

MASTERARBEIT | MASTER'S THESIS

Titel | Title

LiDAR Data Acquisition and Compression for Space Applications

verfasst von | submitted by

Christopher Granabetter BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 861

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Astronomie

Betreut von | Supervisor:

Univ.-Prof. Mag. Dr. Franz Kerschbaum

Abstract

LiDAR (Light Detection and Ranging) ist eine aktive Fernerkundungstechnik, die Laserpulse nutzt, um Entfernungen und Reflektivität verschiedener Oberflächen über Distanzen zu messen. Da diese Technik in den letzten Jahren immer ausgereifter geworden ist, gewinnt diese auch zunehmend an Bedeutung für die Weltraumforschung.

In den letzten Jahren wurde in zahlreichen Weltraummissionen LiDAR-Sensorik verbaut. Die wohl in der Öffentlichkeit am bekannteste Mission ist der Ingenuity Mars Helikopter. Anhand der an Bord befindlichen LiDAR-Technik kann während eines Fluges die Höhe über Grund bestimmt werden. Erdgebundene Systeme sind weitaus komplexer und umfangreicher in der Datenerfassung durch die Mehrfachzielerkennung, die es ermöglicht, Gelände, Baumstämme und Baumkronen gleichzeitig zu detektieren. Des Weiteren können Gewässer präzise vermessen werden, indem die Wasseroberfläche und der Gewässergrund gleichzeitig aufgezeichnet werden. Die Methode der Mehrfachzielerkennung, auch als vollständige Wellenformdetektion bekannt, ist derzeit an Bord der ESA-Raumsonde JUICE (Jupiter Icy Moons Explorer) mit dem Instrument GALA (Ganymede Laser Altimeter) im Einsatz. Eine kleinere Version von GALA, BELA (Bepi-Colombo Laser Altimeter), befindet sich derzeit an Bord der Raumsonde Bepi-Colombo auf dem Weg zum Merkur.

LiDAR wird auch für die Atmosphärensensorik auf der Erde eingesetzt, um detaillierte Daten über Luftströmungen und Wetterphänomene zu erfassen. Der Ansatz zur Atmosphärensensorik unterscheidet sich von den zuvor beschriebenen Methoden, da anstelle eines einzelnen Laserimpulses mehrere Laserimpulse in verschiedene Richtungen ausgesendet werden. Die Detektoren messen dabei nicht nur die Intensität der Laserreflexionen, sondern auch die Dopplerverschiebung des zurückgesendeten Signals. Diese Methode wird hauptsächlich für Erdbeobachtungen verwendet, entweder boden- oder weltraumgestützt, wie bei der ESA-Mission Aeolus. Sie könnte jedoch auch auf Missionen zu anderen Planeten wie der Venus, dem Mars, den Gasriesen des Sonnensystems sowie einigen ihrer Monde ausgeweitet werden.

Contents

1	Introduction	3
2	LiDAR Technology	4
2.1	Basic Concept	4
2.2	Terrestrial LiDAR	4
2.3	Airborne LiDAR	4
2.3.1	Bathymetry	5
2.4	Space-based LiDAR	5
2.4.1	BELA - The Bepi-Colombo Laser Altimeter	5
2.4.2	GALA - Ganymede Laser Altimeter	8
3	LiDAR Data	9
3.1	BELA Data	9
3.2	BELA Data Fitting	14
3.2.1	Used Libraries	14
3.2.2	Main	14
3.2.3	Reading BELA Data File	17
3.2.4	Model Functions	20
3.2.5	Data Fitting	22
3.3	LiDAR Data Generator	24
3.3.1	Used Libraries	24
3.3.2	Input Parameters and Global Variables	25
3.3.3	Main	26
3.3.4	Generator for One Waveform	28
3.3.5	Fitted Waveform File Reader	31
3.3.6	Calculate Area under Gaussian Peak	33
3.3.7	Calculate Sigma of Gaussian Peak by Amplitude and Energy	33
3.3.8	Generation of a Single Pulse	33
3.3.9	Waveform Signal Model	34
3.3.10	Creating Multiple Returns/False Positives	35
3.3.11	Merging All Single Signals	37
3.3.12	Adding Noise	38
4	Data Compression	40
4.1	Entropy of Data	40
4.2	Lossless Compression	40
4.3	Lossy Compression	41
5	Lossy Compression	42
5.1	Peak Extraction Methods	42
5.1.1	Range Window	42
5.1.2	Front2Back	42
5.1.3	Top2Bottom	42

5.1.4	Curve Discussion	43
5.2	Peak Extraction Code Description	43
5.2.1	Range Window	43
5.2.2	Front2Back	46
5.2.3	Top2Bottom	49
5.2.4	Curve Discussion	54
5.2.5	Sorting Peaks	56
5.3	Signal Fitting	58
5.4	Resulting Data Packets	60
5.4.1	Basic Packet Structure	60
6	Lossless Compression	65
6.1	Huffman Coding	65
6.2	Golomb-Rice Coding	66
6.3	Arithmetic Coding	66
7	Results	68
7.1	Peak Extraction Evaluation	68
7.1.1	Front2Back	68
7.1.2	Top2Bottom	69
7.1.3	Curve Discussion	70
7.1.4	Computing Time Comparison	72
7.2	Compression Ratios	76
7.2.1	Lossy Compression	76
7.2.2	Lossless Compression	80
7.3	Accuracy	87
8	Conclusion	89
8.1	Future Prospects	89

1. Introduction

LiDAR (Light Detection and Ranging) is an active remote sensing technique, that utilizes laser light to measure distances and reflectance of different surfaces. Since LiDAR technology is becoming more and more sophisticated, the technology also becomes more relevant for space applications (Fix et al. (2019)).

In recent years many space missions were equipped with LiDAR sensors. The Ingenuity Mars Helicopter is the most publicly known spacecraft with such a system installed. Ingenuity is using its LiDAR sensor for flight altitude measurements while flying autonomously in the shallow atmosphere of Mars (Garmin (2021)). This usage of laser sensing is just the tip of the iceberg of what this technology has to offer. The standout feature of this technology is its utilization of full waveform detection. Full waveform detection makes measuring multiple laser returns with one emitted laser pulse possible. This has the benefit, that terrain measurements can be done in dense forests simultaneously with measurements of the forest stand and the canopy of the trees. Another use case for full waveform detection is the measurement of rivers, riverbeds, and lakes. This technique could be a novel approach to studying small lakes found on planetary bodies such as Titan. The method of full waveform detection is currently aboard ESA’s JUICE (Jupiter Icy Moons Explorer) spacecraft with the GALA (Ganymede Laser Altimeter) instrument (Enya et al. (2022)). A smaller version of GALA, BELA (Bepi-Colombo Laser Altimeter) is currently on its way to Mercury on board the Bepi-Colombo spacecraft (Thomas et al. (2021)).

LiDAR is also being used for atmospheric sensing on Earth. The approach to atmospheric sensing differs from the methods previously described, because instead of one single laser pulse multiple laser pulses are emitted in different directions. The detectors then measure not just the intensity of the laser returns, but also the Doppler shift of the returned signal (Cole (2019)). This method is mainly used for Earth observations either on ground or space-based, like ESA’s Aeolus mission (ESA (2022)), but it could also be expanded to planetary missions to Venus, Mars, the gas giants in the solar system as well as some of their moons.

2. LiDAR Technology

2.1 Basic Concept

The technology is based on a time of flight measurement, where the time a light pulse takes for a specific distance is recorded. The outgoing light pulse is produced by a laser source that can generate pulses on timescales on the order of nanoseconds. A receiver optic is used to detect the reflected light and since the time of the outgoing pulse is known, together with the timestamp of the received light pulse the distance to the target surface can be calculated by:

$$\text{Distance to Target} = \frac{\text{Speed of Light} \cdot \text{Time}}{2} \quad (2.1)$$

The total distance has to be halved since the light travels from the instrument to the target surface and from the target surface back to the instrument.

This simple concept allows for high-precision distance measurements. It found its way into many different applications, such as typical surveying of buildings and landscapes, along with forensics and atmospheric sensing.

2.2 Terrestrial LiDAR

The most powerful and precise instruments are found in terrestrial LiDAR systems. Terrestrial scanners achieve pulse repetition rates in the range of MHz, allowing for data acquisition of over one million points per second, while still achieving a ranging accuracy of 5 mm. The typical range of such systems is below 1 km. A data product of a terrestrial laser scanner can be seen in figure 2.1.

2.3 Airborne LiDAR

To enhance area coverage, LiDAR systems are integrated into various aircraft, including UAVs, helicopters, and airplanes. The scanning system comprises an inertial measurement unit, which



Figure 2.1: High-resolution point cloud of a terrestrial laser scan. The colors indicate the reflectivity of the individual surfaces and textures. The reflectivity is determined by the intensity and shape of the return pulse. (Skyability GmbH (2021))

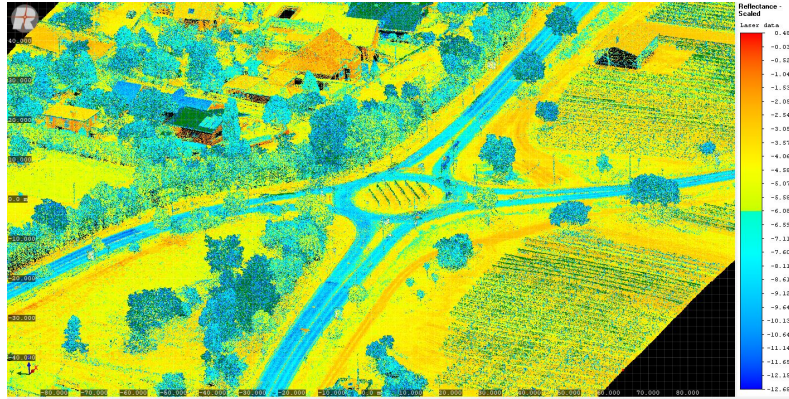


Figure 2.2: Aerial point cloud of a sub-urban area. As illustrated by the color bar on the right side of the figure, the different surfaces have different reflection properties. (Skyability GmbH (2020))

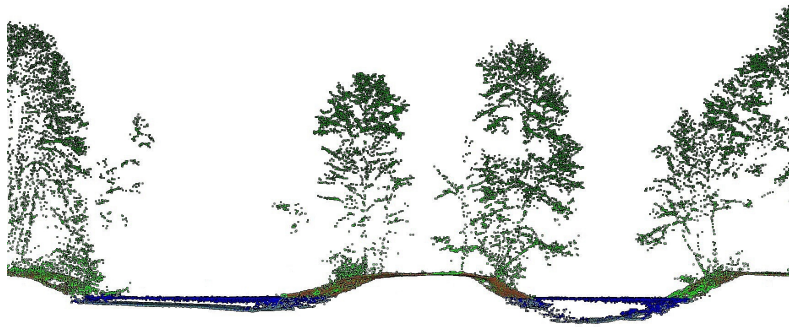


Figure 2.3: Bathymetric laser scan. Multi-target detection and a specific laser source, allow for simultaneous mapping of the forest canopy, leaves, branches, tree stems, water surface, and river bed. (Skyability GmbH (2022))

monitors the position and orientation of the laser scanner. Combined with corrected GPS coordinates, this generates a point cloud, enabling precise surveys of larger-scale terrain and urban areas. Airborne LiDAR is particularly useful for remote or hard-to-access areas such as dense forests and alpine terrain. An airborne LiDAR point cloud can be seen in figure 2.2. Typical scan rates are between 50 to 550 kHz with ranges up to 1.5 km.

2.3.1 Bathymetry

A sub-category of airborne LiDAR is the so-called bathymetry. The multi-target detection of LiDAR enables the simultaneous mapping of various targets such as forest canopy, leaves, branches, and tree stems. The specialty of bathymetry is the additional surveying of waterbodies, measuring the water surface together with a lake floor or river bed, as can be seen in figure 2.3.

2.4 Space-based LiDAR

2.4.1 BELA - The Bepi-Colombo Laser Altimeter

The Bepi-Colombo Laser Altimeter is currently on its way to Mercury on board the Bepi-Colombo spacecraft, launched on October 20, 2018, and is the main focus for the LiDAR data generator

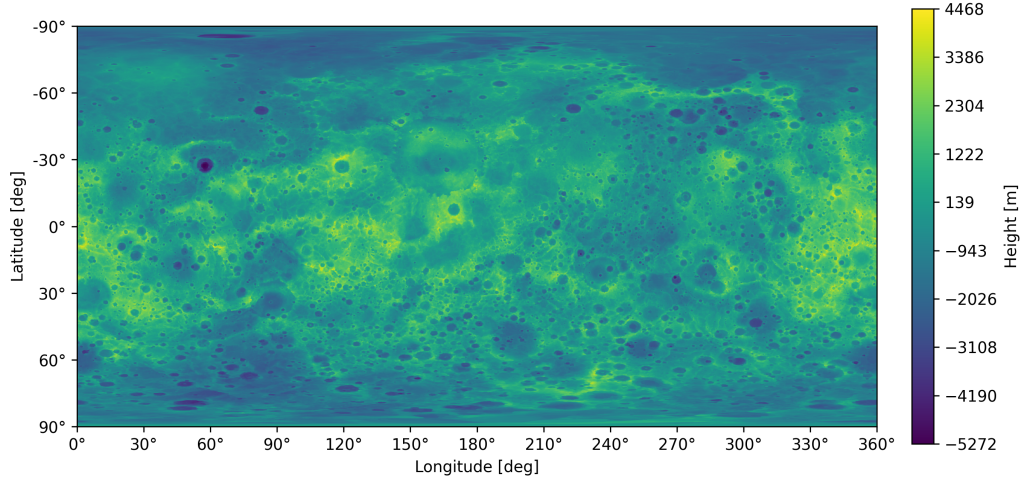


Figure 2.4: Digital elevation model (DEM) of Mercury. (USGS Astrogeology Science Center (2016))

and subsequent compression methods for this thesis.

Scientific Objectives

BELA is a European-built instrument with a Swiss-German joint project lead and smaller involvement from Spain and France. It will primarily focus on establishing precise reference surfaces for Mercury, enabling the measurement of topographic variations and radial tidal deformations relative to these surfaces. A geodetic network will be established based on prominent topographic features with accurately determined positions on the reference surfaces in location and, specifically, height. Additionally, it will assess Mercury's rotational state, including obliquity and libration amplitude. Furthermore, BELA will analyze surface roughness, albedo, and variations thereof across Mercury's terrain. A particular focus of this active measurement technique lies on the permanently shaded craters near the poles of Mercury. BELA is a vital component of a larger geodesy and geophysics package that contains radio science and stereo imaging for an in-depth investigation of Mercury's gravity field and planetary figure, as well as its interior structure, surface morphology, and geology. The instruments also have the possibility to detect tidal deformations and further derive the offset between the center of mass to the center of figure. Based on the aforementioned reference surfaces and the geodetic network, a coordinate system can be established to precisely locate any geological, physical, and chemical points of interest on the surface (Thomas et al. (2019)). The subsequent main data product of the LiDAR measurements is a topographic map of the Hermean surface in the form of a digital elevation model (DEM). The BELA instrument is similar to the laser altimeters for interplanetary flights, built by Goddard Space Flight Center, which are the MESSENGER Laser Altimeter (MLA), the Lunar Orbiter Laser Altimeter (LOLA), and the Mars Orbiter Laser Altimeter in terms of the detector being used, a flight-proven avalanche photodiode (APD).

BELA will operate at altitudes ranging from 400 km to 1500 km with accuracies just below 2 m. The wavelength of the laser source is 1064 nm with a pulse repetition frequency of 10 Hz. The end product is planned to be a more precise model of the Hermean surface compared to the DEM by Becker et al. (2016) (fig. 2.4). A DEM is a digital representation of a terrain or surface of an object represented as a raster grid. The raster consists of cells where the cell size defines the spatial resolution. In the Mercury DEM each pixel relates to 665 meters in longitude and latitude. The elevation is stored as integers or floating point numbers, depending on the height resolution

and precision. The heights are either displayed as grayscale or as false color images, as can be seen in figure 2.4.

Technical Implementation

Laser altimeters from previous missions used analog electronics to detect the incoming laser pulse by leading-edge detection and thresholding algorithms. The advantage of this approach is its robustness, but it lacks accuracy for varying signal strengths, i.e. the detection is earlier the higher the return signal. This behavior can be mitigated by multiple thresholds at the cost of added complexity. Earth-orbit satellites like GLAS aboard ICESat have the advantage of higher telemetry rates which enables the usage of data-intensive methods, such as storing and transmitting the complete digitized waveform after an initial leading-edge detection. Interplanetary missions are more restricted in terms of bandwidth and require more complex data handling and data compression. Therefore, BELA introduced a fitting algorithm after an initial digitization of the waveform. The return pulse is fitted in firmware and only the parameters of the fit are transmitted. This allows the exact determination of the return signal peak and intermittently downlinks of the digitized signal for on-ground verification and a more detailed study of the signal. The downside of working with digitized signals is that the accuracy of the time of flight measurement is dependent on the digitization rate (Thomas et al. (2021)). The dependence on the digitization rate for several sampling rates is shown in section 7.3 figure 7.17.

The team behind BELA was limited by the space-qualified analog-to-digital converters (ADC) at the time of the design. The sampling rate available, which was 40 MHz, is not fast enough to have met the desired accuracy within the system level requirements. Hence, 2 phase-shifted ADCs were combined to achieve 80 MHz sampling (Thomas et al. (2021)). Subsequent LiDAR instruments on Missions such as ESA's JUICE mission to the Jovian system, in particular Ganymede, have much higher sampling rates due to the advancements of space-qualified hardware. JUICE's Ganymede Laser Altimeter (GALA) uses 200 MHz sampling (Hussmann et al. (2019)).

Key design factors for the BELA instrument were the size of the receiver telescope, as it impacts the thermal control sub-system of the spacecraft, and the laser, specifically the power consumption of the laser system. The telescope aperture has to be kept small to minimize the stray light from the Sun from entering the spacecraft and heating the internals. For the laser source an Nd:YAG laser is used, emitting photons at a wavelength of 1064 nm instead of the frequency-doubled 532 nm wavelength, which would be more power efficient. The near-infrared wavelength was chosen, in trade-off of a lesser efficiency, due to the reddening of the Hermean surface and the high performance of the detector in this wavelength range. The Excelitas infrared-enhanced silicon avalanche photodiode yields a higher signal-to-noise ratio for the wavelength of 1064 nm. The power consumption of this type of laser is typically 20-40 times the emitted laser power. Although high laser pulse energies can be chosen, the limiting factor is the maximum dissipation that can be allowed by the spacecraft thermal control sub-system. As a result of these design factors, a receiver telescope with an aperture of 20 cm and a laser pulse energy of 50 mJ at a repetition frequency of 10 Hz was implemented (Thomas et al. (2021)).

Due to the nature of the highly sensitive detector being used, the probability of false detection (PFD) has to be considered. To reduce the PFD, the previous good detections are used to predict the next measurement. For a coarse estimation of the return pulses a range window is applied to the sampling in which a return is to be suspected. This range window is of the size of 30 km to ensure the detection of a return and can be reduced to smaller windows if the tracking recognizes small deviations in the range measurements. This procedure reduced the probability of the rangefinding algorithm to misinterpret a noise spike as a valid return and therefore decreases the PFD. Further, onboard the spacecraft, a filter can be applied to the times of received returns to reduce the PFD statistically. This led to baselining of range tracking and algorithms for the active reduction of the size of the range window within a software range tracking algorithm, based on Bayesian techniques, that was implemented within the BELA onboard computer (Thomas et al. (2021)).

An essential component of every LiDAR instrument is the timing and tracking of the outgoing and receiving light pulse. BELA is based around an oscillator that is required to be stable,

instead of focusing on stability over a large time scale. Since single measurements are on short time intervals, it is more efficient and reliable to calibrate the internal timing against a reference. The internal timing is compared to the pulse per second (PPS) signal of the spacecraft to minimize the internal drift. This results in the time of flight being referenced to the number of clock cycles per second counted by the instrument (Thomas et al. (2021)). The signal generation presented in this thesis is based on a similar approach, as the samples for the signal are generated based on the sampling rate, and the number of samples are counted. The sample count is then translated to a time in seconds, which is then used to make the time of flight computation. Naturally, a simulated dataset, as seen in this thesis, is much more consistent and repeatable than real-world data, which makes it even more important to have the capability to synchronize and calibrate the clock cycles of the instrument to accurate timestamps. With the absolute time reference being established the second essential point of a LiDAR measurement is the reference of the relative time between the outgoing and returning pulse. Here, BELA uses the same detection electronics for both signals. A small fraction of the outgoing light is passed through optical fibers and sent to the detector unit, which initiates the time of flight measurement. Therefore, any drifts of the timing of the pulse emission with respect to the pulse detection is then irrelevant (Thomas et al. (2021)).

2.4.2 GALA - Ganymede Laser Altimeter

The Ganymede Laser Altimeter is implemented into the Jupiter Icy Moons Explorer or in short JUICE, launched on April 14, 2023, and will arrive at the Jupiter system by 2030. The development of GALA is largely based on heritage from BELA since it was built by a similar consortium. Although GALA is based on BELA, several adjustments had to be made to accommodate the different environment of Jupiter compared to Mercury and the different scientific objectives.

To this day, the only topographic data of Ganymede that is available stems from stereo photogrammetry and photo-clinometry of imaging data from the NASA Galileo spacecraft. The coverage of the data sets is minimal and the large-scale topography of Ganymede is completely unknown. GALA will acquire precise topographic data with a radial accuracy of a few meters. The data will provide an improved understanding of the processes that have shaped Ganymede's surface. From the topographic data, information on the physical state of the shallow subsurface at the time of formation of these features will be gained. Not only Ganymede but also the Galilean Moons Europa and Callisto will be investigated. The scientific objectives include an exploration of the morphology and surface geology of the moons, a determination of their interior structures from a combination of shape, topography, and gravitational field data from the radio science experiment 3GM, the investigation of the surface chemistry and physical properties and their formation and evolution, especially with respect to subsurface water oceans. Further, the topography data is needed to account for the effects of topographic heights on the gravity field and to account for near-surface mass distribution anomalies above the reference figure. It will also support geological studies for the characterization and identification of the tectonic and cryovolcanic regions on the icy moons, and measure the radial tidal deformation on the 7.15-day diurnal cycle to detect and characterize a potential sub-surface ocean on Ganymede (Husmann et al. (2019)). The contribution by GALA observations to the orientation and rotational states, including libration, of the Jovian satellites is similar to the contributions going to be made by BELA for Mercury. The difference here lies in the investigation of subsurface oceans, in particular the icy shell in combination with imaging data from the instrument JANUS and Radio Science data from the 3GM package. The objectives are again similar for the characterization of the surface roughness, slopes, and albedo of the bodies.

GALA operates at the same wavelength as BELA (1064nm), but has a maximum shot frequency of 50 Hz instead of 10 Hz, and an altimetry accuracy of up to 1 m compared to 1.87 m. At nominal shot frequency of 30 Hz and at 500 km altitude GALA will sample Ganymede's surface with a spot interval of 50 meters.

3. LiDAR Data

3.1 BELA Data

The data discussed here is a test data set of the Bepi-Colombo Laser Altimeter (BELA) and is provided by Thomas Beck from the University of Bern. He was part of the team that developed the range finder module (RFM) of BELA (Thomas et al. (2021)).

The data is available in the form of an ASCII text file. There are six different types of packets present, *control*, *delay*, *cp_init*, *dp_test*, *cp_control* and *dp_pulse*. In total, there are 4998 *dp_pulse* packets, 4998 *cp_control* packets, 539 *control* packets, 105 *delay* packets, 1 *cp_init* packet, and 1 *dp_test* packet. The packets are arranged so that after a plethora of control packets, that initiate the test measurements, every 9.7 seconds seven control packets followed by a delay - control - delay - control sequence occur. In between those sequences, *cp_control* and *dp_pulse* are alternating making a total of 196 packets (98 *cp_control* and 98 *dp_pulse* packets) for one measurement sequence. The packets and their contents are listed from table 3.1 to 3.6.

The provided data is pre-processed and no additional software or data is provided for the BELA dataset. The relevant samples for the base waveforms to generate the test dataset are contained within the *dp_pulse* packet, more precisely in *packet*, *tx_pulse* and *rx_pulse* (tab. 3.5 and 3.6). The waveforms for the TX and RX packet can be seen in figure 3.1. BELA uses two ADCs to digitize the signal coming from the Avalanche Photo Diode (APD). Each ADC has a different offset (bias) and noise behavior, illustrated in figures 3.2, 3.3, and 3.4. As can be seen in the latter two figures, the noise level for both ADCs is under 3, which sets the baseline for the noise level in the LiDAR data generator.

cp_control	
Content	Type
packet_start	<class 'int'>
cp_type	<class 'int'>
rfm_control_word	<class 'int'>
rx_window_start	<class 'int'>
envelope_start	<class 'int'>
envelope_end	<class 'int'>
rx_gain_code	<class 'int'>
histogram_offset	<class 'int'>
temp_control_setpoint	<class 'int'>
peltier_setpoint	<class 'int'>
spare_word_1	<class 'int'>
hv_setpoint	<class 'int'>
test_pattern_start	<class 'int'>
test_pattern_length	<class 'int'>
spare_words_2	<class 'list'>
packet_end	<class 'int'>
check_sum	<class 'int'>

Table 3.1: Content of *cp_control*

dp_pulse	
Content	Type
expected_packet	<class 'bool'>
valid_checksum	<class 'bool'>
expected_type	<class 'bool'>
packet	<class 'dict'>

Table 3.2: Content of *dp_pulse*

dp_pulse > packet	
Content	Type
packet_start	<class 'int'>
dp_type	<class 'int'>
rfm_status_word	<class 'int'>
hk	<class 'dict'>
tx_sample_offsets	<class 'list'>
tx_pulse	<class 'dict'>
rx_sample_offsets	<class 'list'>
rx_sample_powers	<class 'list'>
rx_pulse	<class 'list'>
hist_corr_indices	<class 'list'>
hist_corr_values	<class 'list'>
hist_maximum	<class 'int'>
hist_maximum_index	<class 'int'>
snr_update_value	<class 'int'>
histogram	<class 'list'>
check_sum	<class 'int'>
packet_end	<class 'int'>

Table 3.3: Content of sub-packet *dp_pulse > packet*

dp_pulse > packet > hk	
Content	Type
tx_gain_ctrl_voltage	<class 'int'>
rx_gain_ctrl_voltage	<class 'int'>
apd_temperature	<class 'int'>
hv_monitor_voltage	<class 'int'>
rfm_temperature	<class 'int'>
peltier_current	<class 'int'>
hv_setpoint_voltage	<class 'int'>
rfm_2v5_ref_voltage	<class 'int'>
rfm_1v5_ref_voltage	<class 'int'>
hk_timestamp	<class 'int'>
regulated_peltier_setpoint	<class 'int'>
spare_word_1	<class 'int'>
pps_main_counter	<class 'int'>
pps_main_timestamp	<class 'int'>
pps_red_counter	<class 'int'>
pps_red_timestamp	<class 'int'>
rfm_version	<class 'int'>
rfm_sub_status_word	<class 'int'>
coefficient_set_id	<class 'int'>
spare_words_2	<class 'list'>
dp_timestamp	<class 'int'>

Table 3.4: Content of sub-packet *dp_pulse > packet > hk*

dp_pulse > packet > tx_pulse	
Content	Type
family_and_corr_index	<class 'int'>
spare_words	<class 'list'>
bfo_scaling	<class 'int'>
bfo	<class 'list'>
samples	<class 'list'>

Table 3.5: Content of sub-packet *dp_pulse > packet > tx_pulse*

dp_pulse > packet > rx_pulse	
4x	
Content	Type
family_and_corr_index	<class 'int'>
spare_words	<class 'list'>
bfo_scaling	<class 'int'>
bfo	<class 'list'>
samples	<class 'list'>

Table 3.6: Content of sub-packet *dp_pulse > packet > rx_pulse*

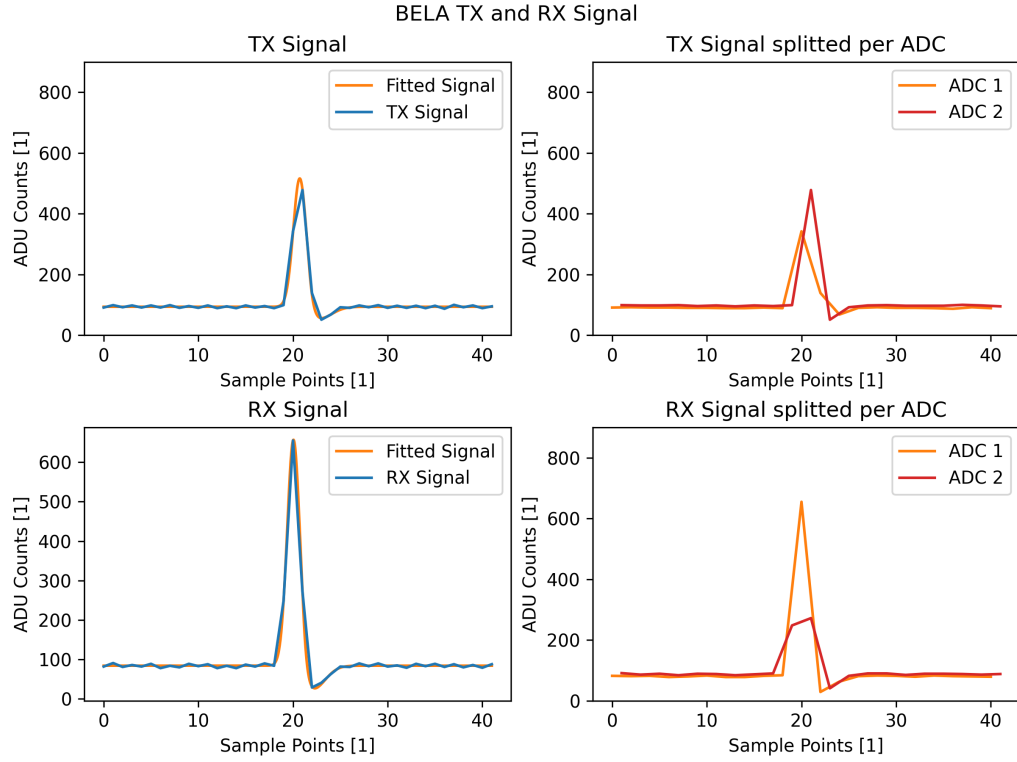


Figure 3.1: BELA data fitted and split for each ADC

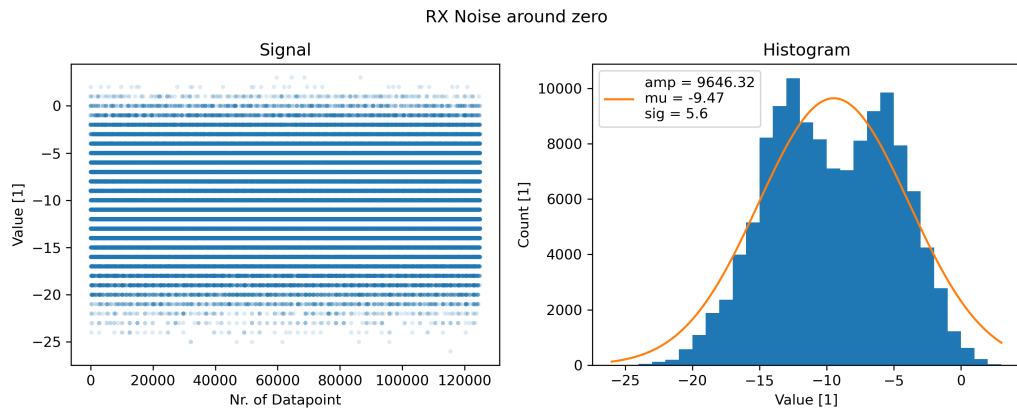


Figure 3.2: RX noise around zero from the BELA dataset. The fit is only to visualize the difference to a single Gaussian noise level.

