet werden können, solange keine neuen GPS-Daten vorliegen.

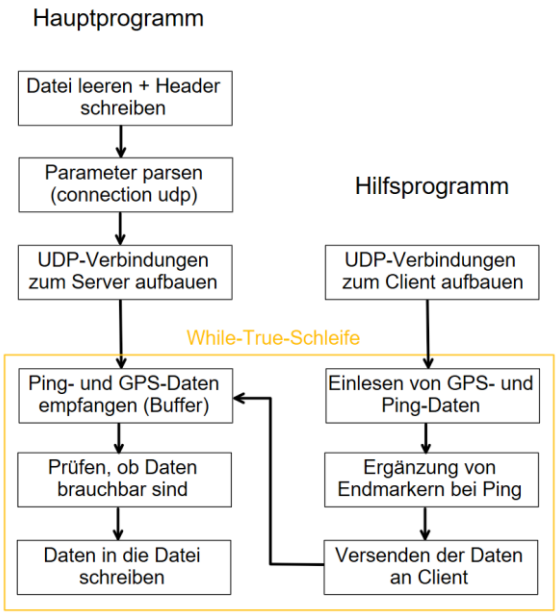


Abb. 2.8: Schematischer Aufbau und Zusammenhang von Hauptprogramm (auf einem extrernen Rechner) und Hilfsprogramm (auf dem RPi) bei einer Verbindung über UDP

Die Echolot-Werte werden auch hier über das importierte pingproxy.py-Programm eingelesen. Hier wird jedoch nur die Einlesefunktion des Programms genutzt, statt dass die Daten damit auch direkt weitergegeben werden. Der angegebene proxy_port sendet die Daten zwar weiter, sie werden jedoch vom Hauptprogramm nicht genutzt, da sie in ihrer Rohform durch den Buffer des Clients (in diesem Fall das Hauptprogramm) nicht korrekt ausgelesen werden können.
Das Hilfsprogramm verpackt die beiden Datenpakete als Bytes und verschickt sie über die zu Anfang aufgebauten UDP-Verbindungen an den Client (das Hauptprogramm auf dem externen Rechner).
Hier war die Idee, dass die Datenströme einfach eingelesen werden können und dann zur weiteren Verarbeitung auf die gleiche Weise wie in Version 1 (über die Funktionen data_to_file und gps_to_file) sowie zum Output über den Socket an den Ping Viewer zur Verfügung stehen. Diese Annahme stellte sich recht schnell als falsch heraus.
Durch den Buffer beim Einlesen von UDP-Datenströmen kommen die Werte von GPS und Echolot im Hauptprogramm nicht mehr Nachricht für Nachricht an, sondern zerstückelt, wann immer der Buffer voll ist. So können sie aber natürlich weder brauchbar in die Datei geschrieben, noch im Falle der Ping-Daten lesbar an den Ping Viewer weitergegeben werden.
Die Lösung dieses Problems war eine Anpassung des Buffers und die Einführung von eindeutigen Endmarkern für die Nachrichten.
Letzteres geschieht noch im Hilfsprogramm. Damit die Datenpakete einen eindeutigen Anfang und Ende haben, wird der String mit GPS-Daten vor der Umwandlung in Bytes in geschweifte

Klammern geschrieben. Bei den Ping-Daten werden am Ende einer Nachricht drei Bytes (b'END') zum gleichen Zweck ergänzt.

```
class Buffer(object):
    def __init__(self, sock):
        self.sock = sock
        self.buffer = b""

    def get_line(self, marker):
        while marker not in self.buffer:
            data = self.sock.recv(4096)
            if not data:
                return None
            self.buffer += data
        line, sep, self.buffer = self.buffer.partition(marker)
        return line.decode()
```

Abb. 2.9: Buffer-Klasse mit get_line-Methode fürs Dekodieren vollständiger Nachrichten aus dem Buffer der UDP-Verbindung

Im Hauptprogramm werden zunächst Objekte der Klasse Buffer() für beide Input-Sockets erstellt, für die dann jeweils die Methode get_line ausgeführt werden kann. Die Endmarker (die Bytefolgen „}" und „END") werden der Methode übergeben, die dann den eingehenden Datenstrom nach dem Marker absucht und die Nachricht entsprechend übergibt (vgl. Abb. 2.9).
Auch hier muss die recv-Funktion der Sockets natürlich eine Buffergröße festgelegt haben. Da die Größe der Datenpakete von Ping- und GPS-Daten jedoch zwar schwanken, allerdings stets in einem ähnlichen Rahmen bleiben, kann diese so deutlich über den Maximum gewählt werden, dass immer mindestens eine ganze Nachricht im Buffer vorliegt.
Die von der Methode zurückgegebenen Pakete werden im Hauptprogramm in beiden Fällen auf ihren Startmarker kontrolliert (im Falle der GPS-Daten die öffnende geschweifte Klammer, im Falle der Ping-Daten die Startbytes BR nach Tabelle 1). Ist dieser ebenfalls korrekt, werden jetzt die Funktionen für die Dokumentation in der Datei und die Weiterleitung der Ping-Daten ausgeführt. Wichtig zu beachten ist dabei, dass zunächst die Ping-Daten kontrolliert werden. Sollte kein nutzbares Datenpaket vorliegen, wird die While-Schleife sofort von vorne gestartet, ohne die GPS-Daten auszuwerten, sodass die Abfolge von immer abwechselnden Ping- und GPS-Daten in der Datei nicht gestört wird.

3 Programm zur Berechnung von Geoidundulation

3.1 Hintergrund des Projektes

Das IMF betreibt seit 2007 in Kooperation mit dem DLR-Flugbetrieb über den Nutzerservice OpAiRS (Optical Airborne Remote Sensing & Calibration Homebase) flugzeuggetragene Sensoren für die Fernerkundung, die auch von externen Partnern in Anspruch genommen werden können. Dazu gehört auch die Planung, Organisation und Durchführung von Messkampagnen (/11/).

Verwendet werden dabei unter anderem auch zwei HySpex-Sensoren: Zeilenkameras, die räumlich und spektral hochauflösende Messdaten vom Flugzeug aus aufnehmen können. Mit ihnen können zum Beispiel auch Daten des deutschen Hyperspektralsatelliten EnMAP (Environmental Mapping and Analysis Program) simuliert und validiert werden, da sich beide im Spektralbereich und in der Funktionsweise ähneln.

Hyperspektrale Daten haben zahlreiche Anwendungen und kennzeichnen sich dadurch, dass sie, anders als z. B. das menschliche Auge, welches die Umwelt in den Wellenlängen rot, grün und blau wahrnimmt (multispektral), sehr viele eng beieinander liegende Wellenlängen, sog. Spektralkanäle, aufzeichnen, die von ultraviolett bis infrarot reichen. Durch die Aufnahme von bis zu 250 Kanälen entsteht eine Datenmenge, aus der unterschiedlichste Schlussfolgerungen zu atmosphärischen, vegetativen, klimatischen oder urbanen Themen gezogen werden können /12/. Ein Beispiel dafür ist der Einsatz von EnMAP-Daten zur Vorhersage, Erkennung und Schadenseinschätzung bei Waldbränden /13/.

