THE FULL-WAVEFORM LIDAR RIEGL LMS-Q680I:  
FROM REVERSE ENGINEERING TO SENSOR MODELING  

André Jalobeanu, Research Scientist  
CGE, University of Évora, Portugal  
jalobeanu@uevora.pt  

Gil R. Gonçalves, Professor  
INESC and Dept. of Mathematics, University of Coimbra, Portugal  
gil@mat.uc.pt  

ABSTRACT  

The development of new data processing methods requires access to the raw data. Unfortunately some LiDAR manufacturers do not provide information about the format and the users can only rely on proprietary software to do their processing. Even if using black boxes might be sufficient for some simple applications, it might be an impediment to scientific research, as the processing would be limited to state of the art methods and their current implementation. In existing full-waveform LiDAR software there is a lack of error propagation methods that might be an issue when making quantitative measurements of topography, reflectance, or vegetation parameters. This problem can only be addressed at the lowest level by working directly on the waveforms. Moreover, to improve range measurement and feature extraction techniques, and compute error bars correctly, one also needs an instrument model describing the data acquisition process.

Here we focus on the Riegl LMS-Q680i airborne LiDAR sensor. We acquired 200 km² of data (nearly 100 GB of undocumented binary files). We performed a reverse engineering to understand how the timestamps, look angles and waveforms were stored. Then we developed a model of this particular sensor: the two nonlinear detector channels, the asymmetric amplifier impulse response and the ringing effect. Assuming this model, not only were we able to match the output of the proprietary software, but we also managed to compute the range uncertainty, and we opened the way to new methodologies to improve the reliability and the accuracy of echo extraction.

KEYWORDS: LiDAR, waveform, modeling, data acquisition, signal processing, error propagation

INTRODUCTION  

The main goal of our research project (AutoProbaDTM) is to develop new methodologies to provide an uncertainty layer to digital elevation models (DEM) obtained from airborne LiDAR data, and demonstrate their applicability to large data sets. This is possible through the use of Bayesian inference (Jalobeanu, 2011) but it requires full waveform recording and processing (Chauve, 2007 and Mallet, 2009), and reading the raw data is essential. Moreover, to improve the range measurement algorithms and go beyond the current state of the art, and to freely process the data without being constrained to use proprietary black boxes provided by the instrument manufacturers, one clearly needs to have unrestricted access to the raw waveform data and all the related parameters.

The funding source required the test area to be located in Portugal; we managed to contract the French company IMAO to fly over 200 km² of land using an IGI LiteMapper 6800, based on a Riegl LMS-Q680i sensor (Riegl); see Fig. 1 for area and flight details. The main dataset consists of 14 sdf files of 6 GB each and a POF (position and orientation file) generated by IMAO from a GPS base station recording and the on-board IMU data. A calibration cross was flown; an initial boresight alignment (Shan, 2008) was performed by the company, and a new method is being developed within the project (Gonçalves, 2011). Experimental lines at 3000 m were flown with various pulse rates, beyond the specifications, to investigate how data processing could help to fly higher and cheaper and still obtain satisfactory topography. More than 9000 control points of centimetric accuracy have been acquired since, and will be used to check the georeferencing and validate the developed probabilistic DEM generation algorithms.

Unfortunately at the time of the acquisition (June 2, 2011) no library nor documentation were available to help read the binary files stored in the proprietary sdf format. Riegl did not release their free library RiWaveLib 1.0.11 until late July, and even the latest version 1.3.51, 32-bit Windows dll, Python wrapper Outputs only timing, origin, direction, facet number and raw data samples. Light velocity given, but amplitude-dependent range correction and amplitude linearization missing.
and computes an incorrect origin vector, so that it could not be used in practice for accurate processing of large data sets. Therefore we decided to start to reverse engineer the sdf format soon after data acquisition.

Figure 1. Study area in Portugal (data acquisition by IMAO France) showing the flight lines and coverage density.

REVERSE ENGINEERING METHODOLOGY

Input Data and Software used in the Reverse Engineering Procedure

- A set of raw binary sdf files taken with various acquisition parameters;
- Copies of the smallest files with modified records, headers or data blocks;
- The proprietary software RiAnalyze (Riegl), used to view waveforms, mirror angles and timestamps;
- Text output generated by RiAnalyze during the extraction (range, angle, time and points in the sensor frame);
- Various log files and sdh parameter files (text files) generated by the software.