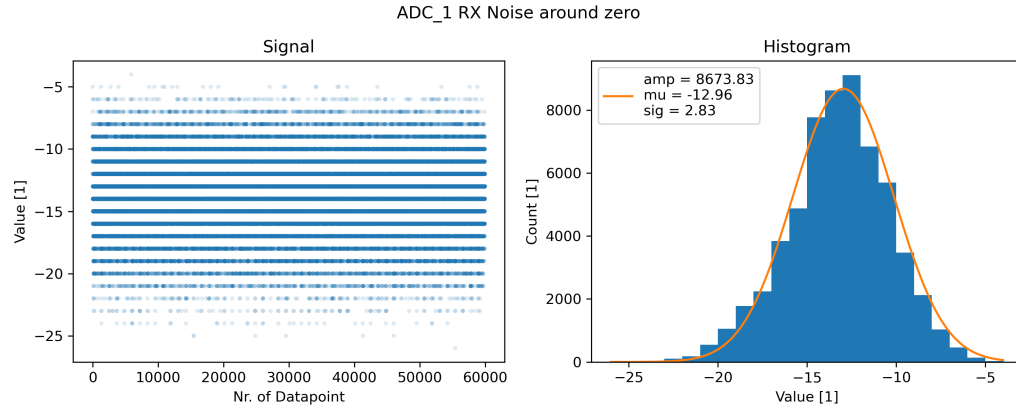


Figure 3.3: RX noise around zero of ACC 1 from the BELA dataset.

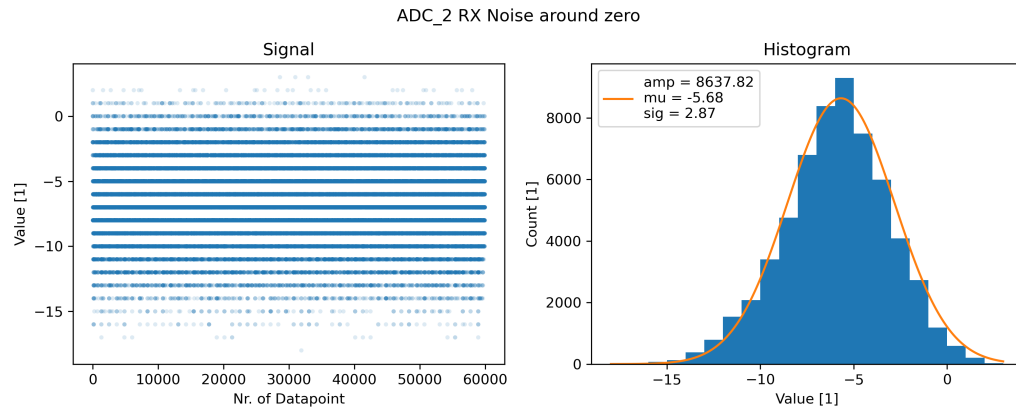


Figure 3.4: RX noise around zero of ACC 2 from the BELA dataset.

3.2 BELA Data Fitting

This section describes the usage of the provided BELA test data for the subsequent generation of new individual LiDAR signals. The following sections explain the developed Python script, that extracts the relevant information produced by the Range Finder Module (RFM) of the Bepi-Colombo laser altimeter. Further, the individual samples of the produced signal are fitted and stored in an ASCII text file.

3.2.1 Used Libraries

For this script, five libraries come into use. The first is *os*, which is used to create the output directories for the text files and plots. *Numpy* is used for basic array creation and manipulation. To plot the fitting parameters *pyplot* from the visualization library *matplotlib* is used. The fitting of BELA data is done with *curve_fit* from the scientific python library *scipy*. The last library *time* is used to create unique timestamps for the fitting parameters to keep track of different runs and tests of initial guesses and boundaries for the fitting algorithm.

```
1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.optimize import curve_fit
5 import time
```

3.2.2 Main

The main part of the script starts with the definition of the input file, which is the file path to the BELA test data file described in section 3.1. The next step uses the function *get_waveforms* described in section 3.2.3, which gathers the waveform relevant data and parses it into the variable *all_data*. Since the data is in the form of a dictionary, the sample data can be accessed with the keywords *tx_samples* and *rx_samples*. A bias level of 100 ADU is added to the signal since the data has been bias-corrected. This is indicated by negative ADU (Analog-Digital-Unit) values in the signal. An ADC output is in an unsigned binary format, which cannot directly display negative values.

The next lines are not that significant for the current usage of the BELA data. The BFO is the so-called basis function output. The extraction of the two BFOs for transmitted (TX) and received (RX) signals was previously useful for the investigation of the particular datasets. As mentioned in section 3.1, the values present in this dataset stem from a classified algorithm, which is not simple to re-engineer and is beyond the scope of this work. But since *tx_bfo* is used to iterate over the whole BELA dataset in line 53, those remnants of previous versions of the code are preserved.

As the next step, the output directories are defined for transmitted and received signals separately. These directories are subsequently created via try-except statements. This prevents the script from raising *FileExistsError*, when the desired output directories already exist. This is intentional and leads to no conflicts, as the saved files have individual filenames and do not overwrite each other. To get individual filenames the date and time are fetched in line 40, shortly before the creation of the file. Lines 43 and 44 create and open the text files, in which the fit parameters are going to be saved, in write mode, for TX and RX signals, respectively.

The boolean *plot_enabled* enables or disables plotting for the current run. A single plot of one waveform generated by this script can be seen in figure 3.1.

The variable *l*, in line 50, is a simple loop counter that is also used for the loop condition. The while loop iterates over all waveforms present in the BELA input file. The condition for this loop is based on the BFOs. Since every signal has a BFO with a length of 9 parameters per waveform the loop condition can be set to $9 \cdot l < \text{len}(\text{tx_bfo})$, where *l* is the number of waveforms in the dataset.

In the loop, firstly, the script prints the current data packet number, or rather waveform number, to give feedback on the progress of the script. Afterward, if plotting is enabled, the subplot is set up. In this case, there are a total of four plots that get printed, two rows and two columns with x and y being 2. Following the if condition the samples for a single waveform are obtained. There are 42 samples per waveform, thus the samples for one waveform are indexed with $42 \cdot l$ to $42 \cdot l + 1$. Lines 70 through 73 perform the data fitting. The fit function is described in more detail in section 3.2.5.

If the fit is successful, the fit parameters are added to the corresponding lists for TX and RX signals. Otherwise, the script stops and raises an error indicating the data packet in which the fit failed. The next step is to write the initial guess $p0$ and the boundaries $bounds$ as the first two lines in the output text file. The subsequent lines write the fitting parameters to the text file.

When the fit parameters are saved and plotting is enabled, lines 98 to 145 are executed. There, the TX signal of one waveform gets split into two ADC signals. Next, the plot is set to 0 to 900 for the y-axis, for better comparison between the individual subplots in the final plot. The same is done for the plot of the different ADC signals. Lines 121 to 135 are the same as the previously described code but for the RX signal. The nested for-loops label the x- and y-axis for each subplot. It also generates the legends for each plot. Subsequently, the titles of the plots are defined. The if-statement block ends with the saving and closing of the created plot. The last line of the while-loop increases the loop counter by 1. Before the script closes, the text files are closed, which saves the written parameters from lines 90 to 98.

```
1
2 # BELA Input Data
3 seq_file = r"BELA_Testdata\Seq_221_result_repeated.txt"
4
5 # extract data from BELA file
6 all_data = get_waveforms(seq_file)
7
8 # extract signal values and add bias level to signal
9 tx_samples = all_data["tx_samples"] + 100
10 rx_samples = all_data["rx_samples"] + 100
11
12 # extract basis function output
13 tx_bfo = all_data["tx_bfo"]
14 tx_bfo = tx_bfo.tolist()
15 rx_bfo = all_data["rx_bfo"]
16 rx_bfo = rx_bfo.tolist()
17
18 # set output directory
19 tx_outdir = r"2024_03_07_TX_reduced"
20 rx_outdir = r"2024_03_07_RX_reduced"
21
22 # create TX output directory
23 try:
24     os.mkdir(tx_outdir)
25 except FileExistsError as fee:
26     rel_dir = tx_outdir.split("\\")[-1]
27     print(fee)
28     print(f"Directory .\\{rel_dir} already exists\n")
29
30 # create RX output directory
31 try:
32     os.mkdir(rx_outdir)
33 except FileExistsError as fee:
34     rel_dir = rx_outdir.split("\\")[-1]
35     print(fee)
36     print(f"Directory .\\{rel_dir} already exists\n")
37
38
39 # get current time
40 timestr = time.strftime("%Y%m%d-%H%M%S")
41
42 # create new txt file for fit parameters
```

```

43 tx_exports_txt = open(tx_outdir + "\\" + timestr + "_exports.txt", "w")
44 rx_exports_txt = open(rx_outdir + "\\" + timestr + "_exports.txt", "w")
45
46 # enable/disable signal plotting
47 plot_enabled = True
48
49 # loop counter
50 l = 0
51
52 # iterate over extracted signals
53 while 9*l < len(tx_bfo):
54     print(f"\nData Packet {l+1} out of {len(tx_bfo)//9} Packets")
55
56     # create plot
57     if plot_enabled:
58         x = 2
59         y = 2
60         fig, ax = plt.subplots(
61             x, y, figsize=(4*y, 3*x), dpi=300, constrained_layout=True)
62
63         fig.suptitle("BELA TX and RX Signal")
64
65     # get samples for one waveform
66     one_tx_samples = tx_samples[42*l : 42*(l+1)]
67     one_rx_samples = rx_samples[42*l : 42*(l+1)]
68
69     # perform signal fitting
70     tx_valid_fit, tx_params, tx_p0, tx_bounds, tx_ax = fit(
71         one_tx_samples, ax, "tx")
72     rx_valid_fit, rx_params, rx_p0, rx_bounds, rx_ax = fit(
73         one_rx_samples, ax, "rx")
74
75     # check if fit was successful
76     if tx_valid_fit == True:
77         print("Successful TX fit")
78     else:
79         raise Exception(f"TX fit failed for data packet {l+1}")
80
81     # check if fit was successful
82     if rx_valid_fit == True:
83         print("Successful RX fit")
84     else:
85         raise Exception(f"RX fit failed for data packet {l+1}")
86
87
88     # write boundaries and initial guess in first lines of txt
89     if l == 0:
90         tx_exports_txt.writelines(f"p0 = {tx_p0}\n")
91         tx_exports_txt.writelines(f"bounds = {tx_bounds}\n")
92
93         rx_exports_txt.writelines(f"p0 = {rx_p0}\n")
94         rx_exports_txt.writelines(f"bounds = {rx_bounds}\n")
95
96     # write fit params to file
97     tx_exports_txt.writelines(f"{tx_params}\n")
98     rx_exports_txt.writelines(f"{rx_params}\n")
99
100    # plot extracted
101    if plot_enabled:
102        # differentiate between the two ADCs
103        tx_adc_1 = [i for index, i in enumerate(one_tx_samples)
104                    if (index+1) % 2 == 1]
105        tx_adc_2 = [i for index, i in enumerate(one_tx_samples)
106                    if (index+1) % 2 == 0]
107
108        # plot all samples together
109        tx_ax[0, 0].set_ylim(0, 900)

```

```

110     tx_ax[0, 0].plot(one_tx_samples, label="TX Signal")
111
112     # plot each ADC separately
113     tx_ax[0, 1].set_ylim(0, 900)
114     tx_ax[0, 1].plot(np.arange(0, 42, 2), tx_adc_1,
115                     label="ADC 1", c="tab:orange")
116     tx_ax[0, 1].plot(np.arange(1, 43, 2), tx_adc_2,
117                     label="ADC 2", c="tab:red")
118
119
120     # differentiate between the two ADCs
121     rx_adc_1 = [i for index, i in enumerate(one_rx_samples)
122                if (index+1) % 2 == 1]
123     rx_adc_2 = [i for index, i in enumerate(one_rx_samples)
124                if (index+1) % 2 == 0]
125
126     # plot all samples together
127     rx_ax[0, 0].set_ylim(0, 900)
128     rx_ax[1, 0].plot(one_rx_samples, label="RX Signal")
129
130     # plot each ADC separately
131     rx_ax[1, 1].set_ylim(0, 900)
132     rx_ax[1, 1].plot(np.arange(0, 42, 2), rx_adc_1,
133                     label="ADC 1", c="tab:orange")
134     rx_ax[1, 1].plot(np.arange(1, 43, 2), rx_adc_2,
135                     label="ADC 2", c="tab:red")
136
137     for i in range(x):
138         for j in range(y):
139             ax[i, j].set_xlabel("Sample Points [1]")
140             ax[i, j].set_ylabel("ADU Counts [1]")
141             ax[i, j].legend()
142
143     tx_ax[0,0].set_title("TX Signal")
144     tx_ax[0,1].set_title("TX Signal splitted per ADC")
145     rx_ax[1,0].set_title("RX Signal")
146     rx_ax[1,1].set_title("RX Signal splitted per ADC")
147
148     fig.savefig(rx_outdir + r"\fit_plots_plot_" + str(l+1) + ".png")
149
150     plt.close()
151
152     l += 1    # loop counter + 1
153
154 tx_exports_txt.close()
155 rx_exports_txt.close()

```

3.2.3 Reading BELA Data File

This function, called by the previously described main part of the BELA data fitting script, reads the BELA test data file from Thomas et al. (2021), and extracts all waveform relevant parameters, such as samples, offsets, and BELA proprietary fit parameters. Due to the confidentiality of the data processing algorithm, only the samples are of use for further processing. The other parameters are still being extracted, because of the reuse of the *get_waveform* method in this script. Previously developed scripts dealt with the analysis of all data present in the test data file, which is why there are more parameters extracted, as needed.

Each type of data points is first and foremost defined as a numpy array, separated by TX (emitted pulse) and RX (received pulse). Next, the BELA data file is read. To prevent errors in evaluating the lines later, the boolean variables *true* and *false* are set. In line 40 an iterable object is created from the input file, where each line of the text file is a separate entry in the generated list.

To avoid any false detection of "*dp_pulse*" packets, the loop skips the first 94 lines in the text file, because they do not contain any pulse data packets. If a "*dp_pulse*" packet is detected from

then on, the line gets evaluated. This turns the line in the ASCII text file from a string to a list, where different parameters are the entries of the list. The first entry is the time and date of the data packet, the second entry is the type of data packet, namely *"dp_pulse"*, *"cp_control"*, *"control"* or *"delay"*. The third entry contains the actual data of the instrument, see tables 3.2 through 3.6. The data is presented as a dictionary with the keys being the names of the different datasets. The code block from lines 53 to 61 reads the datasets for the emitted laser pulse. The subsequent block from lines 63 to 71 adds the read TX data to the previously gathered TX data. Following that, the return pulses are being looked into. Because there are four returns for every RX data packet, the code iterates over all received RX signals. In the for-loop a filter condition is applied, where only returns with ADU values above 5 are preserved. All other returns are neglected, as they only contain noise around the bias level. This results in one valid return for every transmitted pulse. When the return parameters are read, they are stacked onto the previously recorded data.

After every line in the BELA data file is read, the gathered data is put into a single dictionary, that gets returned by the *get_waveform* function.

```

1
2 def get_waveforms(seq_file):
3     """
4     extract information from BELA data file
5
6     Args:
7         seq_file (str): path to BELA data file
8
9     Returns:
10        all_data (dict): gathered relevant BELA data
11    """
12
13    ## Emitted Pulse
14    tx_sample_offsets_list = np.array([])
15    tx_family_and_corr_index_list = np.array([])
16    tx_bfo_scaling_list = np.array([])
17    # Basis Function Output
18    tx_bfo_list = np.array([])
19    # Samples (Integers, supposedly from ADC)
20    tx_samples_list = np.array([])
21
22    ## Received Pulse
23    rx_sample_offsets_list = np.array([])
24    rx_sample_powers_list = np.array([])
25    rx_family_and_corr_index_list = np.array([])
26    rx_bfo_scaling_list = np.array([])
27    # Basic Function Output
28    rx_bfo_list = np.array([])
29    # Samples (Integers, supposedly from ADC)
30    rx_samples_list = np.array([])
31
32    # open BELA data file
33    with open(seq_file, "r") as file:
34
35        # set boolean for eval(line)
36        true = True
37        false = False
38
39        # read all lines in file
40        lines = file.readlines()
41
42        # iterate over lines in file
43        for i, line in enumerate(lines):
44            if "dp_pulse" in line and i >= 94:
45
46                # create list from line
47                all_packets = eval(line)
48
49                # get data packet
50                packet = all_packets[2]["packet"]

```

```

51
52
53     ## Emitted Pulse
54     tx_pulse = packet["tx_pulse"]
55     tx_sample_offsets = packet["tx_sample_offsets"]
56     tx_family_and_corr_index = tx_pulse["family_and_corr_index"]
57     tx_bfo_scaling = tx_pulse["bfo_scaling"]
58     # Basis Function Output
59     tx_bfo = tx_pulse["bfo"]
60     # Samples (Integers, supposedly from ADC)
61     tx_samples = tx_pulse["samples"]
62
63     # write all to list
64     tx_sample_offsets_list = np.hstack(
65         (tx_sample_offsets_list, tx_sample_offsets))
66     tx_family_and_corr_index_list = np.hstack(
67         (tx_family_and_corr_index_list, tx_family_and_corr_index))
68     tx_bfo_scaling_list = np.hstack(
69         (tx_bfo_scaling_list, tx_bfo_scaling))
70     tx_bfo_list = np.hstack((tx_bfo_list, tx_bfo))
71     tx_samples_list = np.hstack((tx_samples_list, tx_samples))
72
73
74     ## Received Pulse
75     rx_pulse = packet["rx_pulse"]
76     rx_sample_offsets = packet["rx_sample_offsets"]
77     rx_sample_powers = packet["rx_sample_powers"]
78
79     # 4 Returns per Emitted Pulse
80     for rtrn in rx_pulse:
81         rx_family_and_corr_index = rtrn["family_and_corr_index"]
82         rx_bfo_scaling = rtrn["bfo_scaling"]
83         # Basis Function Output
84         rx_bfo = rtrn["bfo"]
85         # Samples (Integers, supposedly from ADC)
86         rx_samples = rtrn["samples"]
87
88         # Filter Condition for "correct" returns
89         if max(rx_samples) > 5:
90
91             # write all to list
92             rx_family_and_corr_index_list = np.hstack(
93                 (rx_family_and_corr_index_list,
94                  rx_family_and_corr_index))
95             rx_bfo_scaling_list = np.hstack(
96                 (rx_bfo_scaling_list, rx_bfo_scaling))
97             rx_bfo_list = np.hstack((rx_bfo_list, rx_bfo))
98             rx_samples_list = np.hstack(
99                 (rx_samples_list, rx_samples))
100
101     # write all to list
102     rx_sample_offsets_list = np.hstack(
103         (rx_sample_offsets_list, rx_sample_offsets))
104     rx_sample_powers_list = np.hstack(
105         (rx_sample_powers_list, rx_sample_powers))
106
107
108     # pack data into dictionary
109     all_data = {
110
111         ## Emitted Pulse
112         "tx_sample_offsets" : tx_sample_offsets_list,
113         "tx_family_and_corr_index" : tx_family_and_corr_index_list,
114         "tx_bfo_scaling" : tx_bfo_scaling_list,
115         # Basic Function Output
116         "tx_bfo" : tx_bfo_list,
117         # Samples (Integers, supposedly from ADC)

```

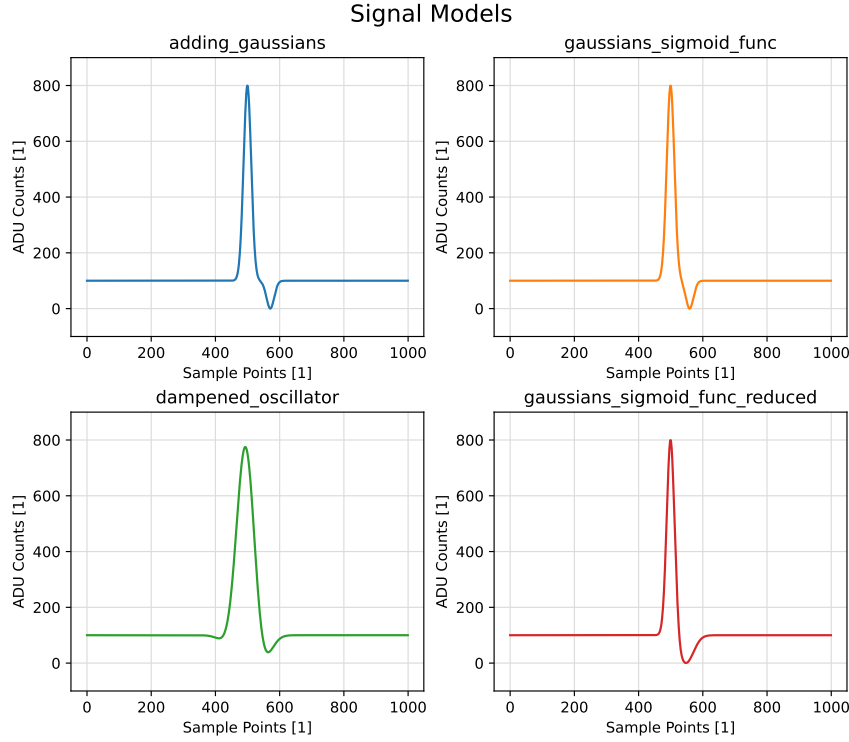


Figure 3.5: Four models used to recreate the response of the APD detector, including the low pass filter signature in the signal, that is sent to the ADC.

```

118     "tx_samples" : tx_samples_list,
119
120     ## Received Pulse
121     "rx_sample_offsets" : rx_sample_offsets_list,
122     "rx_sample_powers" : rx_sample_powers_list,
123
124     "rx_family_and_corr_index" : rx_family_and_corr_index_list,
125     "rx_bfo_scaling" : rx_bfo_scaling_list,
126     # Basis Function Output
127     "rx_bfo" : rx_bfo_list,
128     # Samples (Integers, supposedly from ADC)
129     "rx_samples" : rx_samples_list,
130
131 }
132
133 return all_data

```

3.2.4 Model Functions

During the analysis of the BELA data, several features in the signal surfaced. As can be seen in figure 3.1, the signal starts with a sharp positive slope beginning at the bias level. Then a peak occurs at approximately 10 ns (1 sample point) after the initial rise of ADU counts. This is followed by a steep drop, such that the signal up to this point resembles a simple Gaussian peak. After the initial peak a voltage undershoot is detectable, with a relaxation time of around 25 ns (2 sample points). This phenomenon arises due to a low-pass filter used together with an avalanche photodiode (APD), described by Fraczek et al. (2013). This signal property is expected, as it can be seen in models of BELA and its behavior in Beck (2012).

Adding Gaussians

The first approach was to use two Gaussian signals, one positive and one negative, and to overlap them to recreate the signal. The maximum and minimum of the signal can be easily set via the amplitude parameters A_1 and A_2 . The position of the Gaussian peaks is set by μ_1 and μ_2 , for the positive peak and the negative peak, respectively. σ_1 and σ_2 define the width of the individual peaks. Lastly, the bias level of the signal can be set with *offset*. This results in the following mathematical model:

$$f(x) = A_1 \cdot e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + A_2 \cdot e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} + \text{offset} \quad (3.1)$$

The Python implementation can be seen here:

```
1 def adding_gaussians(x, amp1, mu1, sig1, amp2, mu2, sig2, offset):
2     return amp1 * np.exp(-((x-mu1)**2)/(2*sig1**2)) + amp2 * np.exp(
3         -((x-mu2)**2)/(2*sig2**2)) + offset
```

The major drawback of this approach can be seen in figure 3.5. The two peaks are not joined together, which resulted in bad fits, as the second peak completely separates from the main peak for some data samples.

Gaussians Sigmoid Func

The second attempt builds on the first approach, in the way, that two Gaussians are used. The key difference is that the second peak is skewed with a sigmoid function. The parameter a , of the sigmoid function determines the slope of the sigmoid function and therefore controls the transition from the first Gaussian to the second Gaussian.

The expected outcome was that the fit would yield better results due to the voltage undershoot in the signal not being a perfect Gaussian. The mathematical model of the modified *adding_gaussians* model is as follows:

$$f(x) = A_1 \cdot e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + A_2 \cdot e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} \cdot \frac{1}{1 + a^{-\left(x - \frac{\mu_1 + \mu_2}{2}\right)}} + \text{offset} \quad (3.2)$$

The Python implementation can be seen here:

```
1 def gaussians_sigmoid_func(x, amp1, mu1, sig1, amp2, mu2, sig2, factor, offset):
2     return amp1 * np.exp(-((x-mu1)**2)/(2*sig1**2)) + (
3         amp2 * np.exp(-((x-mu2)**2)/(2*sig2**2))) * (
4         1/(1+factor**-(x-((mu1+mu2)/2)))) + offset
```

This model achieved better results than the previous model, although the problem of separation of the peaks persists. Additionally, this approach adds another parameter *factor* to the model, which reduces the efficiency of the fit.

Dampened Oscillator

A single swing can be seen when looking at the signal at hand, which could be interpreted as a dampened oscillator. The dampened oscillator is then convolved with a Gaussian curve to achieve the constant signal that precedes the peak of a signal. The mathematical model looks like this:

$$f(x) = A \cdot e^{-\frac{bt}{2m}} \cos\left(\frac{\sqrt{4mw^2 - b^2}}{2m}t\right) e^{-\frac{(t-\mu_1)^2}{2\sigma_1^2}} + \text{offset} \quad (3.3)$$

The Python implementation can be seen here:

```
1 def dampened_oscillator(t, amp, mu1, sig1, b, m, w, offset):
2     return amp * np.exp(-b*t/(2*m)) * np.cos(
3         np.sqrt(4*m*w**2 - b**2)/(2*m)*t) * np.exp(
4         -((t-mu1)**2)/(2*sig1**2)) + offset
```

The approach may seem more sophisticated than the previously presented models but lacks simplicity. The parameters highly influence each other, which complicates the overall handling of the model. Over- and undershoots of the desired signal are likely, which is illustrated in figure 3.5.

Gaussians Sigmoid Func Reduced

Building upon the earlier models, the final model uses the general idea of the method *gaussians_sigmoid_func* and reduces the number of free parameters by establishing a dependency of the parameters of the second Gaussian on the parameters of the first Gaussian. The so-called *factor a*, of the sigmoid function, is set to 1000, as that empirically determined value generates smooth transitions from the positive to the negative peak. The mathematical model is the same as *gaussians_sigmoid_func*:

$$f(x) = A_1 \cdot e^{\frac{-(x-\mu_1)^2}{2\sigma_1^2}} + A_2 \cdot e^{\frac{-(x-\mu_2)^2}{2\sigma_2^2}} \cdot \frac{1}{1 + a^{-\left(x - \frac{\mu_1 + \mu_2}{2}\right)}} + \text{offset} \quad (3.4)$$

The Python implementation of the reduced model, on the other hand, is different, as can be seen here:

```

1 def gaussians_sigmoid_func_reduced(x, amp1, mu1, sig1, amp2, offset):
2     mu2 = mu1+2
3     sig2 = 2*sig1
4     factor = 1000
5     return amp1 * np.exp(-((x-mu1)**2)/(2*sig1**2)) + (
6         amp2 * np.exp(-((x-mu2)**2)/(2*sig2**2)))*(
7             1/(1+factor**-(x-((mu1+mu2)/2)))) + offset

```

This model produced the best fits, with a realistic resemblance to the input data (figure 3.5). The computational cost is much lower than the not-reduced version *gaussians_sigmoid_func*. This approach is used for all generated data used in this thesis.

3.2.5 Data Fitting

The data fitting method used for the waveform parameter determination is based around the function *curve_fit* of the module *optimize* from the python library *scipy*. *curve_fit* uses non-linear least squares optimization to fit a function to provided data (Virtanen et al. (2020)).

The primary inputs of *curve_fit* are the model function and the data, the model is fitted to. In the presented case, models outlined in section 3.2.4, are used to fit BELA samples from the *all_data* dictionary of section 3.2.3.