Wie bereits in der Einleitung beschrieben, wird für die Flugplanung die Umrechnung zwischen Höhe über dem Geoid und Höhe über dem Ellipsoid benötigt. Zunächst wären also diese beiden Begriffe zu definieren und zu erläutern, worin sie sich im Wesentlichen unterscheiden.

3.2 Theorie zu Geoid und Ellipsoid

Es ist allgemein bekannt, dass die Erde keine Kugel ist. Die tatsächliche Form der Erde ist durch die Rotation an den Polen abgeflacht, wird von der Oberfläche, z. B. Gebirgen, beeinflusst und ist allgemein mathematisch sehr schwer zu beschreiben. Annäherungen an diese tatsächliche Form sind zum Beispiel sogenannte Rotationsellipsoide. Ein solches entsteht bei der Drehung einer Ellipse um eine ihrer Achsen, im Falle des Erdellipsoids um die kleine Achse (oblates Ellipsoid, vgl. /14/).

Diese Ellipsoide, von denen es unendlich viele gibt, die teils gute und teils weniger gute Annäherungen an die tatsächliche Erdform sind, haben den großen Vorteil, dass sie mathematisch verhältnismäßig leicht zu beschreiben sind. Der zweifellose Nachteil von ihnen ist jedoch, dass sie die wahre Form der Erde im Endeffekt nur grob annähern.

Wenn Punkte auf oder über der Erdoberfläche exakt bestimmt werden sollen, erfolgt dies in der Regel mit drei Koordinaten – dem Breitengrad, dem Längengrad und der Höhe, meist über dem Meeresspiegel. Doch was genau ist der Meeresspiegel? Wo liegt er? Wenn man einen Gipfel im Himalaya angibt, ist das nächste Meer recht weit entfernt, an welchem Meeresspiegel orientiert man sich?

Die Antwort liefert die Gravitationstheorie. Auf der Erde gibt es überall die Fallbeschleunigung g. Diese ist der negative Gradient eines Potenzials U, in der physikalischen Geodäsie bezeichnet als W = -U. Abhängig von der Rotation der Erde sowie Dichteunterschieden im Inneren (inklusive Gebirgen, Ozeanen, etc.), hat jeder Punkt auf oder über der Erdoberfläche ein solches Schwerepotenzial (vgl. /15/). Es setzt sich zusammen aus dem Gravitationspotential und dem Zentrifugalpotenzial an diesem Ort.

Flächen aus Punkten mit dem gleichen Potenzial werden als Äquipotentialflächen bezeichnet. Die wichtigste ist das sog. konventionelle Geoidpotential, welches eine bestimmte Äquipotentialfläche definiert: das Geoid. Dieses ist, nach Johann Benedict Listing (1871) definiert als die, „die Äquipotentialfläche des Schwerefelds der Erde auf dem Niveau des mittleren Meeresspiegels“.

Da die Höhe eines Punktes über dem Ellipsoid nur mit Satelliten präzise bestimmt werden kann, während die Höhe relativ zum Geoid über die Äquipotentialfläche mit Wasserwaage, Meterstab und Pfeilfernrohr messbar ist, sind wir bis heute auf das Geoid angewiesen. Die meisten praktischen Anwendungen, etwa die GPS-Ortsbestimmung, benötigen jedoch eine einfach mathematisch bestimmbare Figur und eine solche ist das Geoid leider nicht. Hier wird deshalb nach wie vor ein Ellipsoid verwendet, das fest definierte Ellipsoid nach dem WGS 84-Standard (World Geodetic System), siehe /2/.

Das WGS-84-Ellipsoid ist das „best fitting“ (am besten passende) Ellipsoid für das Geoid, das ebenfalls in diesem Standard festgelegt ist. Dennoch gibt es Abweichungen der beiden voneinander, die bei einer exakten Bestimmung der Höhe, z. B. im Flugverkehr, berücksichtigt werden müssen. Der Abstand zwischen Geoid und Ellipsoid an einem Punkt mit den Koordinaten φ (Breitengrad) und λ (Längengrad) wird bezeichnet als Geoidhöhe oder Geoidundulation N(φ, λ). Sie liegt generell im Bereich von ca. -100 bis +100 m (vgl. /16/).

Es existieren Online-Rechner, bei denen man sich für einen Punkt auf der Erde mit Breiten- und Längengrad die Geoidundulation berechnen lassen kann. Einer von ihnen ist der der US-amerikanischen National Geospatial-Intelligence Agency (NGA), die auch den im Folgenden verwendeten Standard veröffentlicht hat (Rechner zu finden unter /17/). Die dort ausgegebenen Werte können als Referenz verwendet werden. Ein Problem dieser Daten ist allerdings, dass stets nur ein einziger Ort berechnet wird. Eine Auswertung größerer Datensätze ist somit sehr zeitaufwendig. Zudem kann die Funktion so nicht in ein anderes Projekt integriert werden,

in denen eine Umrechnung benötigt wird. Auch ist der Quellcode nicht einsehbar, es ist also unklar, wie genau die Werte berechnet werden.
Es gibt online einige Python-Bibliotheken, die Methoden zur Umrechnung zur Verfügung stellen. Bei open-source-Codes ist allerdings immer die Frage, wie aktuell die verwendeten Daten sind und auf welchen mathematischen Grundlagen sie basieren. Eine selbstständige Implementierung erschien an dieser Stelle deshalb die beste Lösung.

3.3 Mathematische Ansätze zur Berechnung der Geoidundulation

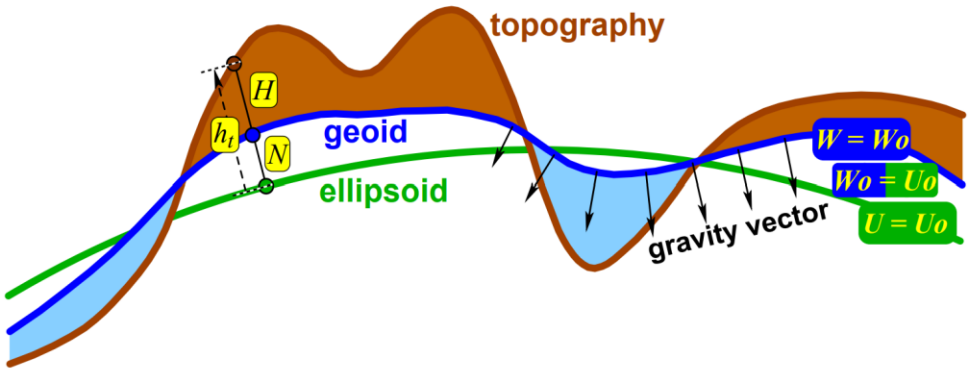


Abb. 3.1: Zusammenhang von Geoid, Ellipsoid und Topologie und den zugehörigen Höhenbezeichnungen (Quelle: /18, S. 6/)