Methodology

**Waveforms.** Starting with the smallest sdf files (a few seconds of acquisition) we used a hex editor to find patterns. The sync field or record separator ‘ZZZZ’ was easily visible. The waveform data, stored as 1-byte unsigned integer arrays, was quickly found by plotting a few selected records and comparing the graphs with the waves in the RiAnalyze viewer; an emitted pulse is followed by one or more wave blocks corresponding to different channels and range intervals (the received wave is chopped into blocks, a block is recorded if the intensity exceeds a threshold, and the maximum number of blocks recorded for each pulse is fixed in advance). The block length was found (stored in 4-byte units, just before the data, along with the channel type). At first sight the two channels (Lo and Hi, respectively designed to acquire low and high power returns) did not look compatible, as the signals for a same return did not match; only a radiometric model could explain how two versions of the same signal were obtained.

**Timing.** Then we looked for the various timestamps, a relative one stored before each data block and an absolute reference within the record containing them. We found them oddly stored as 5-bytes integers, and in 4 ns units (like the data block length), not as they appear in the viewer. However, the record-related timestamp was different from the expected absolute timestamp, and we had to look deeper into the file to understand how the GPS time synchronization was encoded. Actually, after each scan line, a record that does not look like a wave was found; these are the control blocks containing the synchronization timestamps and ancillary data (sensor temperatures, etc.). Eventually, a 5-byte 4 ns-encoded value was found but after subtraction it only gave a fraction of the expected value, and the rest was stored as a more classical 4-byte integer representing the number of ms since the beginning of the week (Sunday 0:00). At last, a correct timestamp was decoded. From the RiAnalyze extraction results we figured how to add this absolute reference, the relative timestamp and the wave optimum location to form an absolute pulse time, which matched the output files to 1μs (rounding errors).

**Header.** We then managed to extract the constant and variable instrument parameters as they were given in the sdh files as ‘name=value’ in the same order as in the binary fields in the sdf header. Switching between viewing options in the hex editor helped to find the encoding (big endian) and the size of each field. Knowing the names and
Scanning angle. The most difficult was to decode the angle θ, expected to be found in each record block as a single float or integer representing a value between 60º and 120º. It turned out to be stored as 4 raw encoder values of 2 bytes each (was the onboard conversion too expensive for the embedded software?). It took some fiddling to get the right combination of fine and coarse coefficients and offsets and convert the 4 integers into an scanning angle. We achieved 0.001º absolute deviation from the RiAnalyze output files.

Ranging. Finding the range turned out to be more complicated than multiplying a travel time by c/2. In the end, we understood what the software computes and it motivated us to develop a radiometric model for the sensor. After adding a fixed combination of offset parameters, one has to add an amplitude-dependent offset, stored in the header as a look-up table. We edited the files, by changing the parameters or replacing the data blocks with synthetic Gaussian waves, to check and refine the equations for range and amplitude correction, and also for angle decoding.

Geometry. Finally, once a range is computed and the angle is known, one needs a geometric model to obtain a 3D point in the sensor frame; all we knew was that there were 4 mirror facets and a 60º field of view, and various geometric parameters were provided. We found a rotating square pyramid scheme that would be compatible with all the constraints and parameter values, and that would produce the same results as RiAnalyze (up to rounding errors).

Some fields in the pulse and control record blocks are still unexplained and we ignore them (e.g. the record checksum); the same is true for some header parameters, not essential to the decoding and extraction process.

**BINARY SDF FORMAT DESCRIPTION**

**File Structure**

The general structure is illustrated in Fig. 2; we keep the same color code in the following tables for clarity. A header is followed by rows of pulse records corresponding to a scan line, θ varying over the entire 60º field of view. Control records separate the rows and encode GPS time synchronization (when available) and housekeeping data. Each pulse record contains timing and angle information, the digitized emitted pulse, and a number of received waveform blocks that can come from previously emitted pulses if there is more than one pulse in the air. The records are separated by 4-byte synchronization fields, and the record length is variable; these fields might be used in a simple indexing procedure if one needs random access to file records.