At the start of the *fit* method two arrays are created, *x* and *y*. The first being the indices of the data points ranging from 0 to 41, i.e. 42 sample points. *y*, on the other hand, is the array of the BELA samples. In preparation for the fit, the initial guess *p0* and boundaries *bounds* of the fit are defined. These values were empirically determined through multiple iterations of the script, with parameters being constrained after each run to establish an optimal parameter set.

The call of *curve_fit* is done with several try-except-statements. The problem that leads to this approach is, that *curve_fit* raises *RuntimeError*, when no fit parameters can be determined after a certain amount of function evaluations. The first try sets the maximum function evaluations to 1000. If the fit is inconclusive, *RuntimeError* is raised and the next higher *maxfev* is used. The maximum function evaluations increase by a factor of 10 every time a fit fails. The maximum *maxfev* that can be reached is 10^7 . A higher value leads to immense computational costs and is no longer feasible. If the fit still fails the fitting procedure is terminated and the script raises an error. With well-chosen boundaries and an appropriate initial guess, no more than 10^4 function evaluations are necessary for *gaussians_sigmoid_func_reduced*. By default a *maxfev* of 10^3 is sufficient and only in rare cases the next higher step is utilized for the reduced model.

After the fit is performed, a new dataset is generated and the residuals to the input data are calculated. If the residuals are hugely different the fit is deemed as invalid, otherwise the parameters are labeled as valid. These are then plotted, depending on the type (TX or RX), with much higher resolution (1000 sample points), to get a concise result of the fit, for later manual assessment of the validity of the parameters.

The method returns the label, if the fit is valid, the fitted parameters, the initial guess, boundaries, and the matplotlib object *ax*.

```

1
2 def fit(data, ax, t_or_r):
3     """
4     fitting function for BELA data
5
6     Args:
7         data (list): BELA signal samples
8         ax (np.ndarray): axes object of matplotlib
9
10    Returns:
11        valid_fit (bool): indicates if fit is valid
12        params (np.ndarray): fit parameters of signal
13        p0 (list): initial guess for fitting parameters
14        bounds (tuple): boundary conditions for fitting parameters
15        ax (np.ndarray): axes object of matplotlib
16    """
17
18    # Define the input and output variables
19    x = np.arange(len(data))
20    y = np.array(data)
21
22    # Fit the model to the data
23    p0 = [700, 20, 0.65, -100, 90]
24    bounds = ((400, 19, 0, -200, 50), (1000, 23, 2, 0, 200))
25
26    # actual signal fitting
27    # the maximal number of calls is increased if fit fails
28    # this approach seemed to be more quickly than just setting
29    # higher maxfev
30    try:
31        print("trying 1000")
32        maxfev = 1000
33        params, cov = curve_fit(gaussians_sigmoid_func_reduced, x, y,
34                                maxfev=maxfev, p0=p0, bounds=bounds)
35    except RuntimeError:
36        try:
37            print("trying 10000")
38            maxfev = 10000
39            params, cov = curve_fit(gaussians_sigmoid_func_reduced, x, y,
40                                    maxfev=maxfev, p0=p0, bounds=bounds)
41        except RuntimeError:
42            try:
43                print("trying 100000")
44                maxfev = 100000
45                params, cov = curve_fit(gaussians_sigmoid_func_reduced, x, y,
46                                        maxfev=maxfev, p0=p0, bounds=bounds)
47            except RuntimeError:
48                try:
49                    print("trying 1000000")
50                    maxfev = 1000000
51                    params, cov = curve_fit(gaussians_sigmoid_func_reduced, x,
52                                            y, maxfev=maxfev, p0=p0,
53                                            bounds=bounds)
54                except RuntimeError:
55                    try:
56                        print("trying 10000000")
57                        maxfev = 10000000
58                        params, cov = curve_fit(gaussians_sigmoid_func_reduced,
59                                                x, y, maxfev=maxfev, p0=p0,
60                                                bounds=bounds)
61                    except RuntimeError:
62                        print("well, this doesn't work")
63                        raise Exception(
64                            "Fitting parameters could not be determined")
65
66    # create new data set from fit parameters
67    fit_data = gaussians_sigmoid_func_reduced(x, *params)

```

```

68
69     # get residuals from actual data
70     residuals = y-fit_data
71
72     # if residuals are too big set fit as invalid
73     if max(abs(residuals)) > 5000*np.std(residuals):
74         valid_fit = False
75         return valid_fit, params, p0, bounds, ax
76     else:
77         valid_fit = True
78
79     # plot smooth fit
80     if plot_enabled:
81         # plot for tx or rx
82         if t_or_r == "tx":
83             tr = 0
84         else:
85             tr = 1
86
87     # sample points for smooth plot
88     x_1000 = np.linspace(0, len(data)-1, 1000)
89
90     # Plot the data and fitted curve
91     ax[tr, 0].plot(x_1000, gaussians_sigmoid_func_reduced(x_1000, *params),
92                   label="Fitted Signal", c="tab:orange")
93
94     return valid_fit, params, p0, bounds, ax

```

3.3 LiDAR Data Generator

To compress a raw data stream, first, the raw data stream has to be generated, when no raw data is available due to confidentiality of the BELA data processing algorithm or patent laws preventing the use of airborne and terrestrial LiDAR scanners. The script described in this section creates signals based on fitting parameters produced by the aforementioned script.

The data stream consists of an initial TX peak preceded by around 100 ns of bias signal. The RX peak is set to a computed position in the signal, reflecting the actual time and distance of a reflected laser pulse of a 500 km distant target. This mimics the MPO orbit close to the initial perihelion of the spacecraft (Hosseiniarani et al. (2020)).

A key element of laser ranging is the ability to detect multiple surfaces within the footprint of the emitted laser pulse. The BELA test data did not contain multi-target returns, which led to the development of the feature *create_false_positives*, which generates randomized Gaussian distributed returns around the computed single return. This method can be enhanced by generating returns based on a digital elevation model (DEM), as characterized by Nishiyama et al. (2022).

The final outputs generated by the signal-generating script are *numpy* proprietary files in which the sample points are stored.

3.3.1 Used Libraries

An essential library for this script is again *numpy*. It helps in merging several waveforms into a single signal, as well as dealing with the overall creation of signals and saving the generated LiDAR data. Output directories are created with *os* and to improve the computation time of the overall script *multiprocessing* is used. With *multiprocessing*, the generation of individual waveforms can be distributed across multiple cores of the executing computer. Since the processes are independent of one another, parallelization is straightforward.

```

1 import numpy as np
2 import os
3 from multiprocessing import Process

```

3.3.2 Input Parameters and Global Variables

These parameters start with the input paths and files of the RX and TX signals produced with the data creation script in section 3.2. The output directory finalizes the first section of parameters.

The simulation parameters start with the distance of the spacecraft to the observing target. This parameter is fixed for the whole simulation. Further versions of the script could make this distance variable, to match the actual orbit, with a periherm of 480 km and an apoherm of 1508 km. Moreover, it could also consider the orbit evolution of the MPO, as described in Hosseiniarani et al. (2020).

The boolean *noise* activates or deactivates noise in the generated signal. The generated noise is characterized by *sigma_noise*, which sets the standard deviation of the Gaussian noise. The next parameter *freq* defines the lowest and highest sampling rate in MHz for the signal simulation. Together with *interval* and *nr_of_waveforms*, the total number of simulations is set. The *interval* defines the step size between sampling rates, i.e. 10 results in sampling rates of 80, 90, 100, etc. The parameter *nr_of_waveforms* specifies how many waveforms per sampling rate are simulated. With the current parameter set a total of 5580 waveforms are generated.

Even though the bias level is provided by the fit parameters in the input files, it can be set differently with the variable *bias*. Each input file contains 4998 waveforms. With *seed* specific waveforms can be chosen. If it is set to -1 a random waveform is chosen.

During the analysis of the BELA data, it was noticed, that the signal ranges from roughly -100 to a maximum of 900, which is about a quarter of the possible range of the 12-bit ADCs used in this instrument. Therefore, the next variable *gain* was implemented. This parameter stretches the resulting signal proportionally to the given value, i.e. a gain of 2 stretches the signal peak from 900 to 1800. For a better comparison of the generated data to the BELA data, a setting of gain = 1 is chosen for all subsequent tests.

For time and distance calculations for the RX signal, the speed of light has to be defined. To keep all distances within the same order of magnitude it is defined in kilometers per second.

The next variable *fp* controls the use of false positive (multiple) returns or single returns. False positives are generated based on the return pulse energy. The base return energy is relatively low in the test dataset. Smaller return energies result in fewer additional return signals. To get more returns per pulse *fp_factor* can be set, as it is a multiplier for the return pulse energy. Together with *amp_fac*, which lessens the initial return energy, the behavior of multiple returns can be adjusted. As can be seen in lines 50, 53, and 56, there are comments present. These comments are the parameters for multiple returns used in this thesis. The active setup, in the code presented, is used for single returns.

The last Boolean in the simulation parameters *randomized* determines if the height and width of the returns, as well as the noise and the position of the false positives, are randomized. The implementation of the deterministic and pseudo-random case is explained in the following subsections.

The plot parameters set the size and resolution of the plots. The Boolean *plot_waveforms* controls, whether or not plots shall be generated, the last variable determines if every generated false positive return gets plotted or not.

```

1
2 # Input\Output files ~~~~~
3
4 RX_folder = r"2024_03_12_RX_reduced"
5
6 RX_file = r"20240315-175956_exports.txt"
7
8 TX_folder = r"2024_03_12_TX_reduced"
9
10 TX_file = r"20240315-175956_exports.txt"
11
12 output_folder = "20240316_npy_single_returns_not_random_80_1000"
13 #output_folder = "20240316_npy_multiple_returns_not_random_80_1000"
14
15 # ~~~~~

```

```
16
17 # Simulation Parameters ~~~~~
18
19 # distance of spacecraft to target [km]
20 distance_to_target = 500
21
22 # add noise
23 noise = True
24
25 # standard deviation of Gaussian noise in signal [ADU]
26 sigma_noise = 3
27
28 # frequencies to simulate [MHz], freq. under 40 MHz not useful (BELA = 80 MHz)
29 freq = [80, 1000]
30
31 # interval between frequencies
32 interval = 10
33
34 # how many TX/RX waveforms to check
35 nr_of_waveforms = 60
36
37 # bias offset of signal
38 bias = 100
39
40 # specify which waveform you want to study, -1 results in random seed
41 seed = -1 #4710
42
43 # gain 1 = same as BELA, gain of signal (to match the maximum of 12-bit, 4096)
44 gain = 1
45
46 # c [km/s]
47 speed_of_light = 299792.458
48
49 # include false returns
50 fp = False #True
51
52 # factor of false positives [energy of RX signal]
53 fp_factor = 0 #5
54
55 # factor (or rather divisor) of initial RX amplitude
56 amp_fac = 1 #2
57
58 # randomize signal
59 randomized = False
60
61 # ~~~~~
62
63
64 # Plot Parameters ~~~~~
65
66 figsize = (8, 6)
67 dpi = 300
68
69 # plot waveforms of rx and tx signal
70 plot_waveforms = True
71
72 plot_single_fps = False
73
74 # ~~~~~
```

3.3.3 Main

The main of the signal-generating script starts by creating the output directory. If the directory already exists, the exception raised by *os* is caught and printed into the command line. The path of the output directory is then checked and if it is not a valid directory, the previously caught

error is raised again and the script gets interrupted.

The next step defines the total number of waveforms that shall be generated. In this case, the total number is 5580 waveforms. With the total number of waveforms being defined, the list of sampling rates for the waveforms is created.

The first step for parallelization is done in line 19. This list is filled with the individual processes in the subsequent for-loop, where each process generates one complete waveform. The loop iterates over all sampling rates present in the list, and it counts the current number of waveforms. *Process* is used to spawn processes by creating objects, that are then called with its *start* method (mul (2024)). The target function of *Process* is described in 3.3.4. When the process object is generated, it is added to the process list *procs* and immediately started. Processes are created as long as the total number of processes does not match or exceed the number of CPU cores of the executing computer. When the maximum number of processes is reached, the script waits for all processes to finish until the next batch of processes is started. After all current processes finish, the list *procs* gets reset to be filled again with new processes in the next iterations. If the loop is finished and the last set of processes is smaller than the CPU count, then the remaining processes are joined outside of the loop. When all processes are completed the script ends.

```
1 if __name__ == "__main__":
2
3     # create output directory
4     try:
5         os.mkdir(output_folder)
6     except Exception as exc:
7         print(exc)
8         if not os.path.isdir(output_folder):
9             raise Exception(exc)
10
11     # number of waveforms to generate
12     length = int(((freq[1]+interval - freq[0]) / interval) * nr_of_waveforms)
13
14     # sampling rates of waveforms to generate
15     sampling_rates = [freq[0] + interval * (i // nr_of_waveforms)
16                       for i in range(length)]
17
18     # define list for parallel processes
19     procs = []
20
21     # prepare and start processes in parallel
22     for wave_count, sampling_rate in enumerate(sampling_rates):
23         # create and start process
24         proc = Process(target=one_waveform, args=(wave_count, sampling_rate,
25                                                  sampling_rates, fp, noise,
26                                                  seed, TX_folder, TX_file,
27                                                  RX_folder, RX_file, amp_fac,
28                                                  fp_factor, sigma_noise,
29                                                  distance_to_target,
30                                                  speed_of_light, randomized,
31                                                  gain, bias, output_folder))
32
33         procs.append(proc)
34         proc.start()
35
36     # start as many processes as cpu cores
37     if len(procs) >= (os.cpu_count()):
38         # complete the processes, i.e. wait for processes to finish
39         for proc in procs:
40             proc.join()
41
42     procs = []
43
44     # complete final processes
45     for proc in procs:
46         proc.join()
```

3.3.4 Generator for One Waveform

When a process is started, this function gets executed. In the beginning *one_waveform* gives feedback on the number of the current waveform that one process currently handles. Afterward, BELA parameters are defined as the basis for the new waveform. When the BELA basis is established, the fitted parameters are read from the generated TX and RX files. Here, the file reader could be implemented outside the waveform generation, so it does not have to be read anew every time a new waveform gets created. This may be considered for future versions of this script.

The next code block comes into play when multiple returns are desired. With the read parameters, the energy of the RX pulse is estimated by computing the area of the first Gaussian peak. Following that, a random factor *rand_fac* is defined, which is used to scale the amplitudes, positive and negative in lines 64 and 75, respectively. The new amplitude makes it possible to compute a new pulse width with the previously calculated pulse energy. This width, i.e. σ , is then scaled down randomly to values between 50% and 100% of the initially computed σ . This prevents the waveform generation from exceeding the initial pulse energy and can leave room for more possible returns, when σ is scaled down. Suppose σ gets smaller than the initial σ . In that case, the initial σ is used, as smaller pulse widths are highly unlikely, because the emitted pulse length is typically broadened by the physical processes present in the instrument's environment.

If false positives are deactivated the RX parameters remain unchanged. In either case, the bias for both the TX and RX parameters gets corrected to create a continuous signal without sudden jumps in the data stream. The next step calculates the theoretical time it takes for a complete laser pulse round trip, from being emitted, reflected from the target surface, and detected by the LiDAR instrument receiver. After the time of flight (ToF) has been calculated, the TX peak is set to -10 code units (CU) of the fitted peak. This results in approximately 135 ns after the whole waveform signal starts. The RX peak gets shifted based on the position of the TX peak in the signal and the previously computed ToF. The ToF is in nanoseconds, which has to be translated in CU, since the fit parameters are for BELA samples (42 samples @ 80 MHz results in 525 ns for one complete BELA waveform).

The if-condition in line 102 serves the purpose of randomization of the RX return time, when *randomized* is set to true. The variable *time_after_rx* sets the recording time of the signal after the theoretical time of return. This is done due to uncertainties of returns when multiple returns are expected and to accurately record the complete return signal including the relaxation time of the receiver. The actual extension of recording time happens in the while-loop, where the *end_time* gets increased until it reaches the desired signal length. An additional time interval is added to increase the certainty of accurately recording the signal. The entire waveform is roughly 0.005 s long, which translates to a theoretical possible scan rate of 200 Hz. For later purposes, this signal length can be reduced when return times are more constrained.

Now that all times for TX and RX peaks are set, the creation of the signal can be performed. To accurately set the sample points of the new signal, the *time_interval* has to be defined. This nanosecond time interval is then used to create the sample points for the current signal by using the *numpy* method *arange*. This method generates an array with equally spaced data points, according to a given step size, between the start and end values. The values in the array are converted to CU to be compatible with the fit parameters. These data points are then fed to the *one_pulse* function described in section 3.3.8.

The initial signals are created. If activated, the next phase involves the modeling of false positives, i.e. multiple returns. At first, a parameter set is established with *create_false_positives*. A detailed explanation of this process can be found in section 3.3.10. These parameters are in the form of a list, which is then iterated. Each waveform parameter set from *create_false_positives* is used to create a signal, identically to the previous modeling of the initial TX and RX signals. In lines 147 and 151 all generated single waveforms are put into the list *all_signals*. These signals are merged into a single continuous waveform containing all generated pulses. Due to the nature of ADCs, the signal values need to be rounded, to comply with the 12-bit data stream of BELA. The waveform is then saved to a *numpy* proprietary file for easier data handling in subsequent scripts.

```

1
2 def one_waveform(wave_count, sampling_rate, sampling_rates, fp, noise, seed,
3                 TX_folder, TX_file, RX_folder, RX_file, amp_fac, fp_factor,
4                 sigma_noise, distance_to_target, speed_of_light, randomized,
5                 gain, bias, output_folder):
6     """
7     Generate one complete waveform with distinct sampling rate, custom
8     Gaussian noise, multiple returns, custom offset and custom gain based on
9     fitted BELA single return waveforms
10
11     Args:
12         wave_count (int): number of waveforms generated
13         sampling_rate (float): current sampling rate
14         sampling_rates (list): all sampling rates
15         fp (bool): generate multiple returns when True
16         noise (bool): add noise to signal when True
17         seed (int): seed for random waveform from sample set
18         TX_folder (str): path to fitted BELA TX waveforms
19         TX_file (str): path to BELA TX waveforms file
20         RX_folder (str): path to fitted BELA RX waveforms
21         RX_file (str): path to BELA RX waveforms file
22         amp_fac (float): factor (or rather divisor) of initial RX amplitude
23         fp_factor (float): factor for false positives [energy of RX signal]
24         sigma_noise (float): standard deviation of Gaussian noise in signal [ADU]
25         distance_to_target (float): distance of spacecraft to target [km]
26         speed_of_light (float): c [km/s]
27         randomized (bool): randomize signal when True
28         gain (float): gain of signal, gain 1 = same as BELA
29         bias (float): ias offset of signal
30         output_folder (str): path to output directory
31     """
32
33     # add to wave count
34     wave_count += 1
35     print(f"Waveform {wave_count} out of {len(sampling_rates)}")
36
37     # Parameters from BELA data to create base waveform ~~~~~~
38
39     # sample indices
40     start_value = 0
41     end_value = 42
42
43     # 80 MHz sampling rate
44     bela_time_interval = 12.5 # ns
45
46     # times in ns
47     start_time = start_value * bela_time_interval
48     end_time = end_value * bela_time_interval
49
50     # ~~~~~~
51
52     # get parameters from txt files
53     all_params_tx, random_int = file_reader(TX_folder + "\\ " + TX_file, randint=
54     seed)
55     all_params_rx, random_int = file_reader(RX_folder + "\\ " + RX_file, random_int)
56
57     # add multiple returns/false positives
58     if fp:
59         # calculate rx pulse energy
60         initial_energy = area_under_gauss(all_params_rx[0], all_params_rx[2])
61
62         rand_fac = np.random.randint(50/amp_fac, 100)/100
63
64         # morph amp1 and sig of rx signal
65         all_params_rx[0] *= rand_fac
66         sig_from_amp = get_sig_from_amp(
67             all_params_rx[0], initial_energy)*np.random.randint(50, 100)/100

```

```

67
68     # correct too small sig
69     if sig_from_amp < all_params_rx[2]:
70         all_params_rx[2] = all_params_rx[2]
71     else:
72         all_params_rx[2] = sig_from_amp
73
74     # morph amp2 of rx signal
75     all_params_rx[3] *= rand_fac
76
77
78     # calculate energy after morph
79     new_energy = area_under_gauss(all_params_rx[0], all_params_rx[2])
80
81     # rest energy for fp
82     rest_energy = (initial_energy - new_energy)+initial_energy*fp_factor
83
84     # correct offsets for continuous signal
85     all_params_tx[-1] = bias/gain
86     all_params_rx[-1] = all_params_tx[-1]
87
88     # compute theoretical time between tx and rx signal
89     # (1*km / km/s) * ns/s
90     theoretical_time = (2*distance_to_target/speed_of_light)*10**9
91
92     ## position tx signal
93     # set tx peak at -10 cu from original signal
94     all_params_tx[1] -= 10
95
96     ## position rx signal
97     # time of rx peak [cu + ns / ns/cu] = [cu]
98     # in cu = code units
99     rx_time = all_params_tx[1] + theoretical_time / bela_time_interval
100
101     # set rx peak at theoretical return time
102     if randomized:
103         # random return time
104         all_params_rx[1] = rx_time * np.random.normal(1, 0.1)
105     else:
106         # exact return time
107         all_params_rx[1] = rx_time
108
109     # record signal a little bit longer
110     time_after_rx = rx_time*bela_time_interval*0.5
111
112     ## calculate new end_time and new end_value of signal based on rx_time ~~~~
113     # this extends the dataset so that tx and rx signal is present
114     while end_time <= rx_time*bela_time_interval+time_after_rx:
115         end_time += end_value * bela_time_interval
116
117     # add one additional time interval
118     end_time += end_value * bela_time_interval
119     end_value = end_value*end_time/525
120     ## ~~~~~
121
122     # time between data points [ns]
123     time_interval = 1/sampling_rate * 1000
124
125     # sample points for signal
126     sample_points = np.arange(
127         start_time, end_time+1, time_interval)/bela_time_interval
128
129     # create tx samples
130     tx_signal = one_pulse(all_params_tx, gain, sample_points)
131
132     # create rx samples
133     rx_signal = one_pulse(all_params_rx, gain, sample_points)

```

```

134
135     if fp:
136         # include false positives
137         false_positives = create_false_positives(all_params_rx, all_params_tx,
138                                                 rx_time, rest_energy,
139                                                 randomized)
140
141         fp_signals = []
142         for fp_param in false_positives:
143             fp_signal = one_pulse(fp_param, gain, sample_points)
144
145             fp_signals.append(fp_signal)
146
147     # merge tx and rx signals
148     all_signals = [tx_signal, rx_signal]
149
150     # add fp signals to tx and rx signals
151     if fp:
152         all_signals.extend(fp_signals)
153
154     # merge all signal features
155     merged_all_signals = merge_all(all_signals, bias, noise, randomized)
156
157     # round signal to create integer values
158     merged_all_signals_rounded = [round(entry) for entry in merged_all_signals]
159
160     # save waveform to npy-file
161     np.save(
162         f"{output_folder}\\LiDAR_time_signal_same_params_{wave_count}_" +
163         f"{distance_to_target}km_{sigma_noise}sig_{sampling_rate}MHz_" +
164         f"{amp_fac}amp_{fp_factor}fp", merged_all_signals_rounded)

```

3.3.5 Fitted Waveform File Reader

This function reads text files created with the BELA data-fitting script. The inputs are the file path of the fit parameters and an optional input for choosing specific waveforms out of the 4998 waveforms. When no waveform or the value -1 is set, a random set of parameters gets chosen.

At first the list *all_params* is defined. Next, the given text file is read, with the method *readlines* a line-wise iterator gets generated, which is used for the subsequent for-loop. The first two lines, which contain the initial guess and boundaries of the fit parameters, are skipped. The following if- and elif-statements are essential for *file_reader*, as they discriminate between the start and the end points of the, as lists formatted, waveform fit values. The lines corresponding to the different cases get formatted and the variable *is_array* gets set to true, when the list of parameters begins and set to false, when the list of parameters ends. When the start of a list in the text file is registered, this line gets saved for the next iteration until the end of the list is reached. A reformatted complete line can then be evaluated in Python, which creates a list, that is then transformed to an array. This array is then added to the variable *all_params*, where all fit parameters are gathered.

Lastly, the index of the specific waveform is determined. As previously described, a given *randint* is adopted into *random_int*, and if *randint* is equal to -1 a random waveform out of the read set of fit parameters is chosen. The return variables are the fit parameters for a specific waveform and the number of the returned waveform.

```

1
2 def file_reader(file, randint=-1):
3     """
4     just a little file reader for files created with get_all_waveforms_*.py
5
6     Args:
7         file (str): file containing parameters for waveform signal functions
8         randint (int, optional): waveform to be used. The default is -1, which
9         results in a random waveform
10

```

```

11 Returns:
12     params (list): parameters for waveform signal function
13     random_int (int): integer of specific waveform
14     """
15
16 all_params = []
17 with open(file, "r") as txt:
18     reader = txt.readlines()
19
20     for line in reader:
21
22         # skip lines with no waveforms
23         if "p0" in line:
24             continue
25         elif "bounds" in line:
26             continue
27
28         # extract parameters of waveforms
29         if "[" in line:
30             is_array = True
31             one_array = []
32             if "]" in line:
33                 is_array = False
34                 one_array.append(line)
35                 np_array_str = " ".join(one_array)
36                 list_str = np_array_str.replace(" ", ",")
37                 list_str = list_str.replace(" ", ",")
38                 list_str = list_str.replace(" ", ",")
39                 list_str = list_str.replace(" ", ",")
40                 list_str = list_str.replace("[ ", "[")
41                 list_str = list_str.replace(" -", "-")
42                 list_str = list_str.replace(" ", ",")
43                 list_str = list_str.replace(".", ".0")
44                 list_str = list_str.replace(",", ",")
45
46                 if type(all_params) != type(np.array([])):
47                     all_params = np.array(eval(list_str))
48                 else:
49                     all_params = np.vstack(
50                         (all_params, np.array(eval(list_str))))
51
52             elif "]" in line:
53                 is_array = False
54                 one_array.append(line)
55                 np_array_str = " ".join(one_array)
56                 list_str = np_array_str.replace(" ", ",")
57                 list_str = list_str.replace(" ", ",")
58                 list_str = list_str.replace(" ", ",")
59                 list_str = list_str.replace(" ", ",")
60                 list_str = list_str.replace("[ ", "[")
61                 list_str = list_str.replace(" -", "-")
62                 list_str = list_str.replace(" ", ",")
63                 list_str = list_str.replace(".", ".0")
64                 list_str = list_str.replace(",", ",")
65
66                 if type(all_params) != type(np.array([])):
67                     all_params = np.array(eval(list_str))
68                 else:
69                     all_params = np.vstack(
70                         (all_params, np.array(eval(list_str))))
71
72             if is_array:
73                 one_array.append(line)
74
75 # get random integer for specific waveform
76 if randint == -1:
77

```

```
78     random_int = np.random.randint(0, len(all_params))
79     else:
80         random_int = randint
81
82     return all_params[random_int], random_int
```

3.3.6 Calculate Area under Gaussian Peak

In order to get the energy of a single return peak, the area under the signal function can be used. The voltage undershoot modeled with equation 3.2.4 can be neglected, as this signal feature does not represent the return energy, but is a property of the detector and data processing electronics. The area under a Gaussian curve is calculated with:

$$F(x) = \int_{-\infty}^{\infty} A \cdot \exp\left(-\frac{(x - \mu)^2}{2 \cdot \sigma^2}\right) dx \quad (3.5)$$

$$\rightarrow F(x) = A \cdot \sqrt{2\pi} \sigma \quad (3.6)$$

where A is the amplitude of the return signal, σ is the standard deviation and μ the position of the Gaussian peak.

The Python implementation is as follows:

```
1 def area_under_gauss(amp, sig):
2     return amp * np.sqrt(2*np.pi) * sig
```

3.3.7 Calculate Sigma of Gaussian Peak by Amplitude and Energy

By restructuring equation 3.3.6, the σ of a return signal can be obtained. This is utilized for the generation of multiple returns, as the result is employed for modeling return signals based on the remaining energy a laser pulse could have, if the initial return does not exhibit the full laser return energy. The energy of a return pulse depends on the target surface, in the same manner as disturbances in the path of the emitted laser signal. In the case of this thesis, the return energies are based on the signals present in the BELA test dataset. The σ of a single return is calculated with:

$$\sigma = \frac{E_{return}}{A \cdot \sqrt{2\pi}} \quad (3.7)$$

where E_{return} is the initial return energy and A the amplitude of the signal.

The Python implementation can be seen here:

```
1 def get_sig_from_amp(amp, initial_energy):
2     return initial_energy/(np.sqrt(2*np.pi)*amp)
```

3.3.8 Generation of a Single Pulse

A single pulse, whether it is a TX, RX, or false positive signal, is generated by the fixed model, corresponding to the input parameters and the sample points. The input parameters for the models are defined by the input file of the BELA data-fitting script. The *gain* variable performs an amplification of the signal model output. This relates to an actual gain present in electrical circuits to enhance the sensitivity of detectors. The gain typically also stretches the noise of a signal, but in this simulation, the noise parameter is completely separated by the gain, which allows to control both parameters independently to simulate different instrument behaviors. The last input variable is the array of sample points according to the new given sampling rate. The function returns a list of ADC values (integers) corresponding to a digitalized single pulse signal.

```
1 def one_pulse(params, gain, sample_points):
2     """
3     generates waveform pulse
4
5     Args:
6         params (list): parameters for waveform signal function
7         gain (float): gain of signal, gain 1 = same as BELA
8         sample_points (np.array): signal sample points with new sampling rate
9
10    Returns:
11        signal (list): created signal (waveform)
12    """
13    signal = gain * gaussians_sigmoid_func_reduced(sample_points, *params)
14
15    return signal.tolist()
```

3.3.9 Waveform Signal Model

This part of the LiDAR data-generating script is adapted from the data-fitting script with slight adjustments. The dependency of the undershoot is altered to create more refined signals, compared to the fitted parameters of the previous script. Important to note, that the estimations of the fitting script do not have to be used exactly as they are. The BELA data-fitting script solely lays the foundation of waveforms that are generated by a space-borne LiDAR instrument for the instrument simulator. The received amplitude and signal width remain unchanged from the fitted parameters, which further leads to the pulse energy maintaining its properties from the initial BELA waveform.