Die Berechnung der Geoidundulation N wird im WGS-84-Standard /2, Kapitel 6, Gleichung 6-2 bis 6-7/ ausführlich beschrieben. Generell wird sie als Abstand zwischen der orthometrischen Höhe H (relativ zum Geoid) und der geometrischen Höhe h (relativ zum Ellipsoid) (vgl. Abb. 3.2, dort h_t) berechnet als

$$N = h - H. \tag{3-1}$$

Dabei wird, wie in Abb. 3.1 erkennbar, die orthometrische Höhe an jeder Stelle senkrecht zum Gravitationsfeld gemessen (vgl. gravity vector in Abb. 3.1). Um die Werte von h und H exakt zu bestimmen, sind allerdings detaillierte Informationen über die Massendichten der Erdkruste nötig, die schwer zu beschaffen sind. Eine Alternative stellt die Berechnung über die Normalhöhe H* und die Höhenanomalie ζ dar:

$$h \approx H^* + \zeta. \tag{3-2}$$

Die Normalhöhe wird definiert über die Potenzialdifferenz zweier Punkte. Da H und H* sehr nah beieinander liegen, stehen N und ζ im direkten Zusammenhang miteinander:

$$N(\varphi,\lambda) = \zeta(\varphi,\lambda,r) + \frac{\Delta g_B(\varphi,\lambda)}{\bar{\gamma}} * H(\varphi,\lambda) + \zeta_Z \tag{3-3}$$

wobei Δg_B die Bouguer Gravitationsanomalie nach EGM2008 und $\bar{\gamma}$ den Mittelwert der normalen Gravitation darstellt. Im WGS-Standard wird eine Null-Grad-Höhenanomalie ζ_Z addiert. Dieser Festwert von -0,41m wird in vielen anderen Publikationen vernachlässigt. Es konnte im

Verlauf der Berechnungen jedoch eine Verbesserung der Genauigkeit der Ergebnisse durch ihn festgestellt werden, weshalb er in die Berechnung mit einbezogen wurde (vgl. Kapitel 3.5.3). Während $\bar{\gamma}$ eine Konstante ist, stellt die Bestimmung der orthometrische Höhe H, die auch für die Berechnung

$$\Delta g_B(\varphi,\lambda) = -0{,}1119 * H(\varphi,\lambda) \tag{3-4}$$

benötigt wird, eine Herausforderung dar. Ähnliches gilt für den Abstand r eines Punktes auf der Erdoberfläche zum Massenschwerpunkt der Erde: Dieser lässt sich nur über ein globales Oberflächenmodell ermitteln.

Der WGS-Standard verweist für beide Probleme auf ein Paper von R. H. Rapp unter dem Titel „Use of potential coefficient models for geoid undulation determinations using a spherical harmonic representation of the height anomaly/geoid undulation difference“ (/19/).

Rapp präsentiert dort eine alternative Berechnung von N als

$$N(\varphi,\lambda) = \zeta_0(\varphi,\lambda,r_E) + C_1(\varphi,\lambda) + C_2(\varphi,\lambda), \tag{3-5}$$

mit ζ_0 in Abhängigkeit von r_E, dem Radius des Ellipsoiden an der Stelle (φ, λ) statt r auf der Topologieoberfläche. Die dadurch entstandene Abweichung wird durch die Konstante C_1 kompensiert, während C_2 den zweiten Term aus Gleichung (3-3) ersetzt (vgl. /19, Gl. (3) bis (6)/).

Die Höhenanomalie ζ_0 wird als mathematische Reihenentwicklung mit Kugelflächenfunktionen berechnet. Die Formel dafür lautet:

$$\zeta_0(\varphi,\lambda,r_E) = \frac{GM}{\gamma(\varphi) r_E}\left[\sum_{n=2}^{N_{max}}\left(\frac{a}{r_E}\right)^n \sum_{m=0}^{n}(\bar{C}_{nm}\cos m\lambda + \bar{S}_{nm}\sin m\lambda)\bar{P}_{nm}(\sin\varphi')\right] \tag{3-6}$$

mit der geozentrischen Gravitationskonstante GM, s. /2, Tabelle 3.1/ und der normalen Gravitation

$$\gamma(\varphi) = \gamma_e \frac{1+k\,sin^2\varphi}{\sqrt{1-e^2 sin^2\varphi}}, \qquad \text{mit} \qquad k = \frac{b\gamma_p}{a\gamma_e} - 1 \tag{3-7}$$

am Punkt P(φ, λ) auf der Erdoberfläche /2, Gl. (4-1)/.

Weiterhin symbolisieren

a und b die große und kleine Halbachse des Ellipsoids,

γ_e und γ_p stehen für die normale Gravitation am Äquator und den Polen und

e^2 ist die quadrierte erste Exzentrizität des Ellipsoiden.

Alle fünf Werte sind Konstanten, die im WGS-Standard definiert sind und Tabelle 3.1, 3.5 bzw. 3.6 aus /2/ entnommen werden können.

Des Weiteren:

r_E	Radius des WGS84-Referenzellipsoiden bei (φ, λ),
$\bar{C}_{nm}$ und $\bar{S}_{nm}$	vollständig normierte Gravitationskoeffizienten,
$\bar{P}_{nm}(\sin\varphi')$	vollständig normierte assoziierte Legendre-Funktion.

Die Gravitationskoeffizienten müssen dabei Tabellen entnommen werden, die dem WGS-Standard beiliegen. Bei der Auswertung der Legendre-Funktion ist zu beachten, dass mit φ‘ der geozentrische Breitengrad gemeint ist, während φ den geodätischen bezeichnet.
N_{max} bezeichnet die Ordnung der unterliegenden Differenzialgleichung, bei der die unendliche Summe abgeschnitten wird, um eine berechenbare, endliche Reihe zu erhalten. Sie ist im WGS-84-Standard in der verwendeten Version aus dem Jahr 2014 mit 2190 vorgesehen. Der Standard verweist auf die Verwendung der Koeffizienten $\bar{C}_{nm}$ und $\bar{S}_{nm}$ aus dem Earth Gravitational Model von 2008 (EGM2008, /20/), welches beide bis zu einem Grad und einer Ordnung von 2159 tabelliert, was somit die theoretische Einschränkung des Programmes darstellt. Die Koeffizienten $\bar{C}$ für m = 0 und gerade n von 2 bis 20 müssen im Anschluss noch mit normalen Gravitationspotenzialkoeffizienten korrigiert werden, die in /2/ in Tabelle 6.1 zu finden sind und von den ursprünglichen Werten subtrahiert werden.
Die Umrechnung von ζ zu N wird nach (3-5) durch die Addition der Korrekturterme gewährleistet. C_1 und C_2 werden dabei zusammengefasst zu einem C, welches ebenfalls durch eine Entwicklung der Kugelflächenfunktionen in Abhängigkeit von Breiten- und Längengrad ermittelt werden können. Die Formel dafür lautet analog zu (3-6):

$$C = \sum_{n=2}^{N_{max}} \sum_{m=0}^{n} (\overline{CC}_{nm} \cos m\lambda + \overline{CS}_{nm} \sin m\lambda) \bar{P}_{nm}(\sin \varphi') \tag{3-8}$$