**Record Blocks and Reading Algorithm**

The header is read first; see Table 1 for a description of the fields. Throughout the file the big endian format is used and the bytes have to be reordered for little endian processors, most commonly used. The different blocks that make the pulse and control records are detailed in Table 2. When reading the Rec blocks, a flag determines the type of record (pulse or control). We seek the first control record of the file to help synchronize the timestamps for the first row. The pulse records are processed as follows: timestamp, angle and the number of blocks are decoded from Rec. For each block DataHdr is decoded to get the type (emitted, high, or low), the block timestamp, and the number of samples, then the corresponding wave is read, and so on. The emitted pulse comes first. For ground extraction, only the last received wave is considered, ignoring the others.

**Variable Decoding and Conversion**

Timestamps in Rec and Ctrl are stored as 5-bytes long integers. All other values are stored in a more conventional way (1, 2 or 4 bytes for integers, 4 bytes for floats). Each waveform sample occupies a single byte to minimize the storage space (the dynamic compression scheme is explained further).

The variables and constants used for waveform extraction, renamed to simplify the equations, are shown in Table 3. The fields are decoded and converted according to rules derived from reverse engineering. In addition to unit conversion, offsets and factors are redefined, again to simplify the equations.

In the header, the light velocity in the air is provided through a PPM correction factor. The range offsets r are provided in mm but actually correspond to time delays, independent of the atmospheric conditions, hence the use of the light velocity correction. A 4 ns delay was also taken into account as a 0.60 m offset. The coarse angle encoder resolution K has a simple expression, but the fine encoders actually measure a time delay since the last coarse encoder step, hence the more complex expression of K involving pulse rate and number of pulses per scan line. The encoder offsets are given in the header, but they had to be adapted, as their unit is not the same as the fine encoders.
**Figure 2.** Q680 sdf format structure diagram showing the different types of blocks and their relative position.
Table 1. Q680 sdf header structure: field name, length (bytes), type (u/i=unsigned/signed integer, f=float, s=string), description and unit of each identified field. Big endian encoding. Names from sdh text file generated by RiAnalyzer.