The alterations in μ_2 , σ_2 , and *factor* morph the negative Gaussian for a smoother transition between the positive signal and the voltage undershoot. The base model remains the same as equation 3.2.4.

```
1 def gaussians_sigmoid_func_reduced(x, amp1, mu1, sig1, amp2, offset):
2     """
3     function for waveform signal
4     positive Gaussian + negative Gaussian * sigmoid + offset
5
6     Args:
7         x (list): x-values of function
8         amp1 (float): amplitude of first Gaussian
9         mu1 (float): center of first Gaussian
10        sig1 (float): standard deviation of main Gaussian
11        amp2 (float): amplitude of second Gaussian
12        offset (float): offset of function from zero (bias)
13
14    Returns:
15        list: waveform signal
16    """
17
18    # get center of second Gaussian
19    mu2 = mu1 + (2/0.7) * (sig1 * 1.5)
20
21    # get sigma of second Gaussian
22    sig2 = 1.5 * sig1
23
24    # determines steepness of sigmoid
25    factor = 1
26
27    return amp1 * np.exp(-((x-mu1)**2)/(2*sig1**2)) + (
28        amp2 * np.exp(-((x-mu2)**2)/(2*sig2**2))) * (
29        1/(1+factor**-(x-((mu1+mu2)/2)))) + offset
```

3.3.10 Creating Multiple Returns/False Positives

A key feature of LiDAR data is the multiple target detection. This allows the measure of elevation differences on a small scale in the area of the laser pulse on-ground footprint. The provided BELA data consists only of separate single returns although a possibility of multiple target detection is implemented in the data processing chain. BELA can measure up to four possible sample windows (Beck (2012)). The analysis of the test dataset showed that only one of four sample windows, for every TX pulse, contains a signal return. All other samples are below 5 ADU, meaning no value in the list of data samples exceeds 5 ADU for three out of four sample windows.

The LiDAR data simulator shall be as close as possible, within the scope of this thesis, to an actual instrument measuring in-situ data. Therefore, the *false_positive* method is introduced, which mimics multiple returns from uneven rugged terrain. The version presented in this thesis generates multiple returns based on randomness or a deterministic algorithm, dependent on the Boolean *randomized*. The number of returns is determined by the energy remaining of the return pulse. The lower the energy of the initial calculated return signal, the more return peaks are to be expected.

The function starts by defining the list *false_positives* and the wave counter for the generated false positives. The returns get generated within a while-loop that is executed as long as energy for additional returns is available. At first, the waveform counter is raised by 1. The next step sets the base list for the parameters of the additional returns. Afterward, the method differentiates between randomized returns and deterministic returns. The randomized signal creation starts by establishing a random number between 0.2 and 0.5. This random factor is then used to set the amplitude of the return. The position of the return in the complete waveform is set by a Gaussian distribution with the mean being the initial return. The standard deviation of this distribution is set by a distance interval of 10 km, where 99% of returns lie from -5 to +5 km of the initial return. This interval gets translated to a timeframe *sigma_time* and to code units, that get further constrained by a factor of 20, so that the returns are closer to the initial return to create overlapping signals.

Typical for LiDAR signals of close laser reflecting surfaces are overlying peaks, which can be challenging for peak extraction methods. Consequently, the aforementioned constraints are applied to create signals that are more difficult to process later on and to create signals that resemble natural signals. With the position being set, the σ can be defined. This is done via the function, described in section 3.3.7. The result of *get_sig_from_amp* is then scaled to reduce the energy of the current return. If no scaling is applied all remaining energy is used up by the present return pulse. The last two parameters are the amplitude of the voltage undershoot and the bias offset. The amplitude gets scaled in the same manner as the amplitude of the positive Gaussian. The offset remains the same as the offset of the initial return.

The creation of deterministic returns differs for the amplitudes and the position of the signals. The alterations are as follows:

The amplitudes are scaled by:

$$A_{\text{new}} = \frac{A_{\text{init}}}{(\text{wave_count} \bmod 3) + 1} \quad (3.8)$$

where A_{init} is the amplitude of the initial return, A_{new} is the amplitude and *wave_count* the number of the current false positive.

The position is set by:

$$\mu_{\text{new}} = \mu_{\text{init}} + \frac{\text{wave_count} \cdot 25}{(\text{wave_count} \bmod 3) + 1} \cdot (-1)^{\text{wave_count}+1} \quad (3.9)$$

where μ_{init} is the position of the initial return, μ_{new} is the position and *wave_count* the number of the current return.

The generated σ of either method, randomized or deterministic, is then checked, whether it is below the σ of the emitted laser pulse. Since the return pulses cannot be of shorter duration than the emitted laser pulse, the generator gets terminated and the generated return gets dismissed. If σ is the same as the TX pulse or larger, the function goes on to compute the rest energy left of the expected return energy. In the case of a premature exit of the while-loop, the whole available energy may not be used entirely. This is done intentionally, because surfaces with a low albedo show this exact property, and therefore a reduction in return energy can occur similarly in an operational scenario. At the end of a loop iteration, if the created parameters are suitable they get added to the false positives list.

The *create_false_positives* function returns a list of parameter sets that are used to model the false positives (multiple returns) as waveforms.

```

1 def create_false_positives(all_params, all_params_tx, index_of_peak,
2                             rest_energy, randomized):
3     """
4     generator of multiple returns/false positives for multiple target detection
5
6     Args:
7         all_params (list): parameters for waveform signal function
8         all_params_tx (list): parameters for tx waveform signal function
9         index_of_peak (float): position of the main return
10        rest_energy (float): energy left of the transmitted light pulse
11        randomized (bool): randomize returns when True
12
13    Returns:
14        false_positives (list): list of parameters for returns
15    """
16
17    # amp1, mu1, sig1, amp2, offset
18    false_positives = []
19
20    # generated waveform counter
21    wave_count = 0
22
23    # generate waveforms as long as energy remains
24    while round(rest_energy) > 0:
25
26        # add waveform
27        wave_count += 1
28
29        # prepare list of params
30        new_all_params = [0] * len(all_params)
31
32        # randomized returns
33        if randomized:
34            # new amp1
35            rand0 = np.random.randint(20, 50)/100
36            new_all_params[0] = all_params[0] * rand0
37
38            # sigma of Gaussian distributed returns
39            # 99% inside of 10 km (gets further constrained at sigma_of_rx_returns)
40            sigma_distance = 10 / 3
41
42            # time frame of Gaussian returns
43            sigma_time = sigma_distance / speed_of_light
44
45            # time frame in code units (sample/12.5ns; derived from BELA signal)
46            code_unit_samples = sigma_time / (12.5*10**-9)
47
48            # further restrict sigma of returns
49            sigma_of_rx_returns = code_unit_samples/20
50
51            # Gaussian distributed index of return
52            gaussian_index = np.random.normal(index_of_peak, sigma_of_rx_returns)
53

```

```

54         # new position (mu1)
55         new_all_params[1] = gaussian_index
56
57         # new sig1
58         new_all_params[2] = get_sig_from_amp(
59             new_all_params[0], rest_energy)*np.random.randint(50, 100)/100
60
61         # new amp2 (same scaling as amp1)
62         new_all_params[3] = all_params[3] * rand0
63
64         # offset
65         new_all_params[4] = all_params[4]
66
67     # deterministic returns
68     else:
69         # new amp1
70         new_all_params[0] = all_params[0]/(wave_count%3+1)
71
72         # new mu1
73         new_all_params[1] = index_of_peak + wave_count*25/(
74             wave_count%3+1)*(-1)**(wave_count+1)
75
76         # new sig1
77         new_all_params[2] = get_sig_from_amp(
78             new_all_params[0], rest_energy)*np.random.randint(50, 100)/100
79
80         # new amp2
81         new_all_params[3] = all_params[3]/(wave_count%3+1)# * 0.5
82
83         # offset
84         new_all_params[4] = all_params[4]
85
86     # terminate generator when sigma is below tx signal sigma
87     if new_all_params[2] <= all_params_tx[2]:
88         break
89
90     # calculate rest energy
91     rest_energy -= area_under_gauss(new_all_params[0], new_all_params[2])
92
93     # save false positive returns
94     false_positives.append(new_all_params)
95
96     return false_positives

```

3.3.11 Merging All Single Signals

All single waveforms need to be merged to create a continuous realistic datastream. The merged signal starts as a numpy array with zeros as floating point variables. Then every signal is added to *merged_signal*. Because the bias level is added in every iteration, it is subsequently subtracted from the aggregated signal, so that the correct bias level is reached. The last step adds a Gaussian-distributed noise to the merged signal, which is described in section 3.3.12.

```

1 def merge_all(all_signals, bias, noise, randomized):
2     """
3     merge signals to get one continuous signal
4
5     Args:
6         all_signals (list): list of signals (TX, RX and FP)
7         bias (float): ADC bias level
8         noise (bool): add noise to signal
9         randomized (bool): randomize noise
10
11     Returns:
12         merged_signal (np.array): continuous signal with all input signals
13     """
14

```

```
15 # prepare merged signal
16 merged_signal = np.array([0.0]*len(all_signals[0]))
17
18 # add signals to merged signal
19 for signal in all_signals:
20     merged_signal += np.array(signal)
21
22 # correct for bias levels in signal
23 merged_signal -= bias*(len(all_signals)-1)
24
25 # add noise to signal
26 if noise:
27     merged_signal = add_noise(merged_signal, sigma_noise, randomized)
28
29 return merged_signal
```

3.3.12 Adding Noise

The BELA test dataset was analyzed and a clear noise behavior was seen. In one complete sample set (42 samples) two peaks emerged from the histogram of the bias signal. These two peaks are due to the two ADCs used in the RFM of BELA. By separating all even and odd sample points a Gaussian distributed noise can be seen with a standard deviation of approximately 3. Therefore the foundation of the noise behavior was set.

If a randomized noise shall be generated, the signal gets iterated and with the *numpy.random* method *normal* a random deviation of the original signal is applied. The method is based on a Gaussian distribution, where the first input parameter denotes the mean of the distribution and the second input is the standard deviation of the Gaussian.

When a deterministic noise, for reproducibility, is needed a seed of 100 is set. This allows the exact same noise behavior for different sample sets and leads to more suitable datasets for the tests of the peak extraction methods in section 5.1. Following the definition of the seed, a noise dataset is generated with a mean of 0 and the given standard deviation. Lastly, the noise dataset is added to the LiDAR signal.

The return of the *add_noise* function is the input signal with a Gaussian distributed noise.

```
1 def add_noise(signal, sigma_noise, randomized):
2     """
3     adding normal distributed noise to signal
4
5     Args:
6         signal (list): list of signal
7         sigma_noise (float): sigma of normal distributed noise
8         randomized (bool): randomized noise
9
10    Returns:
11        with_noise (np.array): signal with normal distributed noise
12    """
13
14    # add random noise or with specific seed
15    if randomized:
16        # add Gaussian noise to signal
17        with_noise = [np.random.normal(i, sigma_noise) for i in signal]
18    else:
19        # Set a seed for reproducibility
20        np.random.seed(100)
21
22        # Generate the Gaussian noise
23        mean = 0 # Mean of the Gaussian noise
24
25        # noise around zero
26        noise = np.random.normal(mean, sigma_noise, len(signal))
27
28        # add noise to signal
29        with_noise = signal + noise
30
31    return with_noise
```

4. Data Compression

4.1 Entropy of Data

A fundamental element in the field of data compression is the concept of entropy for information in data. Claude E. Shannon presents a measure to quantify the information produced by a Markov process. For each data point within a dataset, a probability of occurrence can be evaluated. These probabilities, together with the information contained in the data point, enable the calculation of the uncertainty of an outcome in the dataset. The introduced Shannon entropy is in the form of

$$H = -K \sum_{i=1}^n p(i) \log p(i) \quad (4.1)$$

where K is a positive constant used to apply or scale the unit of measure, and p_i is the probability of the sample i (Shannon (1948)).

To adjust the Shannon entropy to accommodate the usage for digital data, the equation above changes to

$$H(x) = - \sum_{i=1}^n p(x) \log p(x) / \log(2) \quad (4.2)$$

which is the form referred to for all mentions of entropy from this point onward. The unit of measure for this equation is bits/sample, i.e. how many bits are needed on average to represent a single data point.

4.2 Lossless Compression

Lossless compression refers to the compression of data, that involves no loss of information. Lossless compressed data can be fully reconstructed to the original data, without any information being lost (Sayood (2000)). This concept is crucial for scientific data as it does not distort, or disrupt any of the gathered signals. An example of the importance of lossless compression would be the sentences "Do *not* send money" and "Do *now* send money" (Sayood (2000)). Although these sentences only differ in one single character, the meaning could not be more different.

Objects observed by satellites often are processed on ground to enhance different features of the signal. By that time, if the gathered data has errors due to lossy compression techniques, it is too late to do another observation. At first, the faulty compressed data may seem similar to the uncompressed data, but due to post-processing minor errors can have huge implications. Therefore, it is not advisable to allow for any differences to appear in the compression process (Sayood (2000)).

4.3 Lossy Compression

In cases, where the compression ratio is more important than the complete reversibility of the gathered instrument information, lossy compression techniques can be applied. Lossy compression describes processes, which involve some loss of information in the data set when compression is performed. Lossy compressed data cannot be fully recovered or reconstructed, but this allows much higher compression ratios compared to lossless compression techniques (Sayood (2000)).

Many applications do not need an exact reconstruction of gathered data. As an example, when storing or transmitting speech, the exact value of each sample of the spoken words is not necessary. The quality of the transmitted data, in this case speech, can be reduced to the point, where each word can still be understood, but has not the same clarity as an in-person conversation (Sayood (2000)). The probably most common case is the transmission of speech over the telephone. The same goes for the recently increasingly used online meetings, where audio and video problems, due to the lack of appropriate bandwidth of the participants, were recurrent problems. Here, lossy compression was needed to accommodate the audio and video data rate for the given bandwidth. The problem that occurs, is when the available bandwidth gets too small, the audio or video gets choppy and unintelligible.

5. Lossy Compression

5.1 Peak Extraction Methods

This section deals with the different peak extraction methods developed in the course of this thesis.

It is important to note, that the input signal is smoothed with a 1-dimensional Gaussian filter before further processing, except for the Range Window method. The Gaussian filter is similar to an arithmetic mean filter, but it uses a kernel in the shape of a Gaussian distribution (gau (2021)).

5.1.1 Range Window

The first method that is implemented in the lossy compression script is inspired by the range window used for BELA. For BELA $200.8 \mu\text{s}$ is used for the entire range window. This is then further reduced to 525 ns subsets (Beck (2012)). In this case, the range window is chosen by the difference between the highest to the lowest point on Mercury, which is about 10 km.

The range window for the outgoing pulse (TX) is recorded for 300 ns. Prior to the rise of the signal of the APD, 100 ns are recorded, then 100 ns are reserved for the rise, fall and relaxation of the APD, and the remaining 100 ns of the total TX range window are recorded. Together the first and last 100 ns are used to measure the bias level and the noise behavior of the signal. The received signal (RX) is expected to be within a certain range, therefore an appropriately chosen time frame, in which the return is expected, can reduce the number of data points immensely. In this case, the range window is based on the highest and lowest point recorded on Mercury, 4.481 km and -5.38 km respectively. This results in a total range of 9.86 km or rather $32.89 \mu\text{s}$. Depending on the precision of the estimation of the expected return signal, the range window can be chosen broader or narrower.

5.1.2 Front2Back

The basic concept of this method is to first check if a signal in a stream of signals is above a certain threshold. If the threshold is exceeded the signal gets recorded, otherwise it gets discarded. The recording of the signal is stopped if either the signal drops below the threshold or the signal drops by 10 % and a new peak gets detected. A new peak can be detected if the signal drops below 90 % of the maximum value that has already been detected for the current peak. The current recording gets stopped and a new peak starts to be recorded if the signal rises again by $1/4 \sigma_{\text{noise}}$. The whole recording stops at the end of the signal stream.

5.1.3 Top2Bottom

This method starts by indexing all values provided. For this case the entire range window, described in section 5.1.1, is used. Then all values, including their indices, are getting sorted from highest to lowest value. The resulting sorted array is then iterated through. The first index is then saved for the first detected peak. If the subsequent indices are adjacent to the saved indices, they are recognized as being associated with the current peak. Indices count as adjacent, if the index is within the range of

$$-(1 + \text{Counter}_{\text{negative}} + \frac{\text{Sampl. Rate}}{240}) \leq \text{Index} \leq 1 + \text{Counter}_{\text{positive}} + \frac{\text{Sampl. Rate}}{240} \quad (5.1)$$

where *Index* is the current index of the value at hand in the sorted array, *Counter_{negative}* and *Counter_{positive}* are counting variables, which give the number of indices that have been saved for the current peak in positive and negative direction from the highest value index (see fig. 5.1). The term $\frac{\text{Sampl. Rate}}{240}$ was empirically determined, and it yields sufficiently accurate results within the scope of this work, where *Sampl. Rate* represents the sampling rate of the signal.

When a peak is detected and the subsequent values are below the given detection threshold, the peak gets saved and deleted from the previously created sorted array. This procedure is repeated until no values are remaining. If by any circumstance two detected peaks intersect with one another, they are merged into a single peak. The last step is to rearrange the values and indices and reduce the number of detected peaks to 10. These 10 peaks are the most prominent signals in the given range window. This number can be changed, depending on the scientific requirements and laser pulse energy of the instrument. For the case of BELA, more than 10 returns are not feasible, due to the pulse energy being too low for more returns.

5.1.4 Curve Discussion

The name of this method may be a little confusing to non-German speakers. It is a direct translation of the German mathematical term "Kurvendiskussion", commonly used in Austrian and German schools for differential calculus, particularly involving the first and second derivatives of curves.

The method starts with saving the unfiltered signal into another array. For the signal that gets processed, all values below the detection threshold are set to 100, to reduce the false detection of small peaks. The next step is to filter the signal with the Gaussian filter to reduce Gaussian noise in the signal. With the filtered signal the first and second derivative of the signal is computed. The smoothed signal is then indexed. This index array is then reduced to indices of values above -1, for both differential datasets. Both index arrays are then further derived so that all array entries are 1, except where the features to be extracted occur. By searching for entries above 1, the maxima in the signal can be found with the index array of the first derivatives. The points of inflection in the signal with a positive slope can be found with the index array of the second derivatives. The inflection points with a negative slope are similar to the points of inflection with a positive slope but have an index plus 1 in the differential index array of the second derivative array. The final step is to get the closest inflection points to the extracted maxima. For each maximum, a peak is extracted from the raw signal with the indices ranging from the inflection point with a positive slope to the inflection point with a negative slope. The idea of using the derivatives of the signal is inspired by the peak extraction presented by Lindner et al. (2015). A detailed explanation of the method developed in the course of this thesis can be found in section 5.2.4.

5.2 Peak Extraction Code Description

This section describes the code implementation of the previously described peak extraction methods. First, the code is described followed by the code itself. Each method is a separate function in the lossy compression script.

5.2.1 Range Window

The basic concept of this method is described in section 5.1.1. Since the range window represents the sample window used for further processing this function is called *create_sample_window*.

The input parameters are:

- **signal:** This parameter represents the input signal coming from the ADU. In this case the signal is a list of 12-bit integer values of an arbitrary length, similarly to BELA.
- **sampling_rate:** Is the sampling rate of the ADU. The sampling rate is represented with a floating point value. It is used to estimate the size of the range window, i.e. how many values are to be expected within the given range.
- **distance_to_target:** Describes the distance to the observation target. For BELA this would be 400 km - 1500 km above Mercury. In this case, 500 km nominal operation height was assumed in the test dataset.
- **time_of_tx:** This parameter represents the time frame [ns] used for the extraction of the TX signal.
- **timestamps:** Here a numpy array is used for the timestamps [ns] of the individual ADU signals.
- **size:** This optional parameter can be used to set a distinct range for the sample window. If no size is given the maximum height difference of Mercury [~ 10 km] is used.

Description

At first, the length of the TX window in samples is computed by taking the time reserved for TX signal extraction and multiplying it with the sampling rate. The terms $10^{** - 9}$ and $10^{** 6}$ are used to set both input variables to SI units, seconds and Hertz, respectively. The product is then rounded to the next highest integer since only a whole number represents the total number of samples in the given time frame correctly.

The next step is to extract the TX signal and timestamps according to the TX time frame. All values with a timestamp below or equal to the time frame are assigned to the TX signal. Values with timestamps above the TX time frame are assigned to the RX signal.

From line 29 to line 53 the size of the range window for the return signal is determined. If no size is set for the input parameters and the distance to the target is above 10 km, the highest and lowest points of Mercury, according to (Becker et al. (2016)), are used to set the range window. The absolute height difference, as well as the time of flight for that difference, is computed. The next step is to determine the number of samples present for the time of flight. This number is then rounded to the next highest integer value in the same manner as previously mentioned in line 18. For the case that the distance to target is below 10 km the whole RX samples are preserved and the range window is equal to the length of the RX signal. Otherwise, if a size for the range window is predefined that size is then used.

To get the appropriate position of the range window the expected sample for the return pulse is estimated with a basic time of flight computation. Since the pulse travels towards the target, gets reflected, and travels back to the instrument, the distance to the target has to be doubled. The sampling rate is converted from MHz to Hz. With the knowledge of the expected sample, the start and end point of the sample window can be established. First, the size of one-half of the sample window is determined. Again, a whole number is needed, but in this case, the number is rounded to the next lowest value. This rounding down has no particular reason or consequence, other than the middle of the range window getting shifted by a half step. This does not impact the time of flight calculations or the indexing of the signal.

The final step is the extraction of the RX signal and timestamps. If the start of the RX range window is within the TX window, the previously determined start, i.e. end of the TX time frame, is utilized. The same goes for the RX timestamps. Otherwise, the determined start of the range window is used.

The returned parameters are the TX signals and timestamps, for the first 300 ns of the dataset, as well as the RX signal and timestamps for the appropriate range window.

Code

```
1 def create_sample_window(signal, sampling_rate, distance_to_target, time_of_tx,
2   timestamps, size=0):
3     """
4     cuts samples from dataset within range window
5
6     Args:
7         signal (list): ADC signal (list of integers)
8         sampling_rate (float): ADC sampling rate
9         distance_to_target (float): distance to target [km]
10        time_of_tx (int): time reserved for TX signal [ns]
11        timestamps (numpy.ndarray): list of timestamps
12        size (int, optional): size of range window (number of samples). Defaults to
13        0.
14
15    Returns:
16        tx_signal (list): emitted pulse signal
17        tx_timestamps (numpy.array): emitted pulse timestamps
18        rx_signal (list): received pulse signal
19        rx_timestamps (numpy.array): received pulse timestamps
20    """
21
22    # length of tx window
23    tx_window = round(np.ceil(time_of_tx*10**-9 * sampling_rate * 10**6))
24
25    # extract tx signal and timestamps
26    tx_signal = signal[timestamps <= tx_time]
27    tx_timestamps = timestamps[timestamps <= tx_time]
28
29    # all timestamps after tx time belong to rx
30    rx_signal = signal[timestamps > tx_time]
31    rx_timestamps = timestamps[timestamps > tx_time]
32
33    # determine size of range window (rw)
34    if size == 0 and distance_to_target >= 10:
35
36        # highest and lowest points on mercury
37        highest_point = 4.481 # km
38        lowest_point = -5.380 # km
39
40        # absolute height difference
41        height_diff = highest_point - lowest_point
42
43        # time of flight (ToF) for height difference
44        height_diff_time = height_diff/speed_of_light
45
46        # number of sample for ToF (float)
47        hdt_nr_samples = height_diff_time*sampling_rate*1000000
48
49        # number of samples in rw (int)
50        sample_window_size = int(np.ceil(hdt_nr_samples))
51
52    # rw for short range (below 10 km)
53    elif size == 0 and distance_to_target < 10:
54        sample_window_size = len(rx_signal)
55
56    # rw for given range
57    else:
58        # time of flight (ToF) for height difference
59        height_diff_time = size/speed_of_light
60
61        # number of sample for ToF (float)
62        hdt_nr_samples = height_diff_time*sampling_rate*1000000
63
64        # number of samples in rw (int)
65        sample_window_size = int(np.ceil(hdt_nr_samples))
```

```

64
65     # expected return sample
66     # correct for 10 cu offset of tx
67     expected_sample = round((distance_to_target*2/speed_of_light) * sampling_rate
68                             *1000000)+10
69
70     # compute theoretical middle point in rw
71     half_size = int(np.floor(sample_window_size/2))
72     start = expected_sample - half_size
73     end = start + sample_window_size
74
75     # extract rx signal and timestamps
76     # prevent tx to be in rx signal
77     if start <= tx_window:
78         rx_signal = rx_signal[:end]
79         rx_timestamps = rx_timestamps[:end]
80     else:
81         rx_signal = rx_signal[start:end]
82         rx_timestamps = rx_timestamps[start:end]
83     return tx_signal, tx_timestamps, rx_signal, rx_timestamps

```

5.2.2 Front2Back

The basic concept of this method is described in section 5.1.2. This method is intended to read a continuous data stream with on-the-fly peak extraction, therefore this method is called *Front2Back*, as it reads the ADC values from the first produced ADC value to the last produced ADC value. This has a few drawbacks, as it is not as fast as all subsequent methods. A detailed evaluation of performance can be found in section 7.1.

The input parameters are:

- **signal**: This parameter represents the input signal coming from *create_sample_window*. In this case the range window of around 10km. The signal is a list of 12-bit integer values.
- **bias**: A floating point variable describing the bias level of the input signal.
- **thresh**: This parameter sets the detection threshold for the extraction method.
- **sigma_noise**: Describes the standard deviation of the Gaussian noise present in the signal. This parameter controls the sensitivity of detecting subsequent overlapping peaks.
- **sigma_filter**: A globally defined standard deviation for the Gaussian filter. This variable is used in the presented way due to testing during the development phase.

Description

To reduce noise the method starts by applying a Gaussian filter. The Gaussian filter used is from the Python library *scipy*. The input parameters for this filter function are the signal and the standard deviation for the Gaussian kernel. The filter performs a weighted mean calculation with a Gaussian kernel. This greatly reduces the Gaussian distributed noise present in the signal, which results in more robust and accurate peak extraction. The next parameter *min_length_of_peak* is used to mitigate the unwanted detection of insignificant small signals. This is one major drawback of this method because the side of the peaks with a negative slope yield several returns for the same peak. A thorough explanation of this property of *Front2Back* can be found in section 7.1.1.

Lines 24 to 29 are definitions of variables for the main loop of this method. The loop for peak extraction ranges from line 32 to 160. To get the index of each ADC data point *enumerate* is used to iterate over the smoothed signal. The first step is to check whether the data point is above the detection threshold. If the signal exceeds the threshold, the boolean *signal_registered* is set to *True*. When a new peak is detected the variable for the start index of the peak *peak_start* is set

to one index below the current index of the signal. This is due to the fact, that later an additional point is added in front of the signal, to get more data points for a single peak, even if those values are below the detection threshold. The next if-statement is used to get the maximum point of the currently recorded peak. The last step in the currently active parent if-statement is to add the current signal point to the currently extracted peak.

The next if-else pair serves the purpose of getting the difference between adjacent points at the current position in the signal. Because the numpy function *diff* returns an array with the length minus one of the input array, the last difference is computed from the last point to the second to last point.

The following statement is crucial for this method, as it allows to extract multiple peaks and overlapping peaks. It checks whether the current point has a value lower than 90 % of the highest recorded value. Both values, the highest and current value are bias-corrected, to check the 10 % drop relative to the bias level. The second term checks if the current point is a rising signal if the difference between points at the current position is above 1/4th of the standard deviation of the Gaussian noise in the signal. The last term checks if a peak is currently being recorded. If all described conditions are met, the current peak is saved and a new peak can be registered. When the length of the recorded signal is above 1, then, as previously mentioned, a data point is added to expand the signal. Now, if all values in the extracted peak are somehow below the detection threshold, the peak gets deleted and the peak extraction gets reset. In the case that values above the detection threshold are present, the peak gets saved into *extracted_peaks*. The subsequent lines as well as lines 91 through 93 are for resetting the peak extraction. The latter reset is for extracted peaks with only one data point, which gets discarded.

The next case comes into effect when the signal drops below the detection threshold and a signal is being recorded. Lines 99 to 128 are the same as 65 to 93, but at line 102 the current signal is added at the end of the peak, again to gain additional information below the detection threshold.

If no signal is registered the peak extraction variables are reset. At the end of the loop, the current point is saved for the next iteration to calculate the difference of adjacent points.

The returned variable is a list of extracted peaks, where each peak is represented as a list with the start point of the peak as the first entry and the corresponding ADC values as the second entry.