Die Koeffizienten $\overline{CC}_{nm}$ und $\overline{CS}_{nm}$ sind ebenfalls in /20/ für alle Kombinationen von n und m tabelliert.
Im Folgenden soll ein Programm geschrieben werden, welches die Gleichung

$$N(\varphi, \lambda) = \zeta_0(\varphi, \lambda, r_E) + C(\varphi, \lambda) + \zeta_Z \tag{3-9}$$

unter Verwendung von Gleichung (3-6) bis (3-8) für einen gegebenen Ort oder ein Array an Orten P(φ, λ) auswertet. Mit dem Ergebnis kann eine durch einen GNSS-Empfänger bekannte geometrische Höhe mit einem verhältnismäßig kleinen Fehler in eine orthometrische Höhe relativ zum Geoid umgerechnet werden.

3.4 Durchführung der Programmierung

Für das OpAiRS-Projekt existiert bereits eine Python-Bibliothek mit unterschiedlichen Klassen. Darin ist unter anderem auch das WGS84-Ellipsoid mit seinen festgelegten Werten wie den Halbachsen und der Exzentrizität beschrieben.
Für das Geoid bzw. die Berechnung der Geoidundulation wurde eine neue Klasse angelegt, die als Parameter unter anderem die zehn Korrekturkoeffizienten der $\bar{C}_{nm}$-Werte, γ_e und γ_p sowie GM enthält. Als Klassenmethoden werden in ihr die normale Gravitation am Punkt P ($\gamma(\varphi)$), die Höhenanomalie und auch die Geoidundulation selbst berechnet. Weitere statische

Methoden existieren zum Einlesen der Koeffizienten für die Kugelflächenfunktionen bei der Berechnung der Höhenanomalie und der ζ-zu-N-Korrektur.
Für die Ausführung der Kugelflächenfunktion wurde eine zweite Klasse angelegt, SphericalHarmonicsExpansion (von engl. spherical harmonics = Kugelflächenfunktionen), der als Klassenvariablen die entsprechenden C- und S-Werte übergeben werden. In der Geoidklasse werden zwei verschiedene Instanzen der SphericalHarmonicsExpansion angelegt, jeweils eine mit den Koeffizienten für die Höhenanomalie und eine für den Korrekturterm.

```
from opairs.nav.geoid import Geoid
import numpy as np



coeff_path = "Path to file with zeta coefficients"
corr_path = "Path to file with correction coefficients"
lat, lon = np.array()

geoid = Geoid.from_file(coeff_path, corr_path)
N = geoid.geoid_height(lat, lon)

print(N)
```

Abb. 3.2: Main-Programm der Geoid-Berechnung

Für die eigentliche Berechnung der Geoidundulation muss in einem main-Programm zunächst ein Objekt der Klasse Geoid angelegt werden. Die statische Methode from_file liest die Koeffizienten aus den als Pfad übergebenen Dateien ein. Als zweites kann in der so erstellten Instanz der Geoid-Klasse die geoid_height-Methode ausgeführt werden. Ihr werden Breitengrad (lat) und Längengrad (lon) als NumPy-Arrays übergeben. Die Verwendung von NumPy (/21/), einer Bibliothek in Python, die auf die Arbeit und Rechnungen mit Vektoren, Matrizen und mehrdimensionalen Arrays ausgelegt ist, hat den großen Vorteil, dass eine größere Zahl an Punkten auf der Erdoberfläche gleichzeitig berechnet werden können, während die Rechenzeit trotzdem beherrschbar bleibt.
Die Methode geoid_height gibt ein NumPy-Array zurück, welches die Geoidundulationen enthält. Intern berechnet es diese nach Formel (3-6) bis (3-9).

3.4.1 Anpassung an das verwendete Berechnungsmodul SciPy

Der komplizierte Teil der Implementierung der Berechnung war die Reihenentwicklung der Kugelflächenfunktionen. Als Input der eval-Methode werden die NumPy-Arrays der Breiten- und Längengrade sowie der Faktor x (vgl. Gl. (3-10)) gegeben. Berechnet wird nach folgendem Muster, sodass damit mit dem entsprechenden Einsatz von x in die entsprechende Instanz der Klasse mit den korrekten Werten für C und S sowohl Gleichung (3-6) (ohne den Vorfaktor außerhalb der eckigen Klammern) als auch Gleichung (3-8) gelöst werden können:

$$Sph.harm. = \sum_{n=2}^{N_{max}} x^n \sum_{m=0}^{n} (C \cos m\lambda + S \sin m\lambda) \bar{P}_{nm}(\sin \varphi') \tag{3-10}$$

Da es sich bei $\bar{P}_{nm}$ um die vollständig assoziierte Legendrefunktion handelt, ist die Berechnung verhältnismäßig kompliziert. Die assoziierten Legendrepolynome sind nicht immer stabil berechenbar. Insbesondere bei hohen n und m müssen teilweise mathematisch geschickte Umwege genommen werden, um numerisch stabile Ergebnisse zu erhalten.
Die Polynome sind definiert als:

$$\bar{P}_{nm}(\sin \varphi') = \left[\frac{(n-m)!\,(2n+1)k}{(n+m)!}\right]^{\frac{1}{2}} P_{nm}(\sin \varphi') \tag{3-11}$$

Für ihre Berechnung wurde die Python-Bibliothek SciPy importiert, um sie mit dem Modul scipy.special.sph_harm zu lösen. Leider ist die Notation der Legendre-Funktionen nicht vollständig normiert. Laut API-Referenz /22/ berechnet das Programm folgendes:

$$Y_n^m(\theta, \varphi) = \sqrt{\frac{(2n+1)\,(n-m)!}{(4\pi)\,(n+m)!}}\, e^{im\theta} P_n^m(\cos(\varphi)) \tag{3-12}$$