struct Q680HeaderBuf
{
  char headerSize [4]; // 14 size of this header
  char protocolFlags [4]; // 4 ?
  char headerIDMain [5]; // u2 ?
  char headerIDSub [5]; // u2 ?
  char parameterIDMain [4]; // u1 ?
  char parameterIDSub [4]; // u1 ?
  char HeadMain [4]; // u1 ?
  char HeadIDSub [4]; // u1 ?
  char SampleIDLock1Main [4]; // u1 ?
  char SampleIDLock1Sub [4]; // u1 ?
  char HousekeepingIDMain [4]; // u1 ?
  char HousekeepingIDSub [4]; // u1 ?
  char SerialNumber [5]; // s8 serial number
  char InstrumentType [5]; // s8 type (Q680)
  char InstrumentModel [5]; // u2 model number
  char dum [2]; // --
  char syncField [4]; // 14 record synchronization field (=IEEE)
  char beamAperture [2]; // u2 laser beam aperture (?)
  char dum2 [2]; // --
  char beamDivergence [2]; // u2 laser beam divergence (?)
  char dum3 [2]; // --
  char beamFocus [2]; // -- beam focus (?)
  char dum4 [2]; // --
  char LinAngleCircleCount [4]; // 4 encoder steps (for 360 deg)
  char numberFacets [4]; // u1 number of facets [4]
  char dx [4]; // --
  char LinAngleCorrRes [4]; // 4 coarse angle resolution
  char LinAngleOffset0 [4]; // 4 coarse angle encoder 1 offset
  char LinAngleOffset1 [4]; // 4 coarse angle encoder 2 offset
  char LinAngleCorrResOffset0 [4]; // 4 fine angle encoder 1 offset
  char LinAngleCorrResOffset1 [4]; // 4 fine angle encoder 2 offset
  char RangeCorrPRM [4]; // 4 light velocity correction (FPM)
  char lChannelRangeOffset [4]; // 4 low channel range offset (mm)
  char rChannelRangeOffset [4]; // 4 high channel range offset (mm)
  char CorrLaserOriginX [4]; // 4 x laser origin vector (m)
  char CorrLaserOriginY [4]; // 4 y laser origin vector (m)
  char CorrLaserOriginZ [4]; // 4 z laser origin vector (m)
  char CorrLaserDirX [4]; // 4 x laser direction vector
  char CorrLaserDirY [4]; // 4 y laser direction vector
  char CorrLaserDirZ [4]; // 4 z laser direction vector
  char CorrMirrorAxisX [4]; // 4 x mirror axis vector
  char CorrMirrorAxisY [4]; // 4 y mirror axis vector
  char CorrMirrorAxisZ [4]; // 4 z mirror axis vector
  char CorrOffsets [4]; // 4 x offset vector (m)
  char CorrOffsets [4]; // 4 y offset vector (m)
  char CorrOffsets [4]; // 4 z offset vector (m)
  char CorrFaceNormXs [4]; // 4 x facet 1 normal vector
  char CorrFaceNormYs [4]; // 4 y facet 1 normal vector
  char CorrFaceNormZs [4]; // 4 z facet 1 normal vector
  char CorrFaceDistances1 [4]; // 4 facet 1 distance [m]
  char CorrFaceNormX2 [4]; // 4 x facet 2 normal vector
  char CorrFaceNormY2 [4]; // 4 y facet 2 normal vector
  char CorrFaceNormZ2 [4]; // 4 z facet 2 normal vector
  char CorrFaceDistances2 [4]; // 4 facet 2 distance [m]
  char CorrFaceNormX3 [4]; // 4 x facet 3 normal vector
  char CorrFaceNormY3 [4]; // 4 y facet 3 normal vector
  char CorrFaceNormZ3 [4]; // 4 z facet 3 normal vector
  char CorrFaceDistances3 [4]; // 4 facet 3 distance [m]
  char CorrFaceNormX4 [4]; // 4 x facet 4 normal vector
  char CorrFaceNormY4 [4]; // 4 y facet 4 normal vector
  char CorrFaceNormZ4 [4]; // 4 z facet 4 normal vector
  char CorrFaceDistances4 [4]; // 4 facet 4 distance [m]
  char LinearAmplitudeArray [0]; // u2[256] nonlinear amplitude correction (low channel)
  char AmplitudeArrayLO [0]; // u2[100] amplitude-dependent range correction (low channel) (mm)
  char AmplitudeArrayHI [0]; // u2[100] amplitude-dependent range correction (high channel) (mm)
  char ACThreshold&ArrayLO [0]; // u1[2048] ADC thresholds (low channel)
  char ACThreshold&ArrayHI [0]; // u1[2048] ADC thresholds (high channel)
  char ADCOffset [8]; // u2[4] ADC offsets
  char LastPulseSuppression [8]; // u2[4] pulse suppression
  char hearRangeSuppression [8]; // u4 ignore nearest return (unit?)
  char LaserPulseRate [8]; // u4 Pulse Repetition Rate (Hz)
  char MTMaxConfig [8]; // u1 ?
  char dum [3]; // --
  char MTMaxPwrLow [8]; // u4 PWR lower bound (Hz)
  char MTMaxPwrHigh [8]; // u4 PWR higher bound (Hz)
  char MTMaxTime [8]; // u1 number of pulses in the air
  char dum7 [8]; // --
  char LineScanStartValue [8]; // u2[4] ?
  char LineScanIncrement [8]; // u2[4]?
  char LineScanFramesNumber [8]; // u4 measurements per line
  char RangeData [8]; // u2 ?
  char PositionStartPulse [8]; // u2 ?
  char SampleBlkLength [8]; // u1[4] data block lengths (4us)
  char ChannelCount [8]; // u1[4]?
  char ACThreshold [8]; // u1[4]?
  char ChannelSwitchLevel [8]; // u1 hi channel recording threshold?
  char dum [8]; // --
  char LogChanel [8]; // u1 ?
  char dum [8]; // --
  char FirstBlk [8]; // u1 ?
  char LastBlks [8]; // u1 ?
  char dum [3]; // --
  char dum [3]; // --
  char LogStringDecoder [8]; // u2[16] ?
};

Header (6640 bytes) sdf file header.