Code

```
1 def front2back(signal, bias, thresh, sigma_noise, sigma_filter):
2     """
3     Peak extraction method
4     This method iterates over the whole signal starting from the lowest
5     to highest index in the array.
6
7     Args:
8         signal (list): ADC signal (list of integers)
9         bias (float): bias level of signal
10        thresh (int): detection threshold
11        sigma_noise (float): standard deviation of signal noise
12        sigma_filter (float): globally defined sigma for Gaussian filter
13
14    Returns:
15        extracted_peaks (list): extracted peaks of ADC signal
16    """
17
18    # apply gaussian filter
19    smoothed_signal = gaussian_filter1d(signal, sigma=1.5*sigma_filter)
20
21    # minimum length required for valid peak
22    min_length_of_peak = 3
23
24    # prepare start variables for loop
```

```
25     prev_point = 0
26     peak_start = 0
27     signal_registered = False
28     extracted_peak = []
29     extracted_peaks = []
30     max_point = min(signal)
31
32     # iterate through filtered signal
33     for i, point in enumerate(smoothed_signal):
34
35         # register signal above detection threshold
36         if point >= thresh:
37             signal_registered = True
38
39             # set start index and value of peak
40             if peak_start == 0:
41                 peak_start = i-1
42                 max_point = point
43
44             # get maximum point of peak
45             if prev_point < point:
46                 max_point = point
47
48             # add current signal to peak
49             extracted_peak.append(signal[i])
50
51             # get difference from current point to next point
52             if i < len(signal)-1:
53                 diff_point = np.diff(signal)[i]
54
55             # for last point take previous difference
56             else:
57                 diff_point = point-prev_point
58
59             # check if signal dropped by 10%
60             # check if signal rises again
61             # check if a signal is being recorded
62             if (point-bias < (max_point-bias)/1.1 and
63                 diff_point > sigma_noise/4 and
64                 signal_registered):
65
66                 # add one signal point at the beginning of the peak
67                 if len(extracted_peak) > 1:
68                     extracted_peak.insert(0, signal[peak_start])
69
70                 # delete extracted peak if threshold is not surpassed
71                 # reset variables
72                 if max(extracted_peak) < thresh:
73                     extracted_peak = []
74                     signal_registered = False
75                     peak_start = 0
76
77                 continue
78
79             # if threshold is surpassed and length is above minimum length
80             # save peak, else delete peak
81             else:
82                 if len(extracted_peak) > min_length_of_peak:
83                     extracted_peaks.append([peak_start, extracted_peak])
84                 else:
85                     extracted_peak = []
86                     signal_registered = False
87                     peak_start = 0
88
89                 continue
90
91     # reset variables for next iteration
```

```
92     extracted_peak = []
93     signal_registered = False
94     peak_start = 0
95
96     # check if signal is below threshold
97     # check if a signal is being recorded
98     elif point < thresh and signal_registered:
99
100         # add one signal point at the beginning and the end of the peak
101         if len(extracted_peak) > 1:
102             extracted_peak.insert(0, signal[peak_start])
103             extracted_peak.append(signal[i])
104
105         # delete extracted peak if threshold is not surpassed
106         # reset variables
107         if max(extracted_peak) < thresh:
108             extracted_peak = []
109             signal_registered = False
110             peak_start = 0
111
112         continue
113
114     # if threshold is surpassed and length is above minimum length
115     # save peak, else delete peak
116     else:
117         if len(extracted_peak) > min_length_of_peak:
118             extracted_peaks.append([peak_start, extracted_peak])
119         else:
120             extracted_peak = []
121             signal_registered = False
122             peak_start = 0
123
124         continue
125
126     # reset variables for next iteration
127     extracted_peak = []
128     signal_registered = False
129     peak_start = 0
130
131     # reset variables if no signal is registered
132     elif not signal_registered:
133         extracted_peak = []
134         peak_start = 0
135
136     # save current point for next iteration
137     prev_point = point
138
139     return extracted_peaks
```

5.2.3 Top2Bottom

A summary of this method can be found in section 5.1.3. This method is used for range windows, where all data points of one scan are already present. The data points are sorted and read from highest to lowest, hence the name *Top2Bottom*. If two subsequent data points in the sorted array have adjacent indices, they are put into one peak. The algorithm only iterates to values just above the detection threshold, which results in a faster peak extraction than the previous method *Front2Back*.

The input parameters are:

- **signal:** This parameter represents the input signal coming from *create_sample_window*. In this case the range window of around 10 km. The signal is a list of 12-bit integer values.
- **sampling_rate:** Is the sampling rate of the ADU. The sampling rate is represented with a

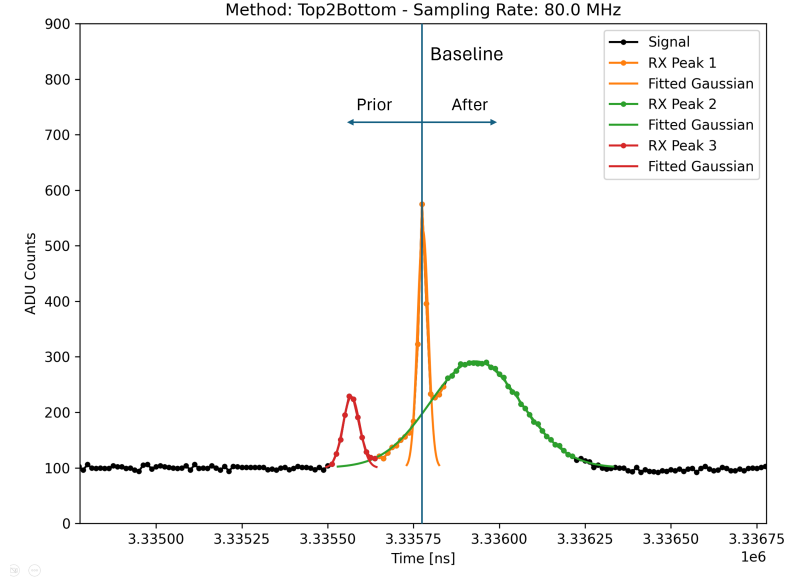


Figure 5.1: Top2Bottom baseline for highest peak, where "Prior" denotes negative indices and "After" stands for positive indices.

floating point value. It is used as a search radius for neighboring indices and peak validity parameters.

- **thresh:** This parameter sets the detection threshold for the extraction method. A higher threshold results in faster peak extraction, but may lead to peaks not being detected.
- **sigma_filter:** A globally defined standard deviation for the Gaussian filter. This variable is used in the presented way due to testing during the development phase.

Description

Similarly to the previous method, the signal gets smoothed with a Gaussian filter. The smoothing parameter is used as defined by the input parameter, in contrast to *Front2Back*. The next two lines create an array, where all ADC signals are indexed according to their position in the input array. In this case, the position in the input array is determined by the time of recording, i.e. first recorded value gets index 0, the second recorded value gets index 1, etc.

The indexed array is then filtered to include only ADC values above the detection threshold. This reduced array is then sorted from highest to lowest ADC value. Subsequently, indices are extracted into the *working_array*. Lines 33 to 35 define variables used in the main loop of this method. Lines 41 to 45 define variables for the single peak extraction loop.

The loop ranging from lines 48 to 77 handles the extraction of one single peak. For multiple peak extraction, the loop from lines 38 to 94 is used. The single peak extraction utilizes the Python function *enumerate* to get the index, together with the value of the working array. The first value in the *working_array* sets the baseline for the single peak extraction. This value represents the index of the highest signal in the range window. From that index the relative index to the baseline is computed, negative indices represent values recorded before the baseline index and positive indices represent indices recorded after the baseline index (see figure 5.1).

The sigma criterion in line 59 is empirically determined and sets the search radius for adjacent indices. The higher the sampling rate the larger the search radius gets, in terms of indices. Lines 62 to 73 determine if an index is valid for the current peak. The variables *neg_counter* and *pos_counter* are counters for how many indices have already been assigned to the current peak, the former counts indices below the baseline and the latter counts indices above the baseline.

These counters extend the search radius for each assigned index, accounting for the divergence of indices from the baseline as peaks are extracted further. When the current index is within the index search radius its value is set to 0.

The next step is to check whether the point is 0 and appends the value from the working array, corresponding to the *point* variable, to *one_peak_indices*. Further, the signal value of the investigated index is added to the list *one_peak_amps*. When all entries in *working_array* are checked, the single peak extraction is finished.

The extracted signal is used to delete the matching indices from the *working_array*. This new array is then used for the subsequent peak extraction. The indices of the extracted signal are then newly arranged to be in the correct order. These are then saved to the collection of peak indices. The main loop is exited, if no values are present in the *working_array*.

This approach tends to extract an actual single peak as several peaks. This behavior is countered by uniting intersecting arrays with the method *unite_intersecting_arrays*. It iterates over the list of arrays and checks every array for potential intersections. With the numpy method *intersect1d* common elements of two arrays are checked and returned. If there are any common elements the two arrays are united with the numpy method *union1d*. Then both arrays are deleted from the input array and the new array is added. *unite_intersecting_arrays* is called recursively until no arrays are overlapping.

The final step is to get the ADC values according to their indices. The signal gets transformed from a list to a numpy array. This array can then be reduced to the desired values via indexing with the extracted indices *one_peak_indices*. To ensure the correct order of the values the reverse of lines 26 and 27 is done. The extracted signal gets indexed with *one_peak_indices* and then sorted according to the indices. When the *peak_array* is not empty it is defined as an extracted peak. An extracted peak is again defined as a list with the first entry being the start index and the second entry being the ADC signal of the peak. At last, the length of the peak (number of signals) is checked, and whether it is a duplicate. If either of these criteria are violated, the peak is discarded, otherwise, it is saved in the list that gets returned by this method. The return variable *extracted_peaks* has the same convention as the previous method *Front2Back*.

Code

```
1 def top2bottom(signal, sampling_rate, thresh, sigma_filter):
2     """
3     Peak extraction method
4     This method sorts the signal from highest to lowest values and
5     extracts peaks based upon adjacent indices of the maximum signal
6     index
7
8     Args:
9         signal (list): ADC signal (list of integers)
10        sampling_rate (float): ADC sampling rate
11        thresh (int): detection threshold
12        sigma_filter (float): globally defined sigma for Gaussian filter
13
14    Returns:
15        extracted_peaks (list): extracted peaks of ADC signal
16    """
17
18    # apply gaussian filter
19    smoothed_signal = gaussian_filter1d(signal, sigma=1.5*sigma_filter)
20
21    # indexing signal
22    indices = np.arange(len(smoothed_signal))
23    signal_array = np.vstack((smoothed_signal, indices)).transpose()
24
25    # sort signal from highest to lowest ADC value
26    signal_array = signal_array[signal_array[:, 0] > thresh]
27    signal_array = signal_array[np.argsort(-signal_array[:, 0])]
28
```

```

29 # get newly arranged indices
30 working_array = signal_array[:,1]
31
32 # prepare start variables for main loop
33 extracted_peaks = []
34 extracted_peak = []
35 one_peak_indicess = []
36
37 # main loop
38 while True:
39
40     # prepare start variables for working array loop
41     baseline = 0
42     neg_counter = 0
43     pos_counter = 0
44     one_peak_indices = []
45     one_peak_amps = []
46
47     # working array loop
48     for k, point in enumerate(working_array):
49
50         # set baseline for looking at adjacent indices
51         # highest signal in current peak
52         if baseline == 0:
53             baseline = point
54
55         # set index relative to baseline
56         point -= baseline
57
58         # sigma criterion for adjacent indices
59         sig_crit = sampling_rate/240
60
61         # look at indices left of the peak
62         if -1-neg_counter-sig_crit <= point <= 0:
63
64             # counter for negative indices from baseline
65             neg_counter += 1
66             point = 0
67
68         # look at indices right of the peak
69         elif 0 <= point <= 1+pos_counter+sig_crit:
70
71             # counter for positive indices from baseline
72             pos_counter += 1
73             point = 0
74
75         # add point to current peak when criteria met
76         if point == 0:
77             one_peak_indices.append(int(working_array[k]))
78
79         # Find the indices of elements in the second_array within the first_array
80         rest_indices = np.where(np.isin(working_array, one_peak_indices))
81
82         # Delete elements from first_array using the obtained indices
83         working_array = np.delete(working_array, rest_indices)
84
85         # create proper indices based on lowest to highest index detected
86         one_peak_indices = np.arange(min(one_peak_indices), max(one_peak_indices)
+1, 1)
87
88         # add indices of current peak to all detected peak indices
89         one_peak_indicess.append(one_peak_indices)
90
91         # if no samples are present close main loop
92         if working_array.tolist() == []:
93             break
94

```

```

95     # unite indices if they are overlapping
96     new_one_peak_indicess = unite_intersecting_arrays(one_peak_indicess)
97
98     # get ADC values for all extracted indices
99     for one_peak_indices in new_one_peak_indicess:
100
101         # extract signal based on indices
102         np_signal = np.array(signal)
103         reduced_signal = np_signal[one_peak_indices]
104
105         # sort signal based on index
106         peak_array = np.vstack((one_peak_indices, reduced_signal)).transpose()
107         peak_array = peak_array[np.argsort(peak_array[:, 0])]
108
109         # create extracted peak if peak_array is not empty
110         if peak_array.tolist() != []:
111             extracted_peak = [peak_array[:, 0][0].tolist(), peak_array[:, 1].tolist()]
112
113         # extract up to 10 peaks
114         if len(extracted_peaks) <= 10:
115             # add peak if criteria are met (e.g. 2 samples @ 80 MHz or more)
116             # and extracted peak is not already saved
117             if (len(extracted_peak[1]) >= sampling_rate/40 and
118                 extracted_peak not in extracted_peaks):
119                 extracted_peaks.append(extracted_peak)
120         else:
121             break
122
123     return extracted_peaks
124
125 def unite_intersecting_arrays(array_list):
126     """
127     unites intersecting arrays from list of arrays
128
129     Args:
130         array_list (list): list of arrays
131
132     Returns:
133         array_list (list): list of united arrays
134     """
135
136     # Iterate through all pairs of arrays
137     for i in range(len(array_list)):
138         for j in range(i + 1, len(array_list)):
139             # Find the common elements between the current pair of arrays
140             common_elements = np.intersect1d(array_list[i], array_list[j])
141
142             # Check if there are 1 or more common elements
143             if len(common_elements) >= 1:
144                 # Union of the two arrays
145                 united_array = np.union1d(array_list[i], array_list[j])
146
147                 # Remove the original arrays from the list
148                 array_list.pop(j)
149                 array_list.pop(i)
150
151                 # Append the united array to the list
152                 array_list.append(united_array)
153
154                 # Recursively call the function to continue checking for
155                 intersections
156                 return unite_intersecting_arrays(array_list)
157
158     # If no intersections found, return input array
159     return array_list

```

5.2.4 Curve Discussion

The basic concept of this method is described in section 5.1.4. The usage is identical to *Top2Bottom*, as a given range window is evaluated at once. This method utilizes the properties of first and second-order derivatives to extract the peaks in the given signal. Maxima are detected by looking for zero crossings in the first derivative when the derived signal drops from positive values to negative. The second derivative is used to get the inflection points of the signal. The features are saved as indices. To assign the inflection points to the correct maxima, the minimum distance between the features, in terms of indices, is determined. It is relatively stable against changing signals, which means it has practically the same computation time for different numbers of peaks. The computation time is also faster than *Front2Back* and is similar to *Top2Bottom*. A more elaborate evaluation of computation times can be found in section 7.1.

The input parameters are:

- **signal:** This parameter represents the input signal coming from *create_sample_window*. In this case the range window of around 10 km. The signal is a list of 12-bit integer values.
- **sampling_rate:** Is the sampling rate of the ADU. The sampling rate is represented with a floating point value. It is used as a search radius for neighboring indices and peak validity parameters.
- **thresh:** This parameter sets the detection threshold for the extraction method. A higher threshold results in faster peak extraction, which may lead to undetected peaks.
- **sigma_filter:** A globally defined standard deviation for the Gaussian filter. This variable is used in the presented way due to testing during the development phase.

Description

At first, the raw input signal gets stored in a separate variable *signal_w_thresh*. Values below the detection threshold get set to 0, to avoid over-extraction of small signal fluctuations. The resulting array is then smoothed with a Gaussian filter. In lines 28 and 29, the first and second derivatives of the smoothed signal are calculated and stretched. This stretching is dependent on the sampling rate. Due to higher sampling rates resulting in smaller differences between neighboring points, the derivatives are stretched.

Following that, indices are created for the smoothed signal. Line 35 defines the differential threshold, which is used to get the zero crossings in the derivatives. Afterward, the indices for values above *diff_thresh* are determined. The index array has to be adapted to the size of the derived signals. The first-order derivative of the signal has one less entry and the second-order derivative has two fewer entries than the original signal. The index array is reduced by a slicing operation, which removes the first and first two entries, respectively. The resulting index arrays *pos_diff1_indices* and *pos_diff2_indices* contain gaps, where the values of the arrays *diff1_smoothed_signal* and *diff2_smoothed_signal* are below *diff_thresh*.

To get the start of the gaps, the next step calculates the difference of the indices in the index arrays. This is done in line 46, by looking for an index difference of greater than 1, in *pos_diff_of_diff1_indices*, and returning the position of the index in *pos_diff1_indices*. The indices of the hereby gathered maxima are saved in *pos_maxima*.

The approach of looking for gaps in the index array has one major drawback, as no gap is detected when the values of *diff1_smoothed_signal* do not reach above the differential threshold *diff_thresh* after the first zero crossing. Thus lines 49 and 50 are implemented. If no gap is detected, the last entry in the index array *pos_diff1_indices* is used as the detected maximum.

The inflection points are extracted in the same fashion as the maximum points, as the gaps are detected by determining the jumps in indices in *pos_diff_of_diff2_indices*. With this technique, the points with a positive slope are identified. To get the points with a negative slope, the jumps are shifted by one index with the numpy function *roll*.

Lines 57 to 59 define variables for the assignment of the inflection points to the maxima in the subsequent for loop. The loop iterates through all detected maximum points. The closest index for the upward slope inflection point is computed by calculating the difference between the index of the inflection point and the index of the maximum *mu*. With that index, the upwards slope inflection point is gathered from *pos_ups* and the downwards slope inflection point from *pos_downs*.

The if condition in line 69 checks whether the inflection points are already used. If they are already used the peak is skipped. This could be a potential error source for skipping valid peaks. In testing, all peaks were correctly detected and no false negatives were present.

The peaks are then extracted via the indices of the inflection points. All ADC values in between two inflection points are saved as one peak. The start index of the peak is the inflection point of the positive slope. The inflection points used are added at the end of the if condition for the next iteration.

The return variable *extracted_peaks* has the same convention as the previous methods *Front2Back* and *Top2Bottom*.

Code

```
1 def curve_discussions(signal, sampling_rate, thresh, sigma_filter):
2     """
3     Peak extraction method
4     This method extracts peaks based on the first and second derivative
5     of a signal.
6
7     Args:
8         signal (list): ADC signal (list of integers)
9         sampling_rate (float): ADC sampling rate
10        thresh (int): detection threshold
11        sigma_filter (float): globally defined sigma for Gaussian filter
12
13    Returns:
14        extracted_peaks (list): extracted peaks of ADC signal
15    """
16
17    # prepare signal without threshold
18    signal_w_thresh = np.array(signal).astype(float)
19
20    # set all values below threshold to 100
21    # to reduce number of peaks due to noise
22    signal_w_thresh[signal < thresh] = 100
23
24    # apply gaussian filter
25    smoothed_signal = gaussian_filter1d(signal_w_thresh, sigma=2*sigma_filter)
26
27    # get derivatives of signal and stretch them for more precise results
28    diff1_smoothed_signal = np.diff(smoothed_signal)*sampling_rate/10
29    diff2_smoothed_signal = np.diff(np.diff(smoothed_signal))*6*sampling_rate/10
30
31    # create indices of signal
32    indices = np.arange(len(smoothed_signal))
33
34    # threshold for extracting information
35    diff_thresh = -1
36
37    # get indices of positive values of derived signals
38    pos_diff1_indices = indices[1:][diff1_smoothed_signal >= diff_thresh]
39    pos_diff2_indices = indices[2:][diff2_smoothed_signal >= diff_thresh]
40
41    # get differences between indices
42    pos_diff_of_diff1_index = np.diff(pos_diff1_indices)
43    pos_diff_of_diff2_index = np.diff(pos_diff2_indices)
44
45    # get maxima based on jumps in indices
```

```

46     pos_maxima = pos_diff1_indices[:-1][pos_diff_of_diff1_index > 1]
47
48     # if no jump in indices is found, last index is taken
49     if pos_maxima.size == 0:
50         pos_maxima = np.hstack((pos_maxima, pos_diff1_indices[-1]))
51
52     # extract inflection points of peaks
53     pos_ups = pos_diff2_indices[:-1][pos_diff_of_diff2_index > 1]
54     pos_downs = pos_diff2_indices[:-1][np.roll(pos_diff_of_diff2_index, 1) > 1]
55
56     # prepare start variables for loop
57     temp_pos_ups = []
58     temp_pos_downs = []
59     extracted_peaks = []
60
61     # get closest inflection points to maximum points
62     for mu in pos_maxima:
63         closest_up_idx = (np.abs(pos_ups - mu)).argmin()
64
65         closest_up = pos_ups[closest_up_idx]
66         closest_down = pos_downs[closest_up_idx]
67
68         # get signal values for inflection point indices
69         if closest_up not in temp_pos_ups and closest_down not in temp_pos_downs:
70
71             # get indices and values between inflection points
72             one_peak_indices = np.arange(closest_up, closest_down)
73             one_peak = signal[one_peak_indices].tolist()
74
75             # add peak to extracted peaks
76             extracted_peaks.append([closest_up, one_peak])
77
78             # save used indices to not double extract the same peaks
79             temp_pos_ups.append(closest_up)
80             temp_pos_downs.append(closest_down)
81
82         else:
83             continue
84
85     return extracted_peaks

```

5.2.5 Sorting Peaks

The peak extraction methods may not return the individual returns according to their significance, but, dependent on the method, sort the peaks based on the order of being detected and extracted. This is where *sort_peaks* becomes relevant. This function resorts extracted peaks based on the maximum ADU count in the signal, the signal length of the peak, and the position relative to the expected return position.

The first three variables in the sorting function are weighting factors for the properties of the individual peaks. The maximum factor *max_fac* controls the importance of the highest ADU, the length factor *len_fac* the importance of the peak signal length, and the position factor *index_fac* the relevancy of the position of the peak in the sorting criterion *sort_crit*. The implementation into the computation of the sorting criterion can be seen in lines 45 through 47. Before *sort_crit* is calculated, two variables have to be defined for the following loop. The for-loop iterates over all extracted peaks and starts by defining the start index of the peak and the extracted signal itself. The index of the maximum is obtained with the *index* method together with the Python method *max* applied to the signal. Next, the absolute index is acquired by adding the start index to the maximum index. By subtracting the expected return index from *abs_index* and taking the absolute of this difference, the relative difference in position of the current peak and the expected peak is computed. If the peak incidentally matches the index of the expected return the relative index is set extremely small to avoid a division by zero in the *sort_crit* calculation.

The sorting criterion is defined by:

$$\text{sort_crit} = \left| \frac{\text{max_fac} \cdot \max(\text{peak_signal}) + \text{len_fac} \cdot \text{len}(\text{peak_signal})}{\text{index_fac} \cdot \text{rel_index}} \right| \quad (5.2)$$

A temporary dictionary is defined, where the individual extracted signals are the values to their respective starting indices. This dictionary later serves the purpose of reorganizing the peaks according to their sorting criterion. The sorting criterion is incorporated into the *to_sort* list, as the list receives a pair of values for every iteration of *peaks*. The pair consists of the sorting criterion and the corresponding start index of one peak. The thereby generated list is transformed into a numpy array, which is then sorted with the numpy proprietary method *argsort* from lowest to highest according to the sorting criterion. If the list *to_sort* consists only of one entry the sorting is skipped entirely.

Now that the start indices are sorted, the resorting of the peaks can be performed. This is realized by iterating over the sorted start indices in reverse order. Therefore, the most prominent peak is the first signal in the *new_peaks* list. The *new_peaks* list is filled with the value pair *start_index* and *peak_signal*, where the first is now the sorted start index and the second is the peak signal from the temporary dictionary established in the previous for-loop.

The return of the *sort_peaks* function are the 10 most relevant peaks in the extracted peaks from the foregoing peak extraction methods.

```
1 def sort_peaks(peaks, expected_index):
2     """
3     sorting peaks according to maximum, length and position (index in signal)
4
5     Args:
6         peaks (list): list of extracted peaks
7         expected_index (float): signal index of expected return
8
9     Returns:
10        list: ten sorted peaks (from most to least prominent)
11    """
12
13    # relevancy factors for peak properties
14    max_fac = 10    # factor for maximum of peak
15    len_fac = 1     # factor for length of peak
16    index_fac = 5   # factor for position of peak
17
18    # prepare start variables for loop
19    to_sort = []
20    temp_dict = {}
21
22    # generate sorting array
23    for peak in peaks:
24
25        # get start index of peak
26        start_index = peak[0]
27
28        # get signal of peak
29        peak_signal = peak[1]
30
31        # get index of maximum in peak
32        peak_index = peak_signal.index(max(peak_signal))
33
34        # correct for peak start index
35        abs_index = start_index + peak_index
36
37        # relative index to expected index
38        rel_index = np.abs(abs_index - expected_index)
39
40        # set relative extremely small when abs_index = expected_index
41        if rel_index == 0.0:
42            rel_index = 10**-9
```

```

43
44     # set sorting criterion
45     sort_crit = np.abs(
46         (max_fac*np.max(peak_signal)+len_fac*len(peak_signal))/
47         (2*index_fac*rel_index))
48
49     # assign peak to start index
50     temp_dict[start_index] = peak_signal
51
52     # assign start index to sort criterion
53     to_sort.append([sort_crit, start_index])
54
55     # transform to numpy array
56     to_sort = np.array(to_sort)
57
58     # sort if more than one peak present
59     if len(to_sort) > 1:
60         # sort according to sorting criterion
61         sorted_indices = to_sort[np.argsort(to_sort[:, 0])]
62     else:
63         # one peak does not need sorting
64         sorted_indices = to_sort
65
66     # define list for sorted peaks
67     new_peaks = []
68
69     # iterate in reverse order over sorted indices (most to least prominent)
70     for sorted_index in sorted_indices[::-1]:
71         # get sorted start index
72         sorted_start_index = sorted_index[1]
73
74         # save peaks in sorted order
75         new_peaks.append([sorted_start_index, temp_dict[sorted_start_index]])
76
77     # return 10 most prominent peaks
78     return new_peaks[:10]

```

5.3 Signal Fitting

A further reduction in data size can be achieved by fitting the data to a predefined model and only saving the fit parameters. A simple Gaussian is used, in the form of equation 5.3, to reduce the fitting procedure's computational cost.

$$f(x) = A \cdot \exp\left(\frac{-(x - \mu)^2}{2\sigma^2}\right) + \text{offset} \quad (5.3)$$

In Python this equation is written as:

```

1 def gaussian_func(x, amp, mu, sig, offset):
2     return amp * np.exp(-(x - mu)**2 / (2 * sig**2)) + offset

```

The *curve_fit* method is incorporated into a separate function, which predefines an initial guess and boundaries for the fit. The start of the function reads the start index of the peak and extracts the peak signal as floats into a numpy array. The signal is then indexed and the indices are saved into *peak_indices*. The next step writes the signal's maximum into the variable *max_signal*.

The function *scipy_gauss* handles the peak's position approximation based on the signal sampling rate. The two highest values in the signal are used for sampling rates below 100 MHz, and the three highest values are used for all higher sampling rates, for weighted mean operations in lines 45/46 and 62/63. The top values are gathered by sorting the peak signal from lowest to highest values and saving only the last two or three entries in the array. Then the positions of these values in the unsorted array are saved to *top2_indices* or *top3_indices*. From these indices,

the values from the unsorted array are stored in *top_signal*. The estimation of μ of the Gaussian is then calculated with *top2_indices* or *top3_indices* with *top_signal* as weights for the indices.

The standard deviation estimation is set to 1, representing one sample point and reflecting a nominal pulse behavior. The complete initial guess is set in line 69 and is a list, that consists of all parameters for model 5.3.

The fitting method is aborted if the maximum signal or the μ of the Gaussian is negative, because these values would result in an error when *curve_fit* is executed. When the initial guess is valid the boundaries of the fit are set. These values are chosen within a narrow range to reduce the computational cost, while ensuring sufficient margin within the boundaries, so that the correctness of the fit is not impeded. The boundaries are tuples with lower and upper bound, that are joined together into a numpy array, as seen in line 81.