Die Parameter der Funktion sind m, n, θ und φ. θ entspricht dabei λ in (3-10). Es fällt auf den ersten Blick auf, dass sich das Innere der Summe über m aus (3-10) und (3-11) von (3-12) unterscheiden. Tatsächlich ist der Unterschied allerdings gar nicht so groß. (3-12) berechnet das Ergebnis als komplexe Zahl, (3-10) ist einfach die Summe aus Real- und Imaginärteil der komplexen Exponentialfunktion. Die wesentlichen Unterschiede finden sich an zwei Stellen: dem Polarwinkel und dem Vorfaktor.
Für die Berechnung von (3-11) mit SciPy muss das gesamte Ergebnis vom Vorfaktor in (3-12) aus angepasst werden. Dies geschieht über eine Multiplikation mit

$$a = \sqrt{4\pi * k} \qquad \text{mit} \qquad k = \begin{cases} 1 & bei\ m = 0 \\ 2 & bei\ m \neq 0 \end{cases}. \tag{3-13}$$

Die Winkelberechnung ist insgesamt etwas komplizierter. Drei Dinge müssen dabei beachtet werden: der Definitionsbereich des Polarwinkels in der Geographie und Mathematik, die Notation der assoziierten Legendre-Funktion und die unterschiedliche Verwendung von Sinus und Kosinus in beiden Gleichungen.
Bei der Berechnung von φ, dem Polarwinkel, aus dem Breitengrad, ist zunächst eine Anpassung des Definitionsbereichs nötig, von [-90°, 90°] auf [0, π]. Während der Breitengrad vom Äquator aus gemessen wird, wird der Polarwinkel von der senkrechten Achse (also von Pol zu Pol) abgetragen. Dies erreicht man mit φ = π/2 – lat (mit lat als dem Breitengrad, bereits im Bogenmaß). Diese Anpassung erklärt auch den Unterschied von Sinus und Kosinus, der damit kompensiert ist, da sin(lat) und cos(φ) dadurch, dass die Winkel addiert immer π/2 (90°) ergeben, stets identisch sind.

Zu beachten ist schließlich noch die Cordon-Shortley-Phase. Sie ist definiert als $(-1)^m$ und wird von der SciPy-Bibliothek mitgerechnet (/22/), im WGS-Standard allerdings nicht (/2, S. 5-3/). Praktischerweise gilt:

$$(-1)^m = e^{\pm i\pi m}. \tag{3-14}$$

Es kann also, zur Beachtung der Cordon-Shortley-Phase auch φ um 180° (= π) verschoben werden. Zusammenfassend gilt für den Polarwinkel also (mit lat als Breitengrad im Bogenmaß):

$$\varphi = \pi - \left(lat + \frac{\pi}{2}\right) = \frac{\pi}{2} - lat. \tag{3-15}$$

Mit den angepassten Winkeln kann jetzt über eine For-Schleife für alle eingegebenen Orte jeweils zunächst die Summe über m mit der SciPy-Funktion berechnet, dann in Real- und Imaginärteil getrennt und mit den Koeffizienten multipliziert und anschließend über alle n, ggf. mit dem Faktor x aufsummiert werden.

Das einzige Problem, das sich stellt, ist, dass die Funktion scipy.special.sph_harm bei hohen Ordnungen NaN-Werte („not a number“, ungültige Werte) zurückgibt. Dies ist ein bekanntes Problem (vgl. /23/) und darauf zurück zu führen, dass bei Graden ab 86 ein Over- oder Underflow auftritt, also ein Rundungsfehler, der geschieht, wenn der Zahlenraum der Variablen überschritten wird.

Eine lineare Skalierung wäre eine Option, dies zu verhindern, wir haben uns allerdings dagegen entschieden, vor allem auch, weil es zwar zu Ergebnissen führen würde, diese aber vermutlich durch dennoch auftretende Rundungsfehler ungenau würden. Das Ziel des Projektes war außerdem unter anderem, ein Berechnungsprogramm zu haben, welches zügig rechnet und wenig Speicherplatz benötigt. Um bis zum 2190. bzw. zumindest 2160. Grad zu rechnen, würde das Programm erstens mehrere Minuten benötigen (allein schon zum Einlesen der Koeffizienten), außerdem müssten die Dateien mit allen Koeffizienten bereitliegen. Bei 2060 n‘s mit jeweils n m‘s, käme man für beide benötigte Koeffizientenpaare auf jeweils Dateien mit über 2,3 Millionen Zeilen (einfach zu berechnen über die Gauss’sche Summenformel). Bei einer Reduzierung auf 85 schrumpft die Länge auf 3652, was sich stark positiv auf benötigten Speicherplatz für die Koeffizientendateien und Rechenzeit beim Einlesen und Ausführen der Reihenentwicklung auswirkt.

Bei der Auswertung der Ergebnisse wurde zudem festgestellt, dass die so erzielten Ergebnisse zwar noch von den exakten Werten abweichen, allerdings in einer Größenordnung, die für den beabsichtigten Einsatzzweck von OpAiRS vernachlässigbar sind, eine genauere Analyse erfolgt in Abschnitt 3.5.2.

3.4.3 Verwendung von Unit-Tests zur Kontrolle und Korrektur der Ergebnisse

Zum Testen der Ergebnisse wurde ein 30x30°-Netz angelegt, mit Breitengrad-Werten von -89° bis 89°. (An den Polen bei ±90° ist keine Berechnung möglich, weil dort kein eindeutiger Längengrad-Wert existiert.) Für die 84 entstehenden Punkte wurden die Geoidundulationen mit einem vorhandenen Online-Rechner der NGA (National Geospacial-Intelligence Agency, /17/) ermittelt, um später Vergleichswerte zu haben und die Berechnung überprüfen zu können.

Zur Eingrenzung von Fehlerquellen wurden für die einzelnen Methoden der Geoid- und SphericalHarmonicsExpansion-Klassen Unit-Tests angelegt, mit denen unter anderem überprüft wurde, ob eine Instanz der Geoid-Klasse korrekt angelegt wird, ob der Radius richtig berechnet wird (muss zwischen den Werten der großen und kleinen Halbachse des Ellipsoids liegen), wie auch die normale Gravitation (Werte zwischen der normalen Gravitation an den Polen und am Äquator). Des Weiteren, ob die Koeffizienten korrekt eingelesen wurden.

Um die richtige Funktion der Entwicklung der Kugelflächenfunktionen zu überprüfen, wurde auf die zehn Korrekturkoeffizienten für die Berechnung der Höhenanomalie zurückgegriffen. Dadurch dass es sich dabei um die Koeffizienten fürs normale Gravitationspotenzial handelt, müsste eine Berechnung des Gravitationspotenzials V mit der Formel (5-3) aus /2/,

$$\mathrm{V} = \frac{GM}{r_E}\left[1 + \sum_{n=2}^{N_{max}} \left(\frac{a}{r_E}\right)^n \sum_{m=0}^{n} (\bar{C}_{nm}\cos m\lambda + \bar{S}_{nm}\sin m\lambda)\bar{P}_{nm}(\sin\varphi')\right] \qquad (3\text{-}15)$$

mit den zehn Koeffizienten in C und allen anderen C- sowie allen S-Koeffizienten = 0, für jeden Ort auf dem Planeten den (annähernd) gleichen Wert ergeben.