LMS-Q680i instrument constants:
- File format parameters
- Angle encoder constants
- Geometry constants
- Range correction arrays
- Amplitude correction arrays

Variables:
- Pulse Repetition Rate (PRR)
- Measurements per line
- MTA zone (pulses in the air)
- MTA PRR bounds
Table 2. Q680 sdf block structure: field name, length (bytes), type (u/i=unsigned/signed integer, f=float, s=string), description and unit of each identified field. Big endian encoding (conversion required for Intel-type processors).

<table>
<thead>
<tr>
<th>Field</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rec (24 bytes)</td>
<td>Record parameters:</td>
</tr>
<tr>
<td>Data/Control switch</td>
<td>- Data/Control switch</td>
</tr>
<tr>
<td>Raw timestamp</td>
<td>- Raw timestamp</td>
</tr>
<tr>
<td>Theta (4 encoders)</td>
<td>- Theta (4 encoders)</td>
</tr>
<tr>
<td>Number of waveform blocks</td>
<td>- Number of waveform blocks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DataHdr (4 bytes)</th>
<th>Waveform block header:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative timestamp</td>
<td>- Relative timestamp</td>
</tr>
<tr>
<td>Type, length</td>
<td>- Type, length</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ctrl (68 bytes)</th>
<th>GPS synchronization:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync timestamp</td>
<td>- Sync timestamp</td>
</tr>
<tr>
<td>GPS weeksec</td>
<td>- GPS weeksec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EmPulse (DataHdr.1 bytes)</th>
<th>Sampled emitted pulse</th>
</tr>
</thead>
<tbody>
<tr>
<td>char EmPulse[DataHdr.1];</td>
<td>u emitted pulse (ns sampling)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RecWave (DataHdr.1 bytes)</th>
<th>Sampled received wave block</th>
</tr>
</thead>
<tbody>
<tr>
<td>char RecWave[DataHdr.1];</td>
<td>u received wave block (ns sampling)</td>
</tr>
</tbody>
</table>

**ACCURATE TIMING AND MTA PROCESSING**

**Accurate Pulse Time Determination**

The time relative to an external synchronization timestamp is determined by adding the integer record and data timestamps to the fractional pulse location within the wave block, obtained from signal processing (in italics).

\[
t_E = t_{Rec} + t_{Data} + t_{EmPulse} \quad \text{and} \quad t_R = t_{Rec} + t_{Data} + t_{RecWave}
\]  

For the emitted pulse, \(t_{EmPulse}\) is obtained via simple Gaussian pulse estimation from 3 points near the optimum or through Gaussian fitting (Guo, 2011) but an estimation window should be used to avoid artifacts due to ringing effects in the tail (see Fig. 3). Given the high signal to noise ratio, the simple 3-point technique performs well.

However, extracting the echo location \(t_{RecWave}\) from the received signal is trickier, due to noise and various contaminants, and depends on the application, for instance topography or canopy mapping. Waveform processing is still open research (Mallet, 2009). Gaussian decomposition is a commonly used method (Hofton, 2000; Wagner, 2006) that provides a series of echo locations and is implemented in RiAnalyze (Riegl). Within the AutoProbaDTM project, we intend to improve the state of the art concerning the estimation accuracy and provide rigorous uncertainty estimates as well, which are currently not available. We obtained more robust estimates of the ground echo (Jalobe-anu, 2011) through the automatic selection of the estimation window size, allowing to reduce the bias when ground peaks are contaminated by vegetation echoes; in these cases the Gaussian decomposition usually fails.

**GPS Synchronization**

When available (it was in our dataset) GPS time synchronization enables one to use the emitted pulse time to interpolate the corresponding position and orientation in the POS file for direct georeferencing (sampled at 256 Hz in the LiteMapper). \(W\) is the number of seconds since either the beginning of the day, or the beginning of the week; \(t_{sync}\) is the current LiDAR ns timestamp counter at the moment of synchronization, therefore we have:

\[
T = W + 10^{-9} (t_E - t_{sync})
\]
**Multiple Pulses in Air (MTA) Processing**

When there are multiple pulses in the air (number MTA) due to a combination of high repetition rate (PRR) and high altitude, the pulse emission times \( t_e \) are placed on a FIFO stack of size MTA so that we can compute:

\[
\Delta t = t_e(n) - t_e(n - \Delta n) \tag{3}
\]

**Table 3.** Decoded variables and constants relevant to waveform extraction and processing, unit when applicable, and equations or sdf fields used. **3D vectors** in bold, **wave block processing results** in italics.