The actual fit, with initial guess, boundaries, and maximum function evaluation *maxfev* of 10000, is performed in lines 84 to 88. The resulting model parameters, amplitude, μ , σ and fitted bias, are extracted in line 91 and returned together with the start index of the peak, which concludes the fitting method *scipy_gauss*.

```

1 def scipy_gauss(peak, bias, time_interval):
2     """
3     fits Gaussian to extracted peak
4
5     Args:
6     peak (list): extracted peak with start index and signal
7     bias (float): bias level of signal
8     time_interval (float): time between sample points
9
10    Returns:
11    amp (float): amplitude of Gaussian peak
12    mu (float): mean of Gaussian peak
13    sigma (float): standard deviation of Gaussian Peak
14    bias2 (float): fitted bias level
15    peak_start (int): start index of peak
16    """
17
18    # get start of peak
19    peak_start = peak[0]
20
21    # get signal of peak as floats
22    peak_signal = np.array(peak[1]).astype(float)
23
24    # create indices of signal
25    peak_indices = np.arange(len(peak_signal), step=1)
26
27    # maximum of signal
28    max_signal = np.max(peak_signal)
29
30    # if sampling rate is low, take less sample points
31    if time_interval > 10:
32        # sort for highest values
33        sorted_peak = np.sort(peak_signal)
34
35        # get 2 highest values
36        top2 = sorted_peak[-2:]
37
38        # get indices of highest signals
39        top2_indices = np.where(np.isin(peak_signal, top2))[0]
40
41        # get highest signal
42        top_signal = peak_signal[top2_indices]
43
44        # calculate weighted mean of indices
45        peak_index_for_fit = np.average(
46            top2_indices, weights=top_signal).astype(float)
47
48    else:

```

```

49     # sort for highest values
50     sorted_peak = np.sort(peak_signal)
51
52     # get 3 highest values
53     top3 = sorted_peak[-3:]
54
55     # get indices of highest signals
56     top3_indices = np.where(np.isin(peak_signal, top3))[0]
57
58     # get highest signal
59     top_signal = peak_signal[top3_indices]
60
61     # calculate weighted mean of indices
62     peak_index_for_fit = np.average(
63         top3_indices, weights=top_signal).astype(float)
64
65     # minimum one sample
66     sigma_guess = 1
67
68     # Initial guess for mean and standard deviation
69     initial_guess = [max_signal, peak_index_for_fit, sigma_guess, bias]
70
71     # return zeros for implausible initial guess
72     if initial_guess[0] <= 0 or initial_guess[1] <= 0:
73         return 0, 0, 0, 0, 0
74
75     # Set bounds for parameters
76     amp_bounds = (max(peak_signal-bias)*0.9, max(peak_signal-bias)*1.1)
77     mu_bounds = (peak_index_for_fit*0.5, peak_index_for_fit*2)
78     sigma_bounds = (0.1, 200)
79     offset_bounds = (bias-0.001, bias+0.001)
80
81     bounds = np.array([amp_bounds, mu_bounds, sigma_bounds, offset_bounds]).T
82
83     # Fit the Gaussian to the data using curve_fit
84     params, covariance = curve_fit(gaussian_func,
85                                   peak_indices,
86                                   peak_signal,
87                                   p0=initial_guess,
88                                   maxfev=10000, bounds=bounds)
89
90     # extract the parameters
91     amp, mu, sigma, bias2 = params
92
93     return amp, mu, sigma, bias2, peak_start

```

5.4 Resulting Data Packets

5.4.1 Basic Packet Structure

Each data package of a different compression stage, created by the extraction methods and fitting procedure, described in chapter 5, has an individual packet structure. The initial two packets within each package (Timestamp and Package ID) are identical and are summarized as follows:

All packets start with a timestamp with 62-bit resolution. This timestamp is constructed on the UNIX timestamp, which is then multiplied by 10^{10} to achieve a time interval of 1/10 ns to map data packets to orbital positions accurately. Depending on the tracking accuracy this time interval can be chosen smaller or larger. The 62-bit reserved for this timestamp is sufficient to record the timestamp until the end of the planned extended mission, which is the 1st May, 2028. (bep (2024)). By subtracting the launch date from the UNIX timestamp 2 bits can be saved, while maintaining the ability to record timestamps across the mission timeframe and possible extensions up to 16th May, 2033. The second packet is the Packet ID which indicates the packet type, RX, TX, or undefined. The ID for undefined packets may be used for additional data packets implemented

in later versions of the simulation scripts. Because the round trip of one signal takes some time, the instrument does not send any data packets for certain time slots. These time slots may be used to send housekeeping or additional scientific data, such as albedo measurements similar to BELA's albedo mode, described by Thomas et al. (2021). The other data packets differ for each data package and are further described in the following sections.

Range Window Packet

The first data packet is the simplest and most raw signal feasible. All compression ratios depicted in this thesis are based on this data packet, as it is the most uncompressed version of the LiDAR signal.

The data packets following the Timestamp and Packet ID packets are the Signal Index, Signal Length, Signal, and Bias. Signal Index is a 22-bit packet, that represents the starting index of the range window for the LiDAR signal. Together with the sampling rate, the time of the sample points of the signal can be determined. The 22-bit size is adequate for distances of around 7850 km with a sampling rate of 80 MHz and 628 km with a sampling rate of 1000 MHz. Additional target distances can be seen in table 5.1.

Sampling Rate [MHz]	Distance to Target [km]
80	7858.88
100	6287.10
200	3143.55
300	2095.70
400	1571.78
500	1257.42
600	1047.85
700	898.16
800	785.89
900	698.57
1000	628.71

Table 5.1: Maximal possible distance to target depending on the sampling rate with a 22-bit Signal Index.

The Signal Length parameter depicts the number of signal samples in the range window. The size of 16-bit was chosen to accommodate a range window of 9861 km, as this is the highest difference possible on Mercury. The 16-bit range is necessary for a sampling rate of 1000 MHz, for lower sampling rates a shorter range is possible. The Signal packet is a collection of all samples in the range window with the length of the Signal Length value, where each sample point is 12-bit in size. The last 14-bit of the Range Window package is the signal bias. To get an adequate resolution this specific size is used to have an accuracy up to two decimal places. The bias signal is multiplied by a factor of 100 to achieve this resolution. Depending on the processing of the signal prior to packaging, this accuracy would be redundant if the signal has only an integer resolution and can be set smaller. The complete package structure can be seen in table 5.2.

Samples Packet

The basic structure of the Samples Packet is the same as the range window packet with the difference being that the signal is subdivided into smaller packages that only contain the relevant peaks of the waveform signal. The number of smaller packages can vary depending on the terrain of the target surface. Therefore, the plural of the Signal Index is used, namely Signals Index, as it denotes the start of the range window for all subsequent peak range windows. The size of 22-bit is the same as the Signal Index of the range window packet, because it describes the same parameter for this packet. The packet Signals Length indicates the number of sample packets in the complete

Range Window	Timestamp	Packet ID	Signal Index	Signal Length	...
Bits	62-bit	2-bit	22-bit	16-bit	...

	x Signal Length	
...	Signal	Bias
...	12-bit	14-bit

Table 5.2: Packet structure in bits for a single range-window-packet. The "Signal Length" value denotes the number of 12-bit signals present.

data package. Up to 10 peaks can be detected by the extraction methods and therefore 4 bits are more than sufficient. The decoded value of Signals Length is then used to decode the subpackets for each peak. These subpackets start with the Signal Length, which relates to the number of samples in the Signal packet and is the same as for the range window package. The Signal Index differs from the previous Signal Index packet, as this parameter now describes the sample index of the first sample in the signal relative to the range window. Thus the absolute sample index of the start of the signal is the sum of the Signals Index and the subpacket Signal Index. The Signal packet is smaller, as it contains only relevant samples for one peak per subpacket. The overall samples package is finalized with the 14-bit bias packet. The complete structure can be seen in table 5.3.

Samples	Timestamp	Packet ID	Signals Index	Signals Length	...
Bits	62-bit	2-bit	22-bit	4-bit	...

	x Signals Length			
			x Signal Length	
...	Signal Length	Signal Index	Signal	Bias
...	16-bit	22-bit	12-bit	14-bit

Table 5.3: Packet structure in bits for a single samples-packet. The "Signals Length" value denotes the number of extracted peaks. For each extracted peak the singular "Signal Length" indicates the total of samples present for the peak.

Fit Packet

This data package starts in the same way as the samples data package, with Timestamp, Packet ID, Signals Index, and Signals Length. The parameters work the same as described in section 5.4.1. Signals Length indicates the number of fit parameter sets in the data package. The fit parameters consist of a 12-bit Amplitude, 26-bit Mu, 26-bit Sigma, and a 14-bit Bias. The Amplitude has the same size as a regular sample point. The position of the peak (μ) and the σ parameter, both are multiplied by a factor of 1000 to achieve a precision up to 3 decimal places when the bitstream gets decoded. The accuracy in seconds or meters depends on the sampling rate of the signal, for 80 MHz this results in a subsampling resolution of 0.0125 ns or 3.7 mm, and for 1 GHz a resolution of 0.001 ns or 0.3 mm can be achieved. Lastly, the 14-bit bias is added to each subpacket separately, because this particular bias parameter is the fitted bias corresponding to the fit parameters in the subpacket.

Fit	Timestamp	Packet ID	Signals Index	Fit Parameters Length	...
Bits	62-bit	2-bit	22-bit	4-bit	...

x Fit Parameters Length				
...	Amplitude	μ	σ	Bias
...	12-bit	26-bit	26-bit	14-bit

Table 5.4: Packet structure in bits for a single fit-packet. The "Fit Parameters Length" value denotes the number of fit parameters of extracted peaks.

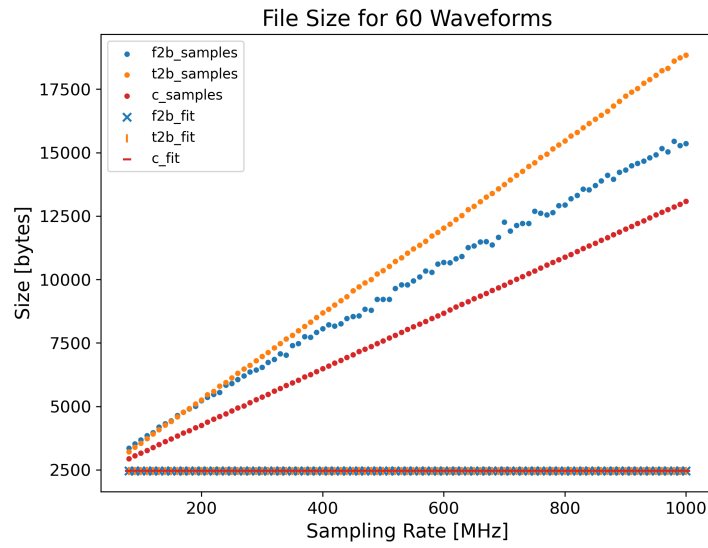


Figure 5.2: File sizes of data packets with 60 waveforms for single returns according to their sampling rate.

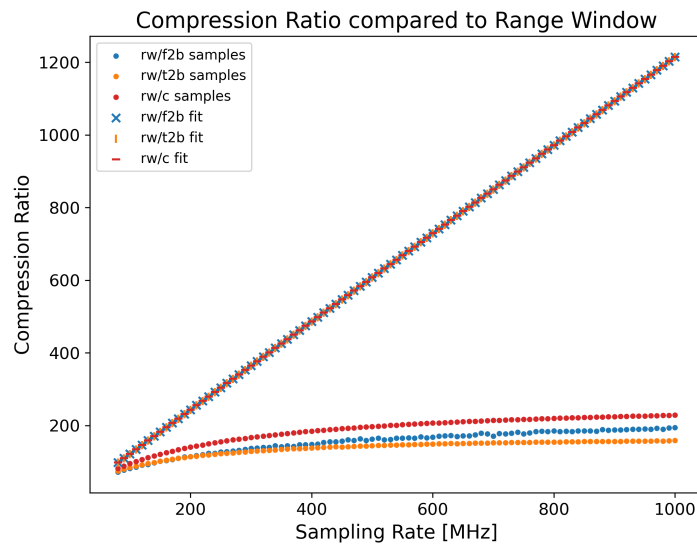


Figure 5.3: Compression ratios of data packets with 60 waveforms for single returns according to their sampling rate. The compression ratio is based on the file size of the range window packet in relation to the compressed data packets.

6. Lossless Compression

There are several ways to reduce the size of data for faster and less costly data transmission from a space-borne scientific instrument. A key concept for lossless data compression is entropy of data, described in section 4.1, that indicates the complexity, or in other words, the surprise present in a given dataset. The lower the surprise for a particular dataset, the more the packaging of the scientific data can be reduced. Depending on the use case for space applications, the computational cost and specific data transmission requirements, several approaches have to be considered, due to the limiting factors of space-grade hardware. The compression algorithms need to be tailored to the particular dataset, as there is no one-trick solution to the problem of lossless data compression, neither for space applications nor terrestrial or airborne LiDAR as well as day-to-day data such as images, video, or audio.

Lossless compression is typically achieved through various techniques, including entropy coding such as variable-length coding (e.g., Huffman coding) and arithmetic coding, as well as methods like run-length encoding and dictionary-based compression. Before applying such algorithms to the dataset, other ways to reduce the entropy may be implemented. For this thesis differential coding is put into practice, as it can be realized straightforward without having much impact on the timeline of this work. Differential coding presents a suitable entropy-reduction method, for time series data or in data where there is a clear pattern over a given timeframe.

The subsequent sections describe potentially suitable compression algorithms, based on the data present in the LiDAR data packets.

6.1 Huffman Coding

This technique, developed by David A. Huffman in 1952, is one of the earliest entropy coding schemes and works by assigning variable-length codewords to individual data points according to their frequency in a given dataset.

At the beginning of a Huffman coding algorithm, the frequency of occurrence of each data point is determined. With this frequency information, a Huffman tree is established, where each leaf node is a data point and the path to that data point on the binary represents the codeword for said data point. To build a Huffman tree a greedy process is used, where the two least occurrent data points get identified and removed from the dataset, leaving a combined internal node with the weight of the sum of the two weights of the removed data points. This procedure is performed $n - 1$ times, where n denotes the number of individual data points until only one node is present (Moffat (2019)).

As the Huffman tree is established, the assignment of the codewords to the data points can be made. The codewords are generated by following the path on the tree that leads to the data points. Each node of the tree has a right and a left path and as the path to the data point is followed the codeword gets assigned either a 1 or a 0. This leads to the creation of distinct variable-length binary codes for each data point, with shorter codewords assigned to frequently occurring data and longer codewords assigned to less common data points.

The dataset gets encoded by replacing the initial data point with the Huffman codeword, resulting in the compressed representation of the data. The decoding of this compressed dataset is performed by traversing the generated Huffman tree according to the encoded sequence of bits.

Each bit determines whether to move to the left or the right child of the current node starting from the root of the tree. When the last bit of the codeword is read, the leaf containing the initial data point is reached.

Although Huffman coding is a typical method used in image compression, traditional implementation limits the practical application of real-time transmission and processing, due to a minimum of 2 iterations over the complete dataset. A more suitable approach for real-time compression can be achieved by dynamically updating the Huffman tree (Zhan (2023)).

6.2 Golomb-Rice Coding

Golomb-Rice coding is based on Golomb coding, which is a variable-length coding scheme, similar to Huffman coding, but instead of treating individual symbols, Golomb code deals with integer values of binary representations. It is based on the probability of these values, where small values are more likely than bigger ones. The relation between the probability and the size of a particular value indicates a divisor. This divisor is utilized to acquire the quotient and remainder, that are then concatenated into a single binary string, where the quotient is in unary notation and the remainder in simple binary notation, separated by a stop bit. As an example a divisor of 4 for a value of 9 results in a quotient of 2 (unary: 00) and a remainder of 1 (binary: 01). The stop bit is 1 and thus resulting in the codeword 00101 (Golomb (1966)). A Golomb-Rice code is a Golomb code with a divisor of a power of two (Anderson (2005)).

6.3 Arithmetic Coding

The key concept of arithmetic coding is based on the probability of certain values in the data stream. Ideally, often occurring symbols or data points are substituted by smaller codewords represented with a floating point number (Ottensamer (2023)).

Arithmetic coding starts by calculating the frequency of occurrence of each data point in the data stream, similar to Huffman coding. This probability is then summed up to get the cumulative probabilities of all data points, which are mapped to a range typically from 0 to 1. This range is subdivided into intervals corresponding to the individual cumulative probabilities of each data point (Langdon (1984)). A visual representation can be seen in figure 6.1. Here, the word "Hello" is encoded using an arithmetic coding approach. The probabilities for each individual character are as follows:

Char.	Probability	Cumulative Prob.
H	0.2	0.2
e	0.2	0.4
l	0.4	0.8
o	0.2	1

Table 6.1: Probabilities and cumulative probabilities of characters in the word "Hello".

Since the input starts with an "H", the range for the next symbol must be between 0 and 0.2. For the next letter, another table is created with the new range established by the former letter and intervals adjusted to this range according to the cumulative probabilities. This procedure is carried out until no symbol remains, resulting in a final range of 0.0672 to 0.688 for the input sequence "Hello".

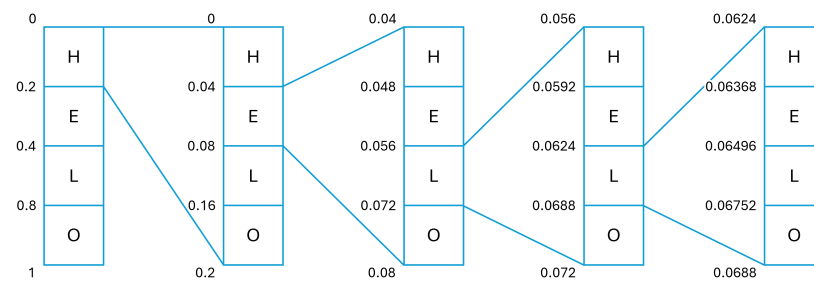


Figure 6.1: Arithmetic coding with the word "Hello". The codeword in this case is some number in the interval ranging from 0.0672 and 0.688, such as 0.68

7. Results

7.1 Peak Extraction Evaluation

The evaluation of the aforementioned extraction methods is solely based on the computing time and the accurate representation of the information found in the raw signal data, for distinct datasets. The first dataset consists of single return pulses with sampling rates ranging from 80 MHz to 1 GHz. The second dataset is a collection of multiple deterministic return pulses, as described in section 3.3.10 and with the same sampling rate range as the single return dataset. The distance stays the same for all simulation pulses. The methods are called one after another to ensure similar conditions for each peak extraction, without any interference from one method to the other. In total, there are 60 waveforms for each sampling rate that get processed and the mean of all computing times for one sampling rate is calculated.

7.1.1 Front2Back

Figure 7.1 shows a graphical representation of the extraction method Front2Back. The waveform in this graphic contains three distinct returns. A major advantage of this method is the real-time extraction of peaks from a continuous data stream. The multiple return detection, in particular the detection of overlapping peaks, leads to complications of the parametrization of the method. If the parameters for extracting subsequent peaks are too lenient, it may lead to the masking of peaks by preceding signals. On the other hand, if the parameters are too strict, it may lead to an over-extraction of peaks, as illustrated by RX peak 4 in figure 7.1. The over-extraction is dependent on the noise present in the signal due to the succeeding peak detection criterion. The criterion is based on the rise after the signal dropped by 10% of the peak's maximum point.

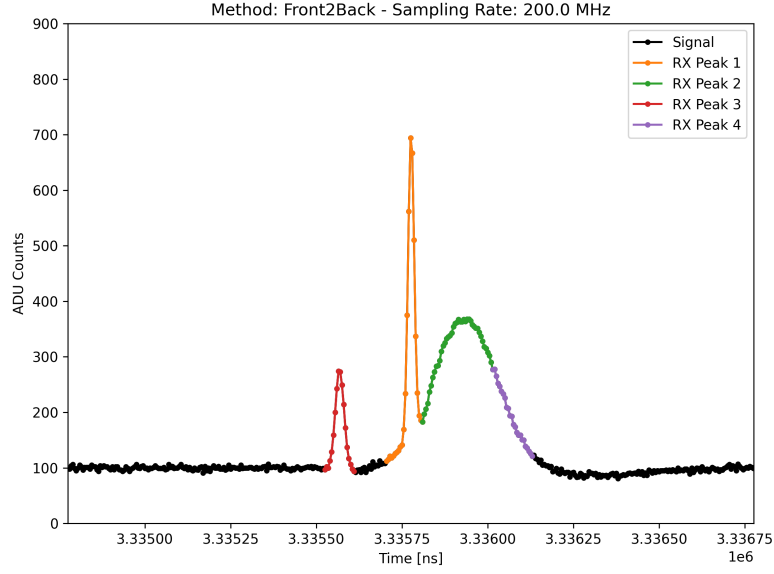


Figure 7.1: Front2Back peak extraction method with three returns present in the signal with four extracted peaks. The method shows real-time possibility with slight compromises in accuracy.

The computation time shown in figure 7.6 is extraordinarily high compared to the other methods. This is due to the use of a range window. While other methods only work for range windows, this method can be used for range windows and real-time peak extraction when the ADC generates the signal data points without the need to complete a full range window. The advantages of this method are for instruments that operate without a range gate. The range gate sets the start and duration, i.e. length of the range window for data acquisition. An in-depth analysis could be made with a special version of the LiDAR-Data-Simulator, where the extraction method is directly implemented in the data generation algorithm, where the compression occurs directly after the ADU values are created. To maintain the seriousness of the schedule of this master's thesis, emphasis was placed on the recreation of the BELA instrument properties that utilize the aforementioned range gate.

7.1.2 Top2Bottom

The result of the method Top2Bottom is visualized in figure 7.2. The signal is the same as the previous method Front2Back with three returns in the waveform. Top2Bottom achieves a precise peak extraction with low computational costs. Opposite to Front2Back this method needs the complete sample set of one measurement at once, in the form of a range window, to work effectively. Considering the order of the peaks, if the requested sorting order is from highest to lowest peak, the list of extracted peaks is already sorted after the method is applied. This leads to a major reduction in computational cost compared to Front2Back or Curve Discussion. The over-extraction occurring with Front2Back is non-existent, which allows for proper data fitting and thus higher compression rates.

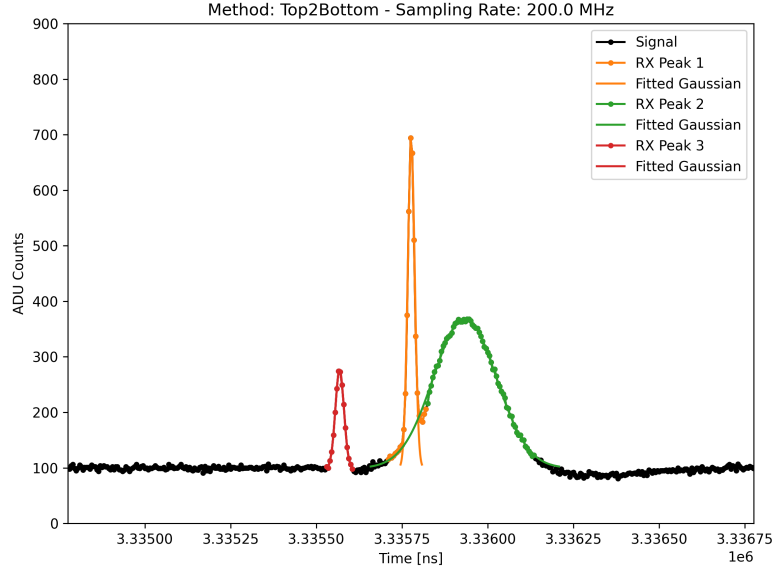


Figure 7.2: Top2Bottom peak extraction method with three returns present in the signal. Three peaks have been determined with minimal excess data points included in the extracted peak. The method shows fast and precise peak extraction for predefined sample windows. The accuracy of this method also allows for peak fitting for further data reduction. The fitted Gaussians are presented as Fitted Gaussian corresponding to the same colored RX Peak.

Top2Bottom is developed based on reducing the amount of data points that have to be evaluated by the compression algorithm. The values of the waveform signal are read from highest to lowest. The number of values to be examined is reduced by following a peak signal until the detection threshold is reached. Therefore the size of the range window impacts the computation time insignificantly. Figure 7.6 shows a notable gain in performance compared to Front2Back. The performance of Top2Bottom is similar to Curve Discussion which is illustrated in figures 7.8, 7.11 and 7.12. There, a similar trend can be seen between both methods. A detailed comparison of all methods' single and multiple return extraction time performance can be found in section 7.1.4.

7.1.3 Curve Discussion

This method is inspired by the Gaussian decomposition approach by Lindner et al. (2015). The working principles are shown in figure 7.3, with the same waveform presented in the previous sections. Similarly to Top2Bottom, Curve Discussion requires a complete range window in the way it is implemented at this point. On-the-fly peak extraction could be possible if values are stored and evaluated simultaneously. Essential for this method are the derivatives of the waveform signal, i.e. the differences between two adjacent data points. The performance gain of the concurrent extraction has to be evaluated in further iterations of the presented work.

The inherent way the peaks are determined leads to shorter peaks being fully characterized, whereas peaks with a higher dispersion are not completely extracted. This results from the initial signal smoothing which spreads the FWHM of the Gaussian returns and therefore sets the points of inflection further from the peaks maximum. The incomplete information of longer peaks can be compensated by signal fitting. Even though the peaks are not fully extracted an appropriate fitting function can recreate the initial signal, as seen in figure 7.4.

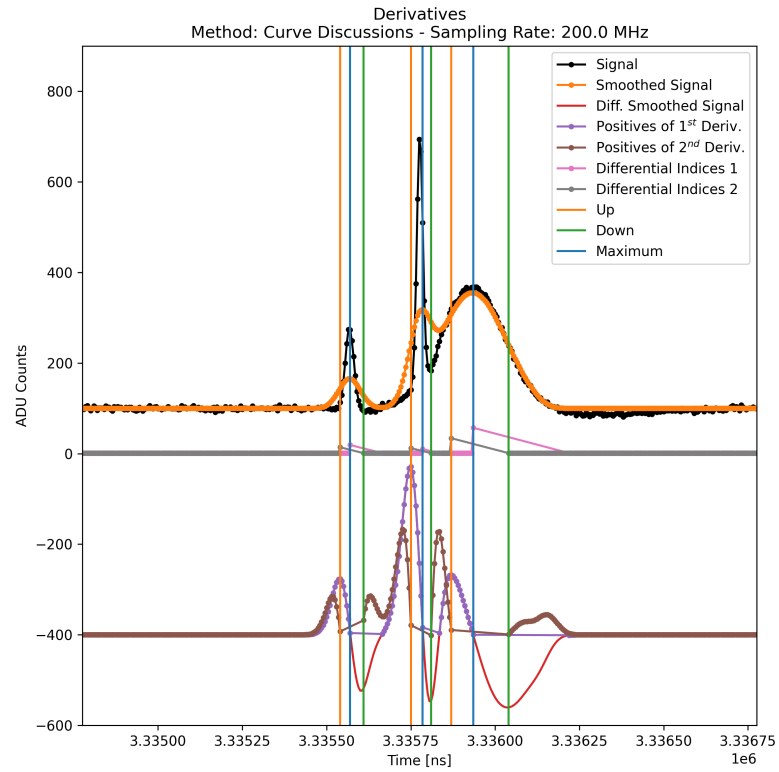


Figure 7.3: The figure contains the initial three-return signal (black) with the smoothed signal (orange), the first derivative of the smoothed signal (red), positive values of the first derivative (purple), and positive values of the second derivative (brown). From those derivatives, the index changes are displayed for the first and second derivatives, in pink and grey, respectively. The vertical lines show the inflection points of the rising signal (orange) and the falling signal (green), as well as the position of the peak maxima (blue).

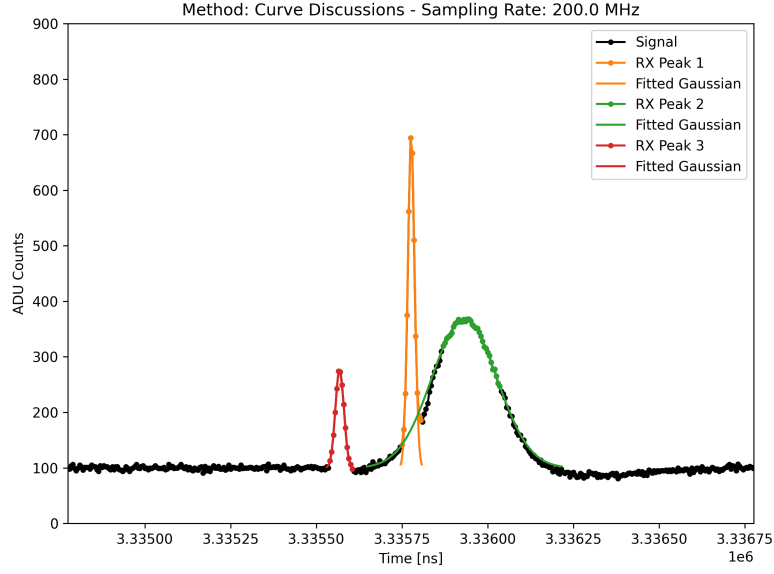


Figure 7.4: Extracted peaks of the given waveform processed with Curve Discussion. Short-duration peaks are fully extracted, while longer peaks lack completeness. The incomplete peaks can be reconstructed with an appropriate fitting procedure.

The performance in computing time is similar to Top2Bottom. The size of the range window influences the execution time of Curve Discussion as well as the storage cost of the products generated during the extraction process. The execution time compared to Top2Bottom differs substantially for single and multiple return signals. Further analysis of the behavior of different input signals can be found in the following section 7.1.4.

7.1.4 Computing Time Comparison

The computing times of the extraction methods are briefly described in the previous sections (7.1.1, 7.1.2, and 7.1.3). This part focuses on the difference in processing time between a simple single-return signal and a multiple-return signal. Figures 7.5 and 7.6 display the execution time of all extraction methods including Front2Back, which highly exceeds the computing time of the other methods. This is due to the nature of this method, which reads and evaluates every single data point in the waveform. This leads to an almost identical processing time for single-return and multiple-return signals.

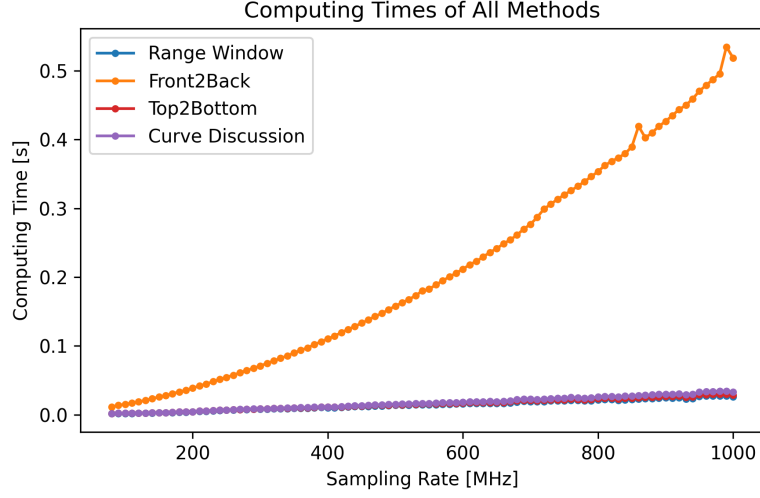


Figure 7.5: This figure shows the mean computation time of all peak extraction methods, as well as the range window extraction, for single returns for individual sampling rates. Because the range window extraction is performed prior to the actual peak extraction, the computation time of the peak extraction is added to the range window extraction time.