Nach Prüfung aller Einzelfunktionen wurde am Ende die Berechnung von Höhenanomalie und Geoidundulation geprüft und die Werte mit den online berechneten verglichen.

3.5 Auswertung der Ergebnisse

3.5.1 Ausprägung der Geoidundulation und ihres Fehlers

Die Geoidundulation, also die Abweichung des Geoiden vom Referenzellipsoiden, bewegt sich, wie bereits in Kapitel 3.2 beschrieben, ungefähr im Bereich von ±100m. Abbildung 3.3 zeigt sie im Farbverlauf, berechnet über ein 1x1-Grad-Netz und dazwischen interpoliert, mit einer Reihenentwicklung der Kugelflächenfunktionen bis Ordnung 85, also so weit, wie es dem Programm in seiner aktuellen Form möglich ist. Eine Abfrage von Maximum und Minimum im ausgegebenen Array liefert 85,18 m und -105,86 m als exakte Werte. Da das Programm über das Netz mit einer Stichprobe von 64800 Werten rechnet und 1° am Äquator über 111 km bedeuten kann, ist es durchaus möglich, dass vereinzelt sogar noch höhere oder niedrigere

Werte auftreten. Diese sollten allerdings nicht mehr wesentlich von den oben genannten Extrema abweichen.

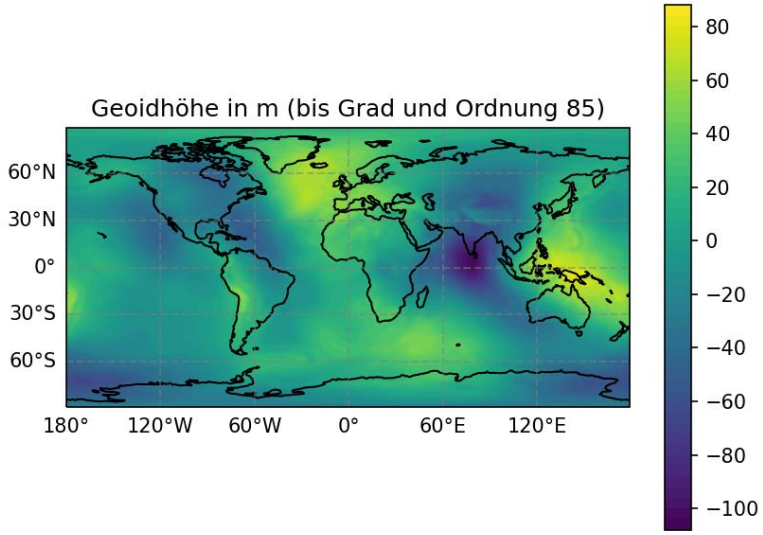


Abb. 3.3: Geoidundulation, berechnet über ein 1x1°-Netz über den Planeten mit einer Reihenentwicklung bis Grad und Ordnung 85 (Höhen im Farbverlauf in m, Karte erstellt mit Jupyter Notebook)

Nach Gleichung (3-1) bedeuten negative Werte in Abb. 3.3, dass das Geoid eine „Ausbeulung" relativ zum Ellipsoid hat, während positive Werte eine „Delle" beschreiben. Dies ist auch auf der Karte deutlich sichtbar. Intensiv gelbe Töne sind zum Beispiel an der asiatischen Pazifikküste sichtbar, wo sich der Marianengraben und weitere tiefe Schluchten im Meeresboden befinden. Daraus ist dort folglich eine geringere Masse zwischen Oberfläche und Erdzentrum, dementsprechend eine größere Nähe der Äquipotentialfläche (des Geoids) zum Erdmassezentrum, was eine besonders große „Delle" im Geoid bedeutet. Das Umgekehrte kann man im Bereich des Himalayas beobachten, wo tief violette Töne darauf hindeuten, dass das Geoid an dieser Stelle eine hohe Entfernung zum Ellipsoid in Richtung der Atmosphäre hat.

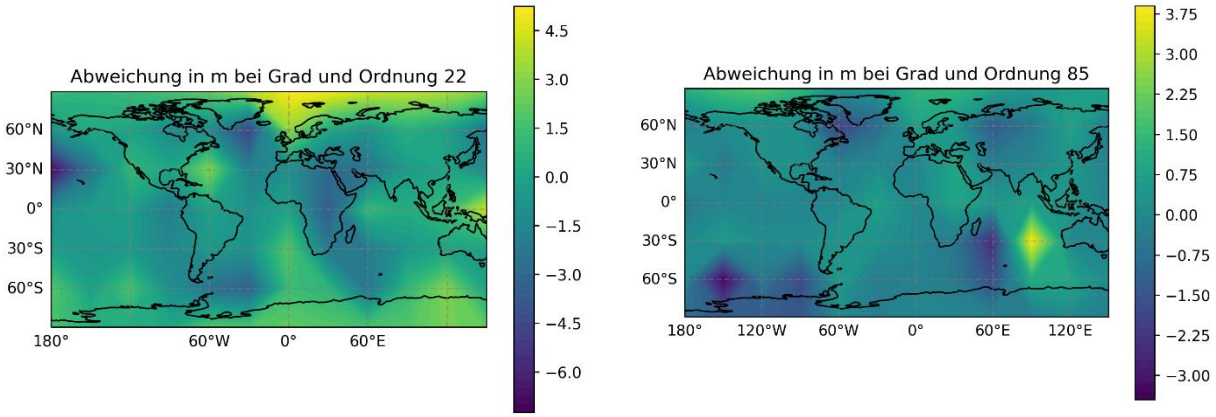
Tabelle 3: Ergebnisse der Geoidundulationsberechnung, beispielhaft für die linke Hemisphäre der Nordhalbkugel, enthält für jede Kombination aus Höhen- und Breitengrad jeweils die exakte Geiodhöhe nach /17/, die berechnete bis Ordnung 85 und die Abweichung (alle Angaben in m)

	-180°	**-150°**	**-120°**	**-90°**	**-60°**	**-30°**	**0°**
89°	13,32	14,55	15,51	16,71	15,78	16,48	16,18
	13,15	13,83	14,85	15,82	16,44	16,72	16,78
	0,17	*0,72*	*0,66*	*0,89*	*-0,66*	*-0,24*	*-0,6*
60°	0,93	14,39	-16,09	-46,78	5,03	63,9	47,88
	1,48	11,63	-16,11	-47,96	6,07	64,13	47,93
	-0,55	*2,76*	*0,02*	*1,18*	*-1,04*	*-0,23*	*-0,05*
30°	-7,20	-15,88	-42,46	-27,44	-40,63	30,22	34,4
	-6,69	-15,08	-42,05	-26,93	-39,9	30,97	34,62
	-0,51	*-0,8*	*-0,41*	*-0,51*	*-0,73*	*-0,75*	*-0,22*
0°	21,28	12,91	-22,98	-4,06	-13,36	2,45	17,23
	21,13	13,3	-22,55	-3,69	-12,94	3,54	17,53
	0,15	*-0,39*	*-0,43*	*-0,37*	*-0,42*	*-1,09*	*-0,30*

Da allerdings auch Gesteinsdichte und Breitengrad eine wesentliche Rolle spielt, ist der Verlauf des Geoids nur ähnlich, jedoch nicht gleich mit den Höhenunterschieden auf der Erdoberfläche, wie sie z. B. auf physischen Karten zu sehen sind.
Tabelle 3 zeigt beispielhaft für die 28 Werte eines 30x30°-Netzes auf der linken Hemisphäre der Nordhalbkugel, wie die berechnete Geoidundulation (mittlere Zeile jeder Zelle, hellgrau) von den exakten, nach /16/ berechneten Werten (obere Zeile, weiß) abweicht. Die Abweichung liegt zumeist bei unter einem Meter, auch wenn vereinzelt größere Werte auftreten, zum Beispiel bei 600°N/-150°W 2,76 m.