<table>
<thead>
<tr>
<th>Variable/Constant</th>
<th>Unit</th>
<th>Decoding function using sdf block fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_0^{Lo} )</td>
<td>m</td>
<td>( (c_{12}/0.15)(10^{-3} \text{Header.LoChannelRangeOffset} - 0.60) )</td>
</tr>
<tr>
<td>( r_0^{Hi} )</td>
<td>m</td>
<td>( r_0^{Lo} + (c_{12}/0.15)(10^{-3} \text{Header.HiChannelRangeOffset}) )</td>
</tr>
<tr>
<td>( \Delta t^{Lo}[256] ), ( \Delta t^{Hi}[256] )</td>
<td>m</td>
<td>( 10^{-3} \text{Header.AmplCorrArrayLo} ), ( 10^{-3} \text{Header.AmplCorrArrayHi} )</td>
</tr>
<tr>
<td>( c_{12} )</td>
<td>m ns(^{-1} )</td>
<td>0.15 ( (1 + 10^{-6} \text{Header.RangeCorrPPM}) )</td>
</tr>
<tr>
<td>( \Delta n )</td>
<td>-</td>
<td>\text{Header.MTAZone} - 1</td>
</tr>
<tr>
<td>( F_{i,Lo}[256] ), ( F_{i,Hi}[256] )</td>
<td>-</td>
<td>\text{Header.LinearizeArray}, \text{Header.DeLogArray}</td>
</tr>
<tr>
<td>( E_{sep} )</td>
<td>deg</td>
<td>360 ( / \text{Header.LineAngleCircleCount} )</td>
</tr>
<tr>
<td>( K )</td>
<td>deg</td>
<td>( 1/2 E_{sep} \text{Header.LineAngleCoarseRes} )</td>
</tr>
<tr>
<td>( K_f )</td>
<td>deg</td>
<td>( 3 \times 10^7 \text{Header.LaserPulseRate} / \text{Header.LineScanMeasNumber} )</td>
</tr>
<tr>
<td>( \theta_0 )</td>
<td>deg</td>
<td>( 1/2 E_{sep}(\text{Header.LineAngleOffset0} + \text{Header.LineAngleOffset1}) ) + ( 3/2 K_f \times (\text{Header.LineEncoderFineCountOffset0} + \text{Header.LineEncoderFineCountOffset1}) )</td>
</tr>
<tr>
<td>( d[4] )</td>
<td>m</td>
<td>\text{Header.CorrFacetDistance}</td>
</tr>
<tr>
<td>( n[4] )</td>
<td>-</td>
<td>\text{Header.CorrFacetNormal}</td>
</tr>
<tr>
<td>( a )</td>
<td>-</td>
<td>\text{Header.CorrMirrorAxis}</td>
</tr>
<tr>
<td>( l )</td>
<td>-</td>
<td>\text{Header.CorrLaserDir}</td>
</tr>
<tr>
<td>( L )</td>
<td>m</td>
<td>\text{Header.CorrLaserOrigin}</td>
</tr>
<tr>
<td>( t_{Rec} )</td>
<td>ns</td>
<td>4 \text{Rec.tc}</td>
</tr>
<tr>
<td>( C_1 ), ( C_2 ), ( C_{1f} ), ( C_{2f} )</td>
<td>-</td>
<td>\text{Rec.C1}, \text{Rec.C2}, \text{Rec.C1f}, \text{Rec.C2f}</td>
</tr>
<tr>
<td>( W )</td>
<td>s</td>
<td>( 10^{-3} \text{Ctrl.weeksec} )</td>
</tr>
<tr>
<td>( t_{sync} )</td>
<td>ns</td>
<td>4 \text{Ctrl.sync}