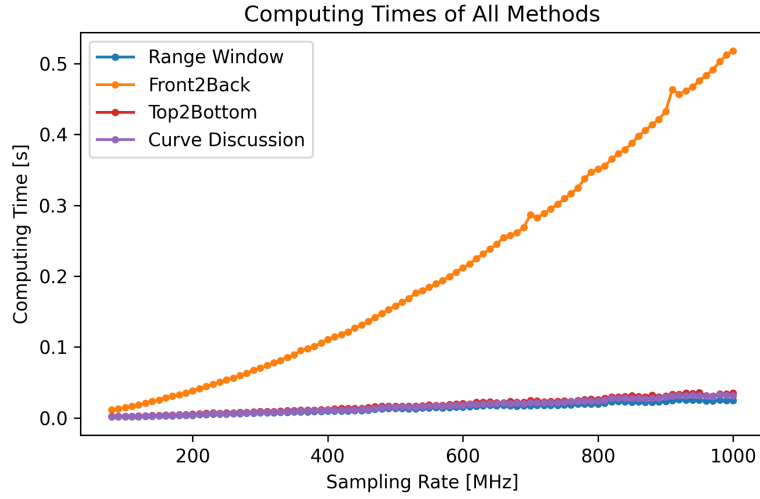


Figure 7.6: This figure shows the mean computation time of all peak extraction methods, as well as the range window extraction, for multiple returns for individual sampling rates. Because the range window extraction is performed prior to the actual peak extraction, the computation time of the peak extraction is added to the range window extraction time.

A similar trend is seen for the differences in execution time for the range window extraction. Since the range window has fixed indices for each sampling rate, the slicing of the signal array remains the same, irrespective of the complexity of the input signal. The slicing operation is only dependable on the size of the input signal. The computing times of the range window extraction, together with the Top2Bottom and Curve Discussion times are illustrated in figures 7.7 and 7.8 for single and multiple-return waveforms, respectively. Although the time complexity should lead to a steady linear increase in processing time, deviations are noticeable. These fluctuations are observed for each simulation run and are constantly changing. The simulations were conducted

on a typical higher-end consumer-grade PC running Windows 10. The deviations are attributed to background processes of the operating system. The magnitude of these fluctuations is on the order of microseconds, where minor interferences become perceptible.

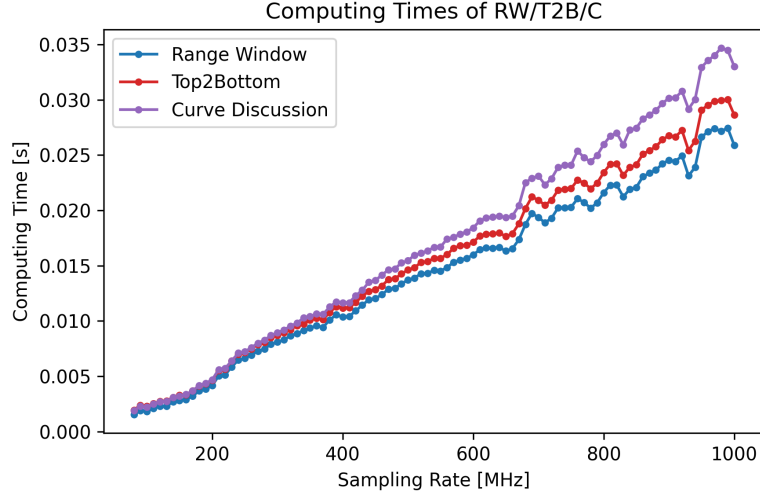


Figure 7.7: This figure shows the mean computation time of the T2B and C peak extraction methods, together with the range window extraction time, for single returns for individual sampling rates.

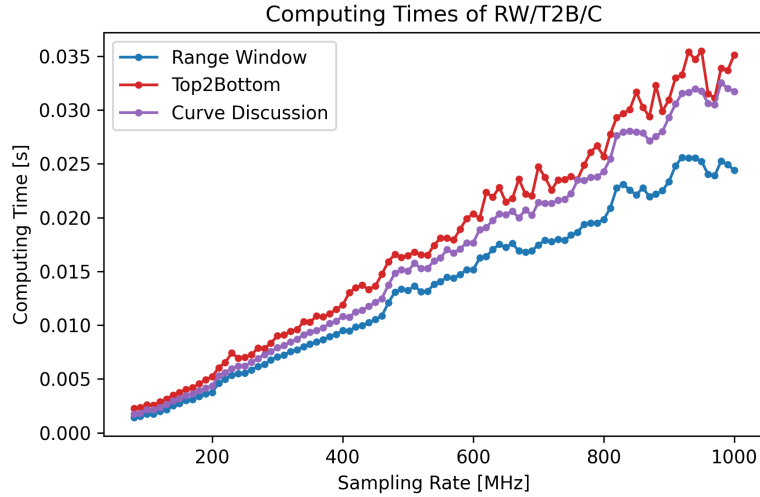


Figure 7.8: This figure shows the mean computation time of the T2B and C peak extraction methods, together with the range window extraction time, for multiple returns for individual sampling rates.

A key difference in the computing time between single-return and multiple-return signals is observed for the Top2Bottom and Curve Discussion method, as the former leads to longer run duration due to the complexity of the input signal. In contrast, Curve Discussion yields a constant run time regardless of signal complexity. The difference becomes more apparent when examining the processing times excluding the range window extraction, as depicted in figures 7.9 and 7.11. There, Top2Bottom outperforms Curve Discussion significantly for higher sampling rates. This

can be attributed to the reduction in data points to be evaluated when only one peak is present in the waveform. This behavior is disadvantageous for more complicated signals, leading to higher variability in run time and longer execution time because every return in the signal has its evaluation loop. Curve Discussion always processes the complete waveform at once, which allows for more consistency in computational time, compared to Top2Bottom. Figures 7.10 and 7.12 again show the mathematical differences of the run times for each sampling rate by subtracting the times of Top2Bottom from the times of Curve Discussion. Positive values therefore favour the usage of Top2Bottom and negative values the implementation of a Curve Discussion algorithm. These graphs thus conclude that the Top2Bottom algorithm is more suitable for simple signals. Curve Discussion is more favorable for more complex signals, occurring in rougher terrain with larger variations in elevation.

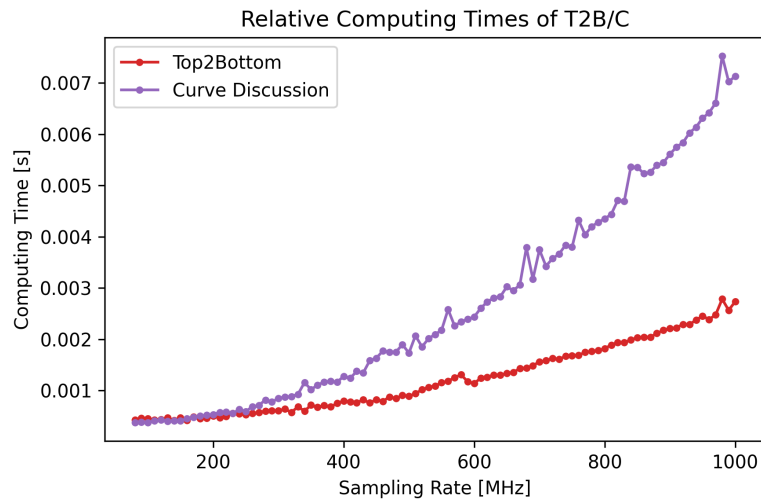


Figure 7.9: This figure shows the mean computation time of the T2B and C peak extraction methods, without the range window extraction time, for single returns.

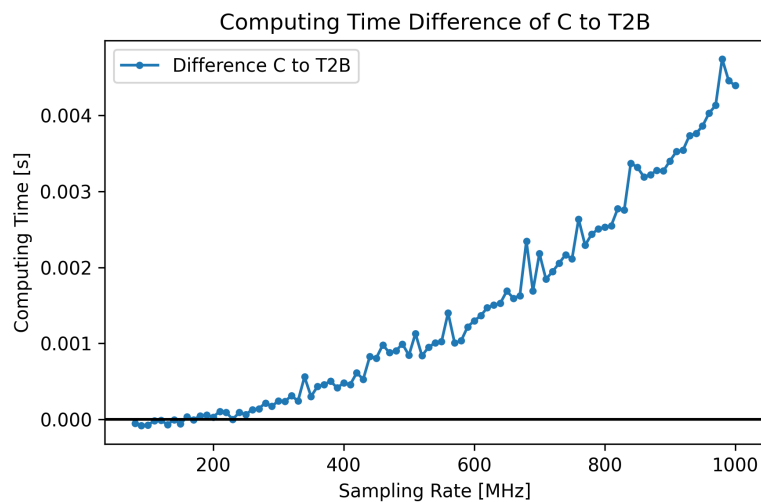


Figure 7.10: Here, the difference between the C method and the T2B method, for single returns, can be seen.

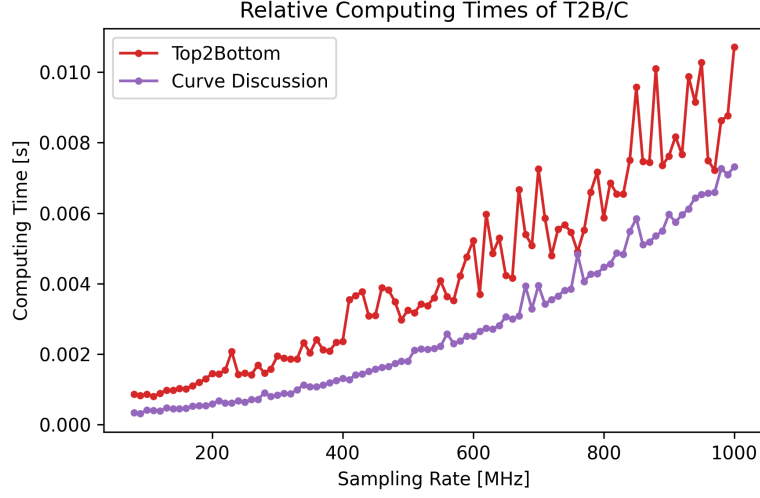


Figure 7.11: This figure shows the mean computation time of the T2B and C peak extraction methods, without the range window extraction time, for multiple returns.

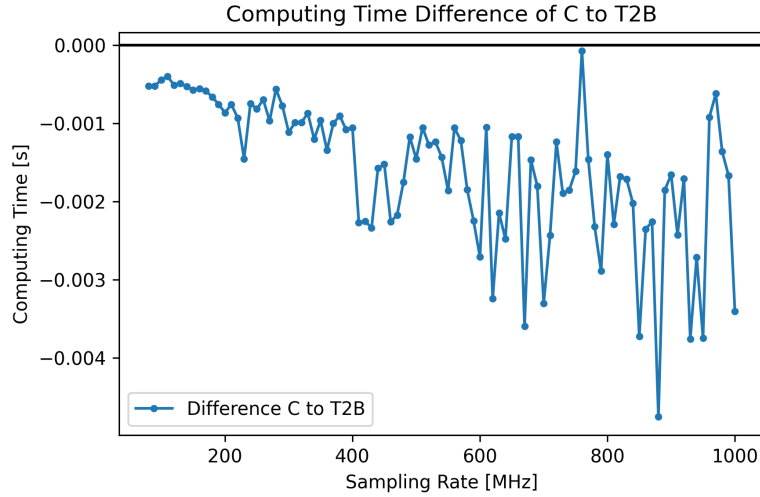


Figure 7.12: Here, the difference between the C method and the T2B method, for multiple returns, can be seen.

7.2 Compression Ratios

7.2.1 Lossy Compression

The lossy compression ratios are based on the file sizes of range window packets to samples and fit packets. Each data package consists of 60 waveforms to create an appropriate sample size to mitigate outliers due to single larger or smaller packages.

Range Window 1 km

The first table 7.1, shows the compression ratios for seven sampling rates ranging from 80 MHz to 1 GHz. The Front2Back samples packet offers the lowest compression ratio of 8.73 for 80 MHz, while Curve Discussion has the highest compression ratio out of the samples packets for 80 MHz.

In figure 7.13 the file sizes and compressions are displayed. A clear trend for Front2Back and Top2Bottom is noticeable. T2B is favorable for lower sampling rates, and F2B for faster sampling rates. However, CD exceeds both methods regarding file size and therefore compression ratio.

The fit packets are another step up compared to the samples packet. Across all sampling rates, the compression ratios of the fit packets surpass the ratios of the samples packets. At 1 GHz the compression ratio is the highest with 145.08 for all three fit packets. As can be seen, the values for all fit packets are identical for each sampling rate. This is due to the number of extracted and fitted peaks, where each peak has a fixed set of model parameters, which in this case is 4 (Amplitude, μ , σ , and Bias). The investigated return signal consists of a single return, which results in 4 parameters per return package and therefore the same compression for each fit method.

Single Return 1 km RW							
Sampling Rate [MHz]	80	100	200	400	600	800	1000
RW/F2B Samples	8.73	9.79	13.28	16.80	18.86	20.64	21.68
RW/T2B Samples	9.12	10.16	13.22	15.59	16.71	17.23	17.62
RW/C Samples	10.02	11.52	16.46	21.04	23.30	24.61	25.49
RW/F2B Fit	12.17	15.07	29.53	58.42	87.30	116.19	145.08
RW/T2B Fit	12.17	15.07	29.53	58.42	87.30	116.19	145.08
RW/C Fit	12.17	15.07	29.53	58.42	87.30	116.19	145.08

Table 7.1: Compression ratios for lossy compressed single-return waveforms and a range window of 1 km

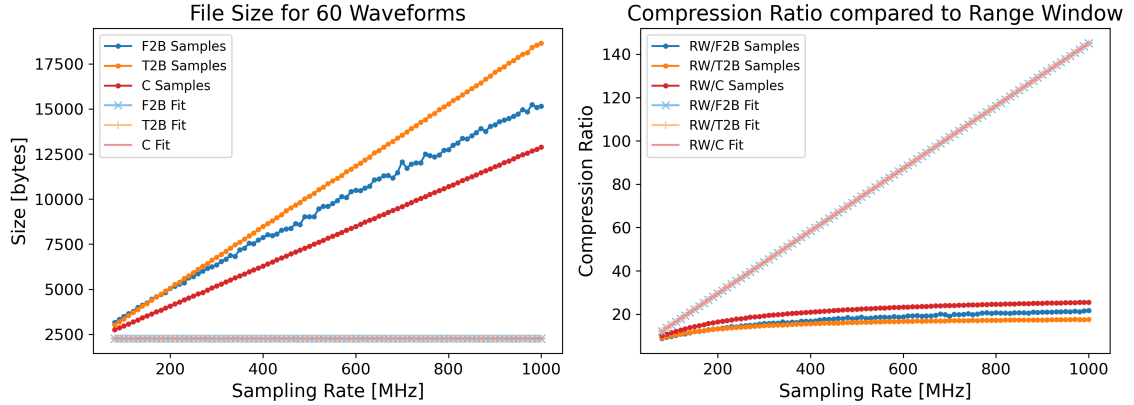


Figure 7.13: File size and total compression ratio of lossy compressed single-return waveforms with a range window of 1 km.

For multiple-return signals the compression ratios are generally lower than for single-return signals. Figure 7.14 shows a similar trend to the single return, except for a larger variance along the mean file size. F2B starts with a lower compression ratio as T2B, but increases less in size than T2B for higher sampling rates. Curve Discussion once again has the best compression ratios across all sampling rates for the samples data packet.

The fit packets differ this time from one another as different extraction methods yield different number of peaks. F2B falls behind most likely due to the over-extraction problem described in section 7.1.1. The most precise method CD achieves the highest compression ratio closely followed by T2B. The extraction methods are tuned to be more generous with the extraction to ensure the completeness of the scientific measurements. CD stands out with its preciseness with no over-extraction and is therefore the best approach to lossy data compression for this dataset. The compression ratios for this dataset are detailed in table 7.2.

Multiple Return 1 km RW							
Sampling Rate [MHz]	80	100	200	400	600	800	1000
RW/F2B Samples	3.42	3.30	3.76	4.40	4.98	5.32	5.77
RW/T2B Samples	3.68	3.66	4.10	4.39	4.59	4.61	4.82
RW/C Samples	5.18	5.39	7.08	7.64	8.59	8.85	9.59
RW/F2B Fit	7.49	7.65	11.26	17.83	26.29	34.95	43.64
RW/T2B Fit	7.76	9.66	17.09	33.40	50.16	65.47	91.51
RW/C Fit	7.98	9.79	19.47	37.56	57.23	75.54	98.74

Table 7.2: Compression ratios for lossy compressed multiple-return waveforms and a range window of 1 km

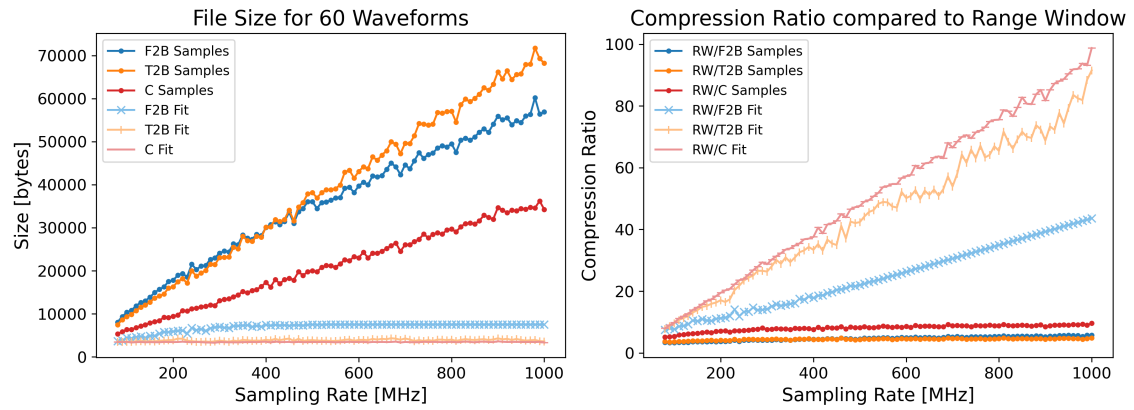


Figure 7.14: File size and total compression ratio of lossy compressed multiple-return waveforms with a range window of 1 km.

Range Window 10 km

As expected the compression ratios for a 10 km range window are roughly on the order of $10\times$ higher than the 1 km range window packets. The trends for each method and the file sizes for the extracted peaks are identical to the smaller range window. This concludes, that the increase in compression ratio directly correlates to the size of the range window. Single-return waveforms are evaluated in table 7.3 and figure 7.15. The evaluated multiple-return waveforms can be found in table 7.4 and figure 7.16.

Single Return 10 km RW							
Sampling Rate [MHz]	80	100	200	400	600	800	1000
RW/F2B Samples	76.17	86.13	118.90	151.94	171.08	187.57	197.16
RW/T2B Samples	79.60	89.33	118.33	140.97	151.54	156.58	160.27
RW/C Samples	87.42	101.37	147.36	190.20	211.38	223.59	231.86
RW/F2B Fit	106.14	132.52	264.40	528.21	791.97	1055.77	1319.53
RW/T2B Fit	106.14	132.52	264.40	528.21	791.97	1055.77	1319.53
RW/C Fit	106.14	132.52	264.40	528.21	791.97	1055.77	1319.53

Table 7.3: Compression ratios for lossy compressed single-return waveforms and a range window of 10 km

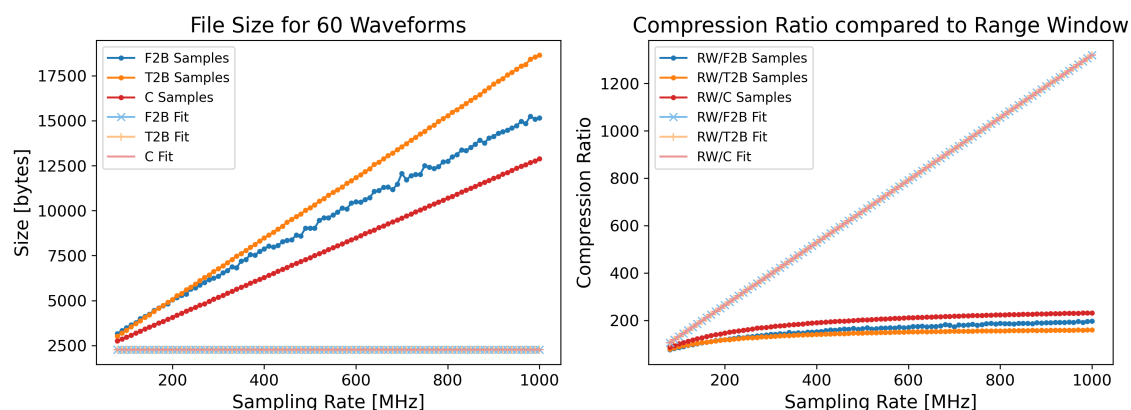


Figure 7.15: File size and total compression ratio of lossy compressed single-return waveforms with a range window of 10 km.

Multiple Return 10km RW							
Sampling Rate [MHz]	80	100	200	400	600	800	1000
RW/F2B Samples	29.80	29.03	33.65	39.80	45.18	48.31	52.51
RW/T2B Samples	32.14	32.18	36.72	39.69	41.64	41.90	43.83
RW/C Samples	45.16	47.40	63.40	69.05	77.92	80.44	87.24
RW/F2B Fit	65.35	67.32	100.80	161.17	238.51	317.57	396.91
RW/T2B Fit	67.68	84.96	153.05	301.96	455.05	594.85	832.28
RW/C Fit	69.58	86.16	174.34	339.59	519.19	686.37	898.06

Table 7.4: Compression ratios for lossy compressed multiple-return waveforms and a range window of 10 km

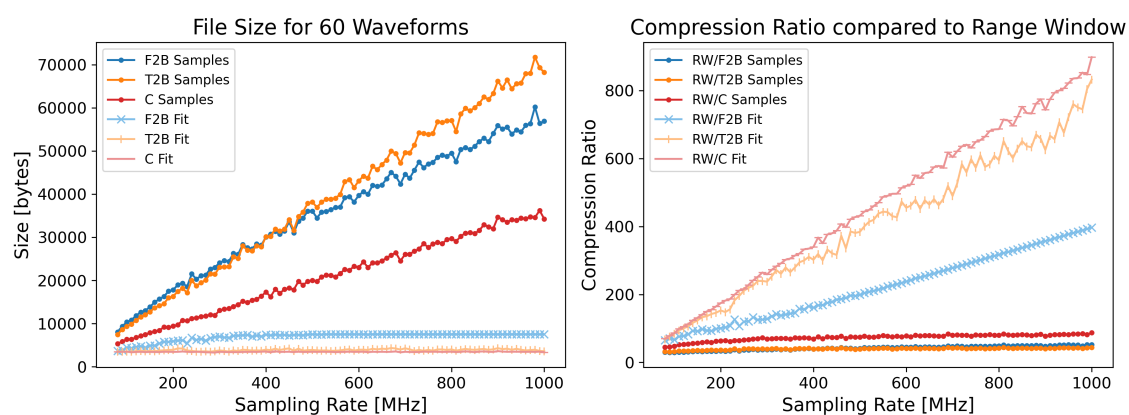


Figure 7.16: File size and total compression ratio of lossy compressed single-return waveforms with a range window of 10 km.

7.2.2 Lossless Compression

This section outlines the theoretical maximum compression achievable for the datasets presented in this thesis. As described in section 4.1, the entropy of a dataset indicates the minimum amount of bits required to represent certain data. Therefore, by dividing the current size of a parameter by the entropy of the same parameter, the maximum theoretical compression can be calculated. With the lossless compression methods covered in section 6 the data can be reduced to get closer to the entropy limit of the dataset. To achieve optimal compression results, these algorithms must be tailored to the specific data. While this customization extends beyond the scope of this work, the information on the data entropy provides a general overview of possible compression ratios.

Range Window Packet

The range window packet has up to six parameters. The timestamp packet is only present in the TX packet, which leads to six science data parameters for the TX packet and five parameters for the RX packet. While the timestamp packet has the largest size in terms of bits, the most impact on the size of a range-window packet has the signal parameter with 266.85 samples at 80 MHz and a 1 km range window and up to 33356.41 samples at 1 GHz and a 10 km range window with each sample point having 12 bits. Thus it is vital to focus on the reduction of the signal parameter.

Single Return For the generated dataset studied in this thesis, a total of 60 waveforms were analyzed. This leads to the following data and entropies for all sampling rates:

- Timestamp: 62 bit - 60 Samples - Entropy: **5.91 bits/sample** after lossy compression and **0.12 bits/sample** after decorrelation. This leads to optimal compression ratios of 10.49 and 516.67, respectively. The difference in time is at a constant value which results in such a high compression ratio. If the time interval deviates slightly, in the realm of nanoseconds, the entropy is expected to be higher.
- Packet ID: 2 bit - 120 Samples - Entropy: **0.00 bits/sample** after lossy compression and **0.12 bits/sample** after decorrelation. The entropy of just lossy compressed is 0 because the value does not change when TX and RX IDs are investigated separately. This means if a constant stream of TX and RX packets can be assured, this parameter can be omitted. If the data packets need to have a packet ID and only these two packets exist, the bit-size can be reduced to 1 bit/sample. The compression ratio is then dependent on the number of waveforms, which would be 60 for this dataset
- Signal Index: 22 bit - 120 Samples - Entropy: **0.00 bits/sample** after lossy compression and **0.12 bits/sample** after decorrelation. The TX data packet always starts with index 0 and therefore may be omitted. The RX packet is fixed for each sampling rate as it is fixed in time. If the index does not change for one complete science phase, this parameter can be reduced to a single value at the start of the to be transmitted data package, in this case the value would be used for all 60 waveforms. The compression ratio is therefore 60.
- Signal Length: 16 bit - 120 Samples - Entropy: **0.00 bits/sample** after lossy compression and **0.12 bits/sample** after decorrelation. The lengths of the TX and RX data packets always remain of the same length if no change to the range window size is made. This parameter can be handled similarly to the signal index parameter in terms of being transmitted once for an entire data package, yielding the same compression ratio as the signal index.
- Signal: 12 bit - 298252.73 Samples - Entropy: ranging from **3.61 - 5.41 bits/sample** after lossy compression and **4.11 - 5.29 bits/sample** after decorrelation. The bit-size of this parameter is directly correlated with the size of the range window. The larger the range window, the larger the number of samples for the signal parameter. The compression ratios according to the entropy ranges from **2.22 - 3.32** for just lossy compressed data and **2.27 - 2.92** for additional decorrelated data.

- Bias: 14 bit - 120 Samples - Entropy: ranging from **0.96 - 3.40 bits/sample** after lossy compression and **1.81 - 4.12 bits/sample** after decorrelation. The entropy for this parameter strongly depends on the noise of the data. The less the bias fluctuates, the more compression can be achieved. The compression ratios for this dataset range from **4.12 - 14.58** for lossy compressed data and **3.40 - 7.74** for additional decorrelated data.

Multiple Return For the multiple return waveforms, the first four parameters have the exact same result as for the single return signals. Only the signal and bias parameters differ as follows:

- Signal: 12 bit - 298252.73 Samples - Entropy: ranging from **3.89 - 5.77 bits/sample** after lossy compression and **4.14 - 5.26 bits/sample** after decorrelation. As can be seen, the entropy slightly increases or stays approximately the same which results in a slightly lower or similar compression ratio compared to the single return signals with **2.08 - 3.07** for lossy compressed data and **2.28 - 2.90** for additional decorrelated data. The behavior in regards to the size of the range window is identical.
- Bias: 14 bit - 120 Samples - Entropy: ranging from **0.84 - 3.28 bits/sample** after lossy compression and **1.46 - 4.22 bits/sample** after decorrelation. The compression ratios for this dataset range from **4.27 - 16.67** for lossy compressed data and **3.32 - 9.59** for additional decorrelated data.

Samples Packet

This packet type consists of eight parameters, where the first four are identical to the range window parameters for each extraction method. These parameters are Timestamp, Packet ID, Signals Index, and Signals Length. Only the last parameter of these four differs for multiple-return signals.

Single Return For single-return packets, the extraction methods accurately extract only a single return pulse keeping the parameter Signals Length constant for the complete data package resulting in an entropy of **0.00 bits/sample**, i.e. a compression ratio of the number of waveforms contained in the data package. In this case, the compression ratio is 60.

The rest of the entropies and compression ratios are as follows:

- Signal Index: 22 bit

	F2B	T2B	C	
Avg. Samples:	60	60	60	Samples
Entropy:				
- Just Lossy:	0.35 - 3.71	0.52 - 3.63	0.52 - 3.62	bits/sample
- Decorrelated:	0.82 - 4.36	1.04 - 4.43	1.04 - 4.34	bits/sample
Compression Ratios:				
- Just Lossy:	5.93 - 62.86	6.06 - 42.31	6.08 - 42.31	
- Decorrelated:	5.05 - 26.83	4.97 - 21.15	5.07 - 21.15	

- Signal Length: 16 bit

	F2B	T2B	C	
Avg. Samples:	60	60	60	Samples
Entropy:				
- Just Lossy:	0.35 - 3.66	0.12 - 2.45	0.12 - 1.10	bits/sample
- Decorrelated:	0.69 - 4.15	0.37 - 2.94	0.12 - 1.68	bits/sample
Compression Ratios:				
- Just Lossy:	4.37 - 45.71	6.53 - 133.33	14.54 - 133.33	
- Decorrelated:	3.86 - 23.19	5.44 - 43.24	9.52 - 133.33	

- Signal: 12 bit

	F2B	T2B	C	
Avg. Samples:	2612.97	3062.76	2054.66	Samples
Entropy:				
- Just Lossy:	6.07 - 8.00	6.31 - 8.01	6.71 - 8.93	bits/sample
- Decorrelated:	5.97 - 7.50	5.88 - 7.64	6.25 - 8.33	bits/sample
Compression Ratios:				
- Just Lossy:	1.5 - 1.98	1.50 - 1.90	1.34 - 1.79	
- Decorrelated:	1.6 - 2.01	1.57 - 2.04	1.44 - 1.92	

- Bias: 14 bit

	F2B	T2B	C	
Avg. Samples:	60	60	60	Samples
Entropy:				
- Just Lossy:	0.96 - 3.40	0.96 - 3.40	0.96 - 3.40	bits/sample
- Decorrelated:	1.81 - 4.12	1.81 - 4.12	1.81 - 4.12	bits/sample
Compression Ratios:				
- Just Lossy:	4.12 - 14.58	4.12 - 14.58	4.12 - 14.58	
- Decorrelated:	3.40 - 7.74	3.40 - 7.74	3.40 - 7.74	

The overall best compression ratios are generated by the Curve Discussion method. The Signal Index parameter has the highest possible compression with the Front2Back method. The lowest compression for Signal Index is similar for all three methods. Top2Bottom and Curve Discussion exceed F2B at the Signal Length parameter for the highest compression, and C has an overall higher compressibility except for the highest compression of T2B. The Signal data entropies closely resemble each other while C still being the best option out of the three extraction methods. The Bias entropies and compression ratios remain the same for all data packets.