3.5.2 Einfluss der Ordnung der Reihenentwicklungen auf die Genauigkeit

Betrachtet man Gleichung (3-6), wird schnell deutlich, dass der wesentliche Faktor, um exakte Ergebnisse zu erhalten, in der Entwicklung der Kugelflächenfunktionen liegt, die laut WGS-Standard bis Ordnung 2190 geführt werden sollen. Tatsächlich ist diese Entwicklung eine der wesentlichen Verbesserungen des EGM2008-Modells gegenüber der Vorgängerversion, EGM96, die lediglich Koeffizienten bis zur Ordnung 360 enthält /2, S. 6-2/.
Wie in Abschnitt 3.4.1 beschrieben, ist dem hier entwickelten Programm aufgrund eines overflows im verwendeten Berechnungsmodul SciPy nur die Berechnung bis zur Ordnung 85 möglich. Um herauszufinden, wie viel Einfluss die Entwicklung auf die Genauigkeit hat, habe ich sie einmal nur bis zur Ordnung 22 durchgeführt und die Berechnungen im 30x30°-Gitter in ihrer Abweichung vom exakten Wert nach /16/ mit denen einer Entwicklung bis zur Ordnung 85 verglichen. Verdeutlicht werden die Ergebnisse in Abb. 3.4.
Der Unterschied ist deutlich sichtbar. Während das Ergebnis bei einer Entwicklung bis zur Ordnung 22 nur bis auf fünf bis sechs Meter genau ist (Mittelwert 1,687 m, Standardabweichung 1,434 m), sinkt die Abweichung bei einer Entwicklung bis zur Ordnung 85 auf drei bis vier Meter (Mittelwert 0,658 m, Standardabweichung 0,724 m).



(a) *(b)*

Abb. 3.4: Vergleich von Abweichung der berechneten Geoidundulation vom exakten Wert bei Reihenentwicklung bis a) Grad und Ordnung 22 und b) Grad und Ordnung 85 (Abweichungen im Farbverlauf in m, Karten erstellt mit Jupyter Notebook)

Dieser Entwicklung folgend könnten sicherlich mit einer Modifizierung des Programms bei Weiterentwicklung bis höhere Grade und Ordnungen noch deutlich genauere Ergebnisse erreicht werden.
Für die Anwendung im Rahmen des Nutzerservice OpAiRS ist eine Genauigkeit von ±5 m ausreichend. Wie in Abschnitt 2.4.2 beschrieben, überwiegen die Vorteile geringer Rechenzeit und reduziertem Speicherplatz für die Koeffizienten, so dass eine mittlere Abweichung von 66 cm und eine maximale von 4,03 m akzeptabel sind.

3.5.3 Einfluss des Korrekturterms C und der Null-Grad-Höhenanomalie ζ_Z

Ein Interessanter Aspekt ist der Einfluss des Korrekturterms C und der Null-Grad-Höhenanomalie ζ_Z. Wie bereits in Abschnitt 3.3 beschrieben, berücksichtigt der Korrekturterm sowohl die Abweichung der Geoidundulation ζ von der Geoidundulation N, als auch die Verwendung des Ellipsoidradius‘ an Stelle des realen Abstands eines Punktes auf der Topographie vom Erdmassezentrum bei gegebenem Breitengrad (vgl. Gl. (3-3), (3-5) und (3-9)).
Die Verwendung der unterschiedlichen Radien macht grob geschätzt einen Unterschied von etwa ±10 km (mit dem Mount Everest als höchstem Punkt bei 8848 m über und dem Marianengraben als tiefstem bei 11.000 m unter dem Meeresspiegel). Bezogen auf einen mittleren Erdradius von 6370 km sorgt diese Abweichung für einen Fehler von ca. 0,16 %.
Der Unterschied zwischen der Geoidundulation ζ (berechnet mit dem korrekten Radius, nicht mit dem vereinfachten des Ellipsoiden) und der Geoidundulation N liegt laut /19/ bei maximal 3,4 m, zumeist deutlich darunter, im Zentimeterbereich.
Die Null-Grad-Höhenanomalie hat, wie bereits in Abschnitt 3.3 erwähnt, den festen Wert von -0,41m.
All diese Abweichungen zusammen sollten in der Theorie dementsprechend keinen allzu großen Fehler hervorrufen. Es stellt sich also die Frage, ob die Berechnung des Korrekturterms, die Rechenzeit und Speicherplatz für eine weitere Datei mit Koeffizienten kostet, nicht überflüssig ist und welchen Einfluss der Term überhaupt auf die Genauigkeit des Ergebnisses hat.

Tabelle 4: Auswirkungen von Korrekturterm und Null-Grad-Höhenanomalie auf die durchschnittliche sowie maximale Abweichung der Geoidundulation vom erwarteten Wert (absolut)

	Nur Höhenanomalie	**Mit Korrekturterm**	**Mit Korrekturterm und Null-Grad-Höhenanomalie**
Durchschnitt [m], absolut	0,768	0,795	0,660
Maximaler Wert [m], absolut	4,028	4,302	3,892

Für die Werte in Tabelle 4 wurde die Differenz der Geoidundulationen im 30x30°-Netz von den mit /17/ berechneten Orientierungswerten berechnet. Die Tabelle zeigt Durchschnitt und Maximum der Absolutwerte, jeweils für die reine Berechnung der Höhenanomalie gänzlich ohne Korrektur, für die mit dem Korrekturterm aus (3-8) korrigierten Werte und für die Werte, bei denen sowohl die Korrekturterme als auch die Null-Grad-Höhenanomalie berücksichtigt wurden. Die Auswirkungen sind geringfügig, aber dennoch vorhanden. Interessant ist dabei vor allem, dass eine alleinige Addition des Korrekturterms zunächst zu einer Verschlechterung der Genauigkeit führt. Die Null-Grad-Höhenanomalie scheint diese Abweichung zu korrigieren, sodass das Ergebnis am Ende durchschnittlich und auch maximal weniger stark vom korrekten Wert abweicht als die reine Höhenanomalie.

Ob die kleinen Unterschiede von wenigen Zentimetern den zusätzlichen Rechen- und Speicheraufwand des Korrekturterms rechtfertigen, bleibt dem Anwender überlassen. Es steht jedoch außer Frage, dass eine Verbesserung eintritt.

4 Zusammenfassung

Im ersten Teil meines Praktikums hatte ich die Möglichkeit, mich mit hardwarenaher Programmierung zu beschäftigen. Bei der Optimierung der Funktionsweise eines Messboots sollten verschiedene Datenströme von einem GPS-Empfänger und einem Echolotgerät eingelesen, verknüpft, verarbeitet und weitergeleitet werden. Dazu musste ich mich in die Open-Source-Codes der Messgeräte einarbeiten und ein Programmierungskonzept entwickeln, das die Anforderungen der Aufgabenstellung erfüllt.
Das Ergebnis ist ein Python-Code, der auf einem auf dem Messboot installierten Raspberry Pi läuft. Er liest die Datenströme von Echolot und GPS-Gerät ein, entweder mit einer seriellen Schnittstelle oder über UDP-Verbindungen. Im Programm selbst werden die Datenströme synchronisiert, jedem hereinkommenden Set Echolotdaten werden die gleichzeitig aufgenommenen GPS-Daten zugeordnet, sodass nach Abschluss der Messungen eine Karte aus den Echolotdaten erstellt werden kann. Das Programm leitet die Echolotdaten über UDP unverändert weiter und schreibt die verknüpften Daten in eine Datei. Eine besondere Herausforderung war hierbei, die Formatierung der Daten so anzupassen (zum Beispiel Nachrichten mit Headern zu bestücken), dass sie hinterher noch von bestehender Software ausgelesen und verarbeitet werden können.
Im zweiten Teil des Praktikums ging es um die Umrechnung von Höhen über dem Referenzellipsoid in Höhen über dem Geoid, einer dem mittleren Meeresspiegel möglichst nahen Äquipotentialfläche. Dazu mussten zunächst die mathematischen Grundlagen erarbeitet und dann eine Umsetzung dieser Berechnung in Python entwickelt werden. Die Aufgabe hatte zudem eine zentrale Ausrichtung auf objektorientierte Programmierung. Hier war eine große Herausforderung, neben der Recherche aller relevanten Daten und Kennwerte, die Implementierung der Entwicklung von Kugelflächenfunktionen. Umgesetzt wurde das mit dem Python-Modul SciPy, welches sich allerdings in der Notation teilweise stark von der des WGS-Standards unterscheidet, auf dessen Grundlage die Berechnung durchgeführt wird.
Das Programm wurde anhand von Referenzdaten validiert, welche mit dem Online-Rechner der NGA in /17/ berechnet wurden. Im Anschluss konnte eine Auswertung bezüglich der Abweichung der Werte von der Realität und des Einflusses von Korrekturtermen sowie Ordnung der Reihenentwicklung durchgeführt werden. Diese beiden Schritte waren für mich eine gute Möglichkeit, mich mit der Verwendung von Unit-Tests und der Software Jupyter Notebook vertraut zu machen.

5 Literaturverzeichnis

/1/ Earth Observation Center – IMF: Abteilung Experimentelle Verfahren
https://www.dlr.de/eoc/desktopdefault.aspx/tabid-5294/8937_read-16241/
03.05.2023

/2/ NGA Standardization Document: World Geodetic System 1984
Version 1.0.0, 2014

/3/ Ping Viewer Documentation
https://docs.bluerobotics.com/ping-viewer/
12.07.2023

/4/ GitHub: ArduPilot / pymavlink
https://github.com/ArduPilot/pymavlink
03.05.2023

/5/ GitHub: bluerobotics / ping-python
https://github.com/bluerobotics/ping-python
03.05.2023

/6/ GitHub: ping-viewer/examples
https://github.com/bluerobotics/ping-viewer/tree/master/examples
03.05.2023

/7/ ping-python – Ping 1D Class Reference: Public Member Functions
https://docs.bluerobotics.com/ping-python/classbrping_1_1ping1d_1_1Ping1D.html
04.05.2023

/8/ Ping Protocol – Message Format
https://docs.bluerobotics.com/ping-protocol/
04.05.2023

/9/ Ping-Proxy-Code
https://github.com/bluerobotics/ping-python/blob/master/tools/pingproxy.py
04.05.2023

/10/ Mavlink Dialect: ArduPilotMega
https://mavlink.io/en/messages/ardupilotmega.html
04.05.2023

/11/ Earth Observation Center – OpAiRS
https://www.dlr.de/eoc/desktopdefault.aspx/tabid-5328/19980_read-46622/
07.06.2023

/12/ Bundesverband Geothermie – Hyperspectral-Imaging (Remote Sensing)
https://www.geothermie.de/bibliothek/lexikon-der-geothermie/h/hyperspectral-imaging-remote-sensing.html
07.06.2023

/13/ EnMAP – Wildfire derivation and analysis using EnMAP
https://www.enmap.org/news/2023-06-02
07.06.2023

/14/ Mathepedia: Rotationsellipsoid
https://mathepedia.de/Rotationsellipsoid.html
05.06.2023

/15/ Wikipedia: Äquipotentialfläche
https://de.wikipedia.org/wiki/%C3%84quipotentialfl%C3%A4che
05.06.2023

/16/ mapitGIS: Geoid Height
https://mapitgis.com/geoid-height-extension/
06.06.2023

/17/ Office of Geomatics: NGA EGM96 and EGM08 Geoid Calculators
https://earth-info.nga.mil/index.php?dir=wgs84&action=egm96-geoid-calc
21.06.2023

/18/ Barthelmes, F.
Definition of Functionals of the Geopotential and Their Calculation from Spherical Harmonic Models
Helmholz-Zentrum Potsdam, Deutsches Geoforschungszentrum
Scientific Technical Report SRE09/02, überarbeitete Fassung, Januar 2013

/19/ Rapp, R. H.
Use of potential coefficient models for geoid undulation determinations using a spherical harmonic representation of the height anomaly/geoid undulation difference
The Ohio State University, 1996

/20/ Pavlis, N.K., S.A. Holmes, S.C. Kenyon, and J.K. Factor
An Earth Gravitational Model to Degree 2160: EGM2008, presented at the 2008 General Assembly of the European Geosciences Union,
Wien, Österreich, 13.-18. April 2008.

/21/ Harris, C.R., Millman, K.J., van der Walt, S.J. et al.
Array programming with NumPy
Nature 585, 357–362 (2020)

/22/ Scipy.special.sph_harm API reference
https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.sph_harm.html
21.06.2023

/23/ Github: Overflow in sph_harm for high degrees #7778
https://github.com/scipy/scipy/issues/7778
21.06.2023