Multiple Return Since multiple-return waveforms can have up to 10 return signals the entropy differs to single-return signals. The following parameters illustrate each entropy and compression ratio of the differing data of multiple-return waveforms:

- Signals Length: 4 bit

	F2B	T2B	C	Samples
Avg. Samples:	60	60	60	
Entropy:				
- Just Lossy:	0.00 - 2.84	0.00 - 2.60	0.00 - 1.51	bits/sample
- Decorrelated:	0.12 - 3.81	0.12 - 3.64	0.12 - 2.27	bits/sample
Compression Ratios:				
- Just Lossy:	1.41 - 60.00	1.54 - 60.00	2.65 - 60.00	
- Decorrelated:	1.05 - 33.33	1.10 - 33.33	1.76 - 33.33	

- Signal Index: 22 bit

	F2B	T2B	C	Samples
Avg. Samples:	301.05	142.18	120.37	
Entropy:				
- Just Lossy:	0.52 - 7.08	0.61 - 7.63	0.93 - 6.29	bits/sample
- Decorrelated:	1.13 - 8.09	1.09 - 7.77	1.53 - 6.88	bits/sample
Compression Ratios:				
- Just Lossy:	3.11 - 42.31	2.88 - 36.07	3.50 - 23.66	
- Decorrelated:	2.72 - 19.47	2.83 - 20.18	3.20 - 14.38	

- Signal Length: 16 bit

	F2B	T2B	C	Samples
Avg. Samples:	301.05	142.18	120.37	
Entropy:				
- Just Lossy:	0.57 - 6.06	0.12 - 7.27	0.12 - 6.12	bits/sample
- Decorrelated:	1.13 - 6.67	0.37 - 7.48	0.37 - 6.87	bits/sample
Compression Ratios:				
- Just Lossy:	2.64 - 28.07	2.20 - 133.33	2.61 - 133.33	
- Decorrelated:	2.40 - 14.16	2.14 - 43.24	2.33 - 43.24	

- Signal: 12 bit

	F2B	T2B	C	Samples
Avg. Samples:	10627.3	12282.63	6247.78	
Entropy:				
- Just Lossy:	6.09 - 8.41	6.40 - 8.35	6.78 - 8.47	bits/sample
- Decorrelated:	4.74 - 7.19	4.56 - 7.32	4.94 - 7.84	bits/sample
Compression Ratios:				
- Just Lossy:	1.43 - 1.97	1.44 - 1.88	1.42 - 1.77	
- Decorrelated:	1.67 - 2.53	1.64 - 2.63	1.53 - 2.43	

- Bias: 14 bit

	F2B	T2B	C	Samples
Avg. Samples:	60	60	60	
Entropy:				
- Just Lossy:	0.84 - 3.28	0.84 - 3.28	0.84 - 3.28	bits/sample
- Decorrelated:	1.46 - 4.22	1.46 - 4.22	1.46 - 4.22	bits/sample
Compression Ratios:				
- Just Lossy:	4.27 - 16.67	4.27 - 16.67	4.27 - 16.67	
- Decorrelated:	3.32 - 9.59	3.32 - 9.59	3.32 - 9.59	

The Signals Length and Bias values appear as often as a TX signal is produced, thus a total of 60 entries occur in this dataset. Signal Index and Signal Length appear an equal amount across one data package for a single extraction method. F2B has the most data points with approximately 5 per waveform, followed by T2B with 2.34, and C with roughly 2 data points per waveform. The highest possible compression for Signal Index can be seen at F2B. Interestingly, decorrelation seems to increase the entropy resulting in lower possible compression. The Signal Length parameter for each extracted peak has the overall highest possible compression of all parameters for T2B and C.

The largest set of data points is again the Signal parameter. While F2B extracts the most peaks, the amount of samples is less than T2B, indicating the extracted peaks have fewer data points for an individual extracted peak. C keeps the least number of samples, thus having the highest lossy compression. For lossless compression, the extraction methods are mostly equal. Decorrelation achieves an additional reduction of entropy increasing the compressibility from approximately 1.97 to 2.53 for F2B, 1.88 to 2.63 for T2B, and 1.77 to 2.43 for C. The Bias data is identical for all extraction methods and decorrelation does not help reduce the entropy favoring entropy coding to the initial data.

Fit Packet

This packet is similar to the Samples packet as it also contains eight parameters, where the first four (Timestamp, Packet ID, Signals Index, and the previously called Signals Length, Fit Parameters Length) and the last parameter (Bias) are synonymous.

Single Return Since the Fit packet is a derivation of the Samples packet, the Signals Length and Fit Parameters Length describe the same property, namely the number of extracted peaks. As this parameter does not change for all single-return signals the compression ratio is equal to the number of TX/RX pairs, i.e. 60 for this dataset.

- Fit Parameters Length: 4 bit

	F2B	T2B	C	Samples
Avg. Samples:	60	60	60	
Entropy:				
- Just Lossy:	0.00 - 0.00	0.00 - 0.00	0.00 - 0.00	bits/sample
- Decorrelated:	0.12 - 0.12	0.12 - 0.12	0.12 - 0.12	bits/sample
Compression Ratios:				
- Just Lossy:	60.00 - 60.00	60.00 - 60.00	60.00 - 60.00	
- Decorrelated:	33.33 - 33.33	33.33 - 33.33	33.33 - 33.33	

- Amplitude: 12 bit

	F2B	T2B	C	
Avg. Samples:	60	60	60	Samples
Entropy:				
- Just Lossy:	5.03 - 5.81	4.98 - 5.76	4.98 - 5.76	bits/sample
- Decorrelated:	5.24 - 5.83	5.26 - 5.83	5.31 - 5.83	bits/sample
Compression Ratios:				
- Just Lossy:	2.06 - 2.39	2.08 - 2.41	2.08 - 2.41	
- Decorrelated:	2.06 - 2.29	2.06 - 2.28	2.06 - 2.26	

- μ : 26 bit

	F2B	T2B	C	
Avg. Samples:	60	60	60	Samples
Entropy:				
- Just Lossy:	5.77 - 5.91	5.77 - 5.91	5.77 - 5.91	bits/sample
- Decorrelated:	5.82 - 5.91	5.82 - 5.91	5.82 - 5.91	bits/sample
Compression Ratios:				
- Just Lossy:	4.40 - 4.51	4.40 - 4.51	4.40 - 4.51	
- Decorrelated:	4.40 - 4.47	4.40 - 4.47	4.40 - 4.47	

- σ : 26 bit

	F2B	T2B	C	
Avg. Samples:	60	60	60	Samples
Entropy:				
- Just Lossy:	5.42 - 5.91	5.42 - 5.91	5.42 - 5.91	bits/sample
- Decorrelated:	5.51 - 5.91	5.51 - 5.91	5.51 - 5.91	bits/sample
Compression Ratios:				
- Just Lossy:	4.40 - 4.80	4.40 - 4.80	4.40 - 4.80	
- Decorrelated:	4.40 - 4.72	4.40 - 4.72	4.40 - 4.72	

- Bias: 14 bit

	F2B	T2B	C	
Avg. Samples:	60	60	60	Samples
Entropy:				
- Just Lossy:	0.96 - 3.32	0.96 - 3.32	0.96 - 3.32	bits/sample
- Decorrelated:	1.59 - 4.11	1.59 - 4.11	1.59 - 4.11	bits/sample
Compression Ratios:				
- Just Lossy:	4.22 - 14.58	4.22 - 14.58	4.22 - 14.58	
- Decorrelated:	3.41 - 8.80	3.41 - 8.80	3.41 - 8.80	

The signal data in this case is transferred to the fit parameters Amplitude, μ , and σ . All extraction methods provide exactly one peak for each TX pulse yielding the same number of Amplitudes, μ s, and σ s across this dataset. In an ideal scenario, the entropies are equal for each extraction method. F2B seems to differ slightly for the Amplitude parameter as the entropies are ever so slightly higher compared to the other two methods. The entropies and compression

ratios for the positional parameter μ are identical for each method with compression ratios of 4.4 to 4.51. The width parameter of the signal peak σ is synonymous to μ as the compression ratios are 4.4 to 4.8 across all methods. Decorrelation cannot improve the compression for all described parameters in this case.

Multiple Return As expected, the Fit Parameters Length varies for multiple-return signals. The number of data points is equal to the number of TX signals. In the multiple-return dataset it is still possible, that the number of extracted peaks is constant, resulting in a compression ratio equal to the number of signals in the data package, which is 60 in this case.

- Fit Parameters Length: 4 bit

	F2B	T2B	C	
Avg. Samples:	60	60	60	Samples
Entropy:				
- Just Lossy:	0.00 - 2.84	0.00 - 2.60	0.00 - 1.51	bits/sample
- Decorrelated:	0.12 - 3.81	0.12 - 3.64	0.12 - 2.27	bits/sample
Compression Ratios:				
- Just Lossy:	1.41 - 60.00	1.54 - 60.00	2.65 - 60.00	
- Decorrelated:	1.05 - 33.33	1.10 - 33.33	1.76 - 33.33	

- Amplitude: 12 bit

	F2B	T2B	C	
Avg. Samples:	301.05	142.18	120.37	Samples
Entropy:				
- Just Lossy:	5.02 - 7.86	5.04 - 7.44	5.00 - 7.23	bits/sample
- Decorrelated:	5.24 - 8.11	5.26 - 7.55	5.26 - 7.41	bits/sample
Compression Ratios:				
- Just Lossy:	1.53 - 2.39	1.61 - 2.38	1.66 - 2.40	
- Decorrelated:	1.48 - 2.29	1.59 - 2.28	1.62 - 2.28	

- μ : 26 bit

	F2B	T2B	C	
Avg. Samples:	301.05	142.18	120.37	Samples
Entropy:				
- Just Lossy:	5.77 - 9.02	5.77 - 8.09	5.77 - 7.63	bits/sample
- Decorrelated:	5.83 - 9.22	5.83 - 8.07	5.83 - 7.62	bits/sample
Compression Ratios:				
- Just Lossy:	2.88 - 4.51	3.21 - 4.51	3.41 - 4.51	
- Decorrelated:	2.82 - 4.46	3.22 - 4.46	3.41 - 4.46	

- σ : 26 bit

	F2B	T2B	C	
Avg. Samples:	301.05	142.18	120.37	Samples
Entropy:				
- Just Lossy:	5.43 - 9.22	5.43 - 8.08	5.43 - 7.64	bits/sample
- Decorrelated:	5.59 - 9.22	5.59 - 8.10	5.59 - 7.65	bits/sample
Compression Ratios:				
- Just Lossy:	2.82 - 4.79	3.22 - 4.79	3.40 - 4.79	
- Decorrelated:	2.82 - 4.65	3.21 - 4.65	3.40 - 4.65	

- Bias: 14 bit

	F2B	T2B	C	
Avg. Samples:	301.05	142.18	120.37	Samples
Entropy:				
- Just Lossy:	0.84 - 3.49	0.80 - 3.50	0.81 - 3.44	bits/sample
- Decorrelated:	0.50 - 4.15	0.60 - 4.15	0.63 - 4.15	bits/sample
Compression Ratios:				
- Just Lossy:	4.01 - 16.67	4.00 - 17.50	4.07 - 17.28	
- Decorrelated:	3.37 - 28.00	3.37 - 23.33	3.37 - 22.22	

Regarding the fit parameters, the number of data points depends once again on the extraction method. F2B has the most samples with an average of 301.04 samples for 60 waveforms, followed by T2B with 142.18, and C with 120.37 samples. For each individual fit parameter, the entropies and compression ratios are mostly equal for each extraction method, only differing marginally at sampling rates with the highest entropy. All parameter entropies are indifferent to decorrelation, again favoring the entropy coding on the initial dataset to reduce computational cost.

7.3 Accuracy

To ensure the consistency and scientific value of the compressed dataset the distance measurements were carried out on the decoded data packets created with the previously described compression methods. The measured distances as well as the theoretical uncertainty of the measurements can be seen in figure 7.17. The left side of the plot shows distances measured solely based on the maximum signal data point. The grey line indicates the exact distance to the target, a data point would fall on that line, if the distance is perfectly in line with the sampling rate, i.e. it takes the light a non-fractional amount of samples to travel 1000 km (500 km distance to target). The orange lines show the maximum uncertainty according to the sampling rate. A sampling rate of 80 MHz results in an uncertainty of approximately ± 1.874 m. The right side of the figure shows the distances measured from fitted signals, significantly reducing measurement errors in most cases. The outlier at 140 MHz may be attributed to a sample shift in the digitized waveform, which shifts the signal roughly one sample point. This could be due to a pre-existing shift in the BELA dataset and therefore a false generation of the new waveform, but the possibility of a general error in the LiDAR data generator cannot be ruled out either.

Altogether, no major deviance in the data generation and compression could have been detected. The data points outside of the uncertainty lines do not hold for higher sampling rates due to the present noise in the data stream. Therefore, the here presented distance measurements hold true for the used simulation parameters, as well as the presented lossy compression algorithms.

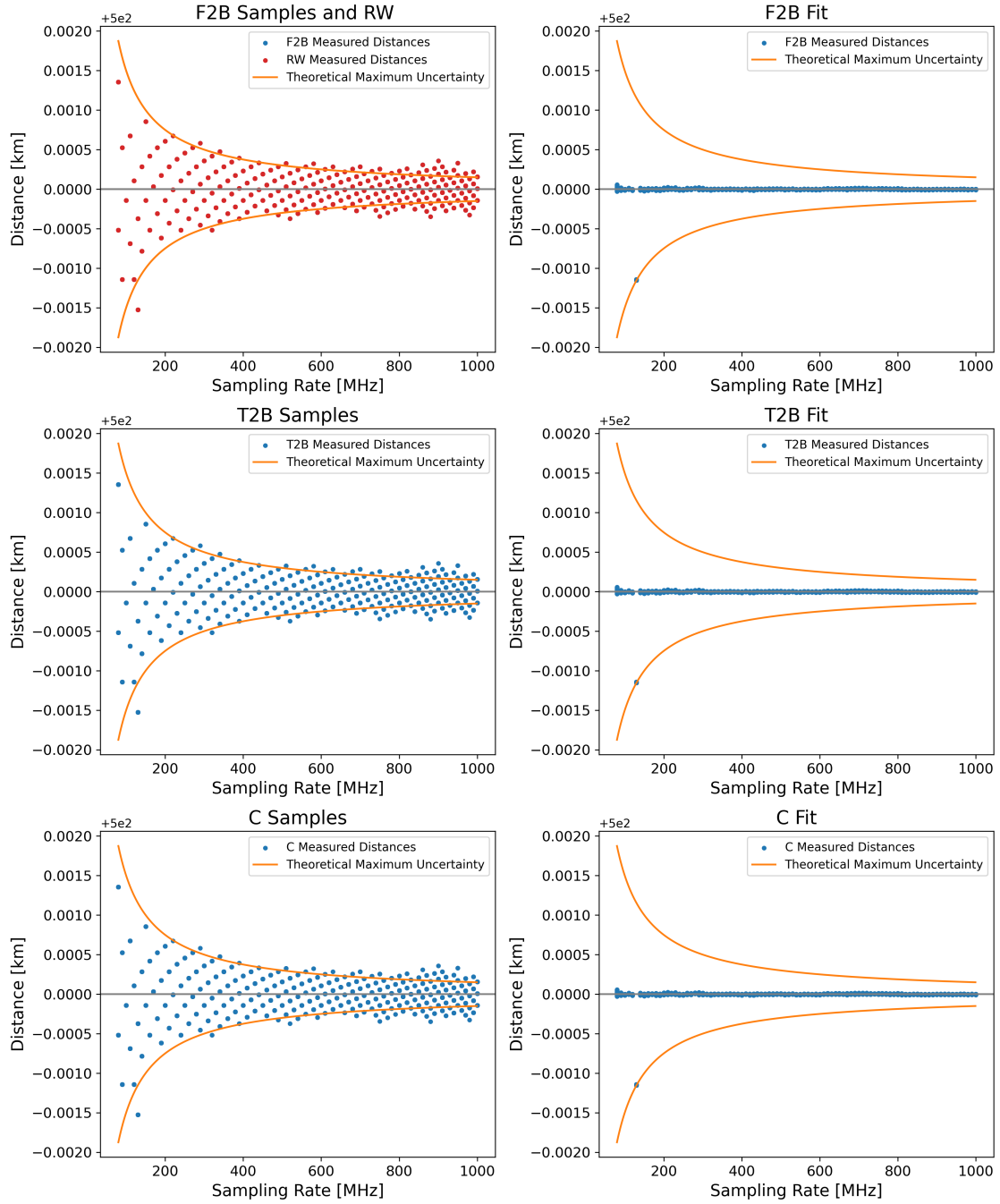


Figure 7.17: Distances and uncertainties of the measurements. The grey line indicates the distance to the target surface. The orange lines indicate the maximum uncertainty due to the sampling rate of the ADC. The left side shows the distances solely based on the peak signal of the waveform, while the right side shows the distances measured from the fitted waveforms.

8. Conclusion

The described LiDAR data generator together with the three presented extraction methods show different approaches to LiDAR data acquisition and compression of scientific measurements of Light Detection and Ranging instruments found in current space-based missions.

Although laser altimeters are already frequently used and have been implemented into several space-borne science missions, where the amount of data is relatively low, recent developments of terrestrial and air-borne LiDAR systems produce vast amounts of data points, the understanding of data handling becomes increasingly significant for space exploration.

Current LiDAR in-space technologies are vastly less powerful than current terrestrial and air-borne laser scanners. The rapid advancements of LiDAR instruments will sooner or later make their way into space. Due to the limitations of flight-approved hardware for space missions, data handling and transmission play a critical role regarding the amount of scientific value a single instrument can deliver. The on-ground often overlooked aspect of data acquisition is the management of storage and data amount, which must be carefully handled for space missions.

This thesis shows potential strategies for different datasets and sampling rates, particularly higher sampling rates than current space-based LiDAR. All in all, the best and most robust techniques are the Top2Bottom and Curve Discussion extraction methods with fitting procedures applied after the initial peak detection. CD leads in overall computational efficiency, however, F2B could be a better use for on-the-fly peak detection where no range window can be provided. The approach of implementing a range window (RW) is a fast and easy technique to quickly reduce the amount of data produced by the ADC of the instrument. The narrower the RW can be set, the faster the subsequent peak detection can be fulfilled. The resulting compression ratios are dependable on the initial size of the range window and the parameters that eventually get sent back to the ground. Therefore, the lossy and lossless compression ratios have to be evaluated separately. Concludingly it can be said, that the best practice for LiDAR instruments is to choose the appropriate size of the range window, so that all expected peaks are recorded, where the peaks are then extracted with a CD-esque algorithm and subsequently fitted with an appropriate fit-algorithm suitable for the specific hardware of the instrument.

8.1 Future Prospects

The information provided by this thesis sets the basis for an implementation of a lossless compression algorithm, as presented in section 6. The hereby determined algorithms could then be adapted to the specific hardware (i.e. Data Processing Unit, FPGA, etc.) of future LiDAR instruments with faster sampling rates and higher scan rates, resulting in a more detailed elevation model of the planetary bodies examined by future space missions.

Acknowledgements

First and foremost I want to thank my parents and my sister who always supported me during my journey through university. Special thanks to my supervisors who agreed to let me carry out this thesis and have helped with their fundamental knowledge of data compression, coding expertise, and for providing me with the necessary literature. I want to thank Armin Luntzer for looking into the air-borne LiDAR data, provided by Skyability, in the programming language C.¹

Thanks to Skyability for providing me with air-borne LiDAR data and additional resources such as images, and for helping me understand the practical aspects of LiDAR technology.

I also want to thank my fellow students Matthias Stallovits and Robert D. Anczyck for keeping me on track during our studies and providing vital insights and ideas from organizational to thesis-specific matters.

Thank you to my friends, Patrick Graßl and Matthias Knaak for their discussions, outside opinions, and suggestions on different topics and for improving my Python programming skills.

Last but not least, I want to thank Dagmar Fister for having an open ear and keeping me grounded during the course of this master's thesis.

¹The decoded data was unfortunately already compressed and had to be discarded, as patent laws protect further insights into the airborne LiDAR instruments and therefore were of no use in this thesis.

Bibliography

- Andreas Fix, Giampiero Naletto, Ian Hutchinson, Nikos Karafolas, Wolfgang Riede, Arnaud Hellere, Bob Menzies, and Haris Riris. Space lidar and space optics. *CEAS Space Journal*, 11(4):359–362, Dec 2019. ISSN 1868-2510. doi: 10.1007/s12567-019-00288-2. URL <https://doi.org/10.1007/s12567-019-00288-2>.
- Garmin. Garmin on mars. <https://www.garmin.com/en-US/blog/general/garmin-on-mars/>, 04 2021. Accessed: 2024-05-07.
- Keigo Enya, Masanori Kobayashi, Jun Kimura, Hiroshi Araki, Noriyuki Namiki, Hirotomo Noda, Shingo Kashima, Shoko Oshigami, Ko Ishibashi, T. Yamawaki, Kazuyuki Tohara, Yoshifumi Saito, Masanobu Ozaki, Takahide Mizuno, Shunichi Kamata, Koji Matsumoto, Sho Sasaki, Kiyoshi Kuramoto, Yuki Sato, Takeshi Yokozawa, Tsutomu Numata, Satoko Mizumoto, Hiroyuki Mizuno, Kenta Nagamine, Akihiko Sawamura, Kazuo Tanimoto, Hisato Imai, Hiroyuki Nakagawa, Okiharu Kirino, David Green, Masayuki Fujii, Satoru Iwamura, Naofumi Fujishiro, Yoshiaki Matsumoto, Kay Lingenauber, Reinald Kallenbach, Christian Althaus, Thomas Behnke, Jan Binger, Anna Dauriskikh, Henri Eisenmenger, Ulrich Heer, Christian Hüttig, Luisa M. Lara, Alexander Lichopoj, Horst-Georg Lötze, Fabian Lüdicke, Harald Michaelis, Juan Pablo Rodriguez Garcia, Kerstin Rösner, Alexander Stark, Gregor Steinbrügge, Pascal Thabaut, Nicolas Thomas, Simone del Togno, Daniel Wahl, Belinda Wendler, Kai Wickhusen, Konrad Willner, and Hauke Hussmann. The ganymede laser altimeter (gala) for the jupiter icy moons explorer (juice): Mission, science, and instrumentation of its receiver modules. *Advances in Space Research*, 69(5):2283–2304, 2022. ISSN 0273-1177. doi: <https://doi.org/10.1016/j.asr.2021.11.036>. URL <https://www.sciencedirect.com/science/article/pii/S0273117721008851>.
- N. Thomas, H. Hussmann, T. Spohn, L. M. Lara, U. Christensen, M. Affolter, T. Bandy, T. Beck, S. Chakraborty, U. Geissbuehler, M. Gerber, K. Ghose, J. Gouman, S. HosseiniArani, K. Kuske, A. Peteut, D. Piazza, M. Rieder, A. Servonet, C. Althaus, T. Behnke, K. Gwinner, C. Hüttig, R. Kallenbach, A. Lichopoj, K. Lingenauber, H. G. Lötze, F. Lüdicke, H. Michaelis, J. Oberst, R. Schrödter, A. Stark, G. Steinbrügge, S. del Togno, K. Wickhusen, J. M. Castro, M. Herranz, J. Rodrigo, H. Perplies, T. Weigel, S. Schulze-Walewski, S. Blum, A. Casciello, E. Rugi-Grond, W. Coppoolse, M. Rech, K. Weidlich, T. Leikert, R. Henkelmann, B. Trefzger, and B. Metz. The BepiColombo Laser Altimeter. *Space Science Reviews*, 217(1):25, February 2021. doi: 10.1007/s11214-021-00794-y.
- Rob Cole. The ddl difference: Comparing direct detect lidar and doppler lidar for meteorology. <https://www.nrgsystems.com/blog/the-ddl-difference-comparing-direct-detect-lidar-to-doppler-lidar/>, 05 2019. Accessed: 2024-05-07.
- ESA. Lasers in space. https://www.esa.int/Applications/Observing_the_Earth/Aeolus/Lasers, 2022. Accessed: 2024-05-07.
- Skyability GmbH. High-resolution point cloud of a terrestrial laser scan, 2021.
- Skyability GmbH. Aerial point cloud of a sub-urban area, 2020.

Skyability GmbH. Bathymetric laser scan, 2022.

USGS Astrogeology Science Center. Mercury messenger global dem 665m. https://astrogeology.usgs.gov/search/map/mercury_messenger_global_dem_665m, 2016.

Nicolas Thomas, Hauke Hussmann, and L. Lara. The bepicolombo laser altimeter (bela): a post-launch summary. *CEAS Space Journal*, 11, 08 2019. doi: 10.1007/s12567-019-00270-y.

K. J. Becker, M. S. Robinson, T. L. Becker, L. A. Weller, K. L. Edmundson, G. A. Neumann, M. E. Perry, and S. C. Solomon. First Global Digital Elevation Model of Mercury. In *47th Annual Lunar and Planetary Science Conference*, Lunar and Planetary Science Conference, page 2959, March 2016.

Hauke Hussmann, Kay Lingenauber, Reinald Kallenbach, Keigo Enya, Nicolas Thomas, Luisa M. Lara, Christian Althaus, Hiroshi Araki, Thomas Behnke, Jose M. Castro-Marin, Henri Eisenmenger, Thomas Gerber, Miguel Herranz de la Revilla, Christian Hüttig, Ko Ishibashi, Jaime Jiménez-Ortega, Jun Kimura, Masanori Kobayashi, Horst-Georg Lötze, Alexander Lichopoj, Fabian Lüdicke, Ignacio Martínez-Navajas, Harald Michaelis, Noriyuki Namiki, Hirotomo Noda, Jürgen Oberst, Shoko Oshigami, Juan Pablo Rodríguez García, Julio Rodrigo, Kerstin Rösner, Alexander Stark, Gregor Steinbrügge, Pascal Thabaut, Simone del Togno, Kazuyuki Touhara, Sebastian Villamil, Belinda Wendler, Kai Wickhusen, and Konrad Willner. The ganymede laser altimeter (gala): key objectives, instrument design, and performance. *CEAS Space Journal*, 11(4):381–390, Dec 2019. ISSN 1868-2510. doi: 10.1007/s12567-019-00282-8. URL <https://doi.org/10.1007/s12567-019-00282-8>.

Michael Fraczek, Andreas Behrendt, and Nikolaus Schmitt. Short-range optical air data measurements for aircraft control using rotational raman backscatter. *Optics express*, 21:16398–414, 07 2013. doi: 10.1364/OE.21.016398.

Thomas Beck. *Modeling and verification of BELA’s thermal behavior and range finder performance*. Phd thesis, Universität Bern, March 2012.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

Alireza Hosseiniarani, Stefano Bertone, Daniel Arnold, Alexander Stark, Hauke Hussmann, T. Beck, Clémence Herny, Antoine Pommerol, Adrian Jäggi, and Nicolas Thomas. Comprehensive in-orbit performance evaluation of the bepicolombo laser altimeter (bela). *Planetary and Space Science*, 195:105088, 09 2020. doi: 10.1016/j.pss.2020.105088.

Gaku Nishiyama, Alexander Stark, Christian Hüttig, Hauke Hussmann, Klaus Gwinner, Ernst Hauber, L. Lara, and Nicolas Thomas. Simulation of laser pulse shapes received by the bepicolombo laser altimeter (bela): Implications for future constraints on surficial properties of mercury, 01 2022.

multiprocessing — process-based parallelism. <https://docs.python.org/3/library/multiprocessing.html>, 03 2024. Accessed: 2024-03-16.

C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

- Khalid Sayood. Introduction to Data Compression. Morgan Kaufmann Publishers, San Francisco, CA, USA, second edition, 2000.
- Gaussian filter. http://wiki.awf.forst.uni-goettingen.de/wiki/index.php/Gaussian_filter, 06 2021. Accessed: 2024-03-21.
- Robert R. Lindner, Carlos Vera-Ciro, Claire E. Murray, Snežana Stanimirović, Brian Babler, Carl Heiles, Patrick Hennebelle, W. M. Goss, and John Dickey. Autonomous gaussian decomposition. The Astronomical Journal, 149(4):138, mar 2015. doi: 10.1088/0004-6256/149/4/138. URL <https://dx.doi.org/10.1088/0004-6256/149/4/138>.
- Bepicolombo factsheet. https://www.esa.int/Science_Exploration/Space_Science/BepiColombo/BepiColombo_factsheet, 03 2024. Accessed: 2024-03-28.
- Alistair Moffat. Huffman coding. ACM Comput. Surv., 52(4), aug 2019. ISSN 0360-0300. doi: 10.1145/3342555. URL <https://doi.org/10.1145/3342555>.
- Honghui Zhan. Image compression and reconstruction based on gui and huffman coding. Journal of Physics: Conference Series, 2580:012025, 09 2023. doi: 10.1088/1742-6596/2580/1/012025.
- S. Golomb. Run-length encodings (corresp.). IEEE Transactions on Information Theory, 12(3): 399–401, 1966. doi: 10.1109/TIT.1966.1053907.
- Tom Anderson. Golomb-Rice Coding — urchin.earth.li. https://urchin.earth.li/~twic/Golomb-Rice_Coding.html, 2005. [Accessed 04-05-2024].
- Roland Ottensamer. Lecture notes in computational concepts, March 2023.
- G. G. Langdon. An introduction to arithmetic coding. IBM Journal of Research and Development, 28(2):135–149, 1984. doi: 10.1147/rd.282.0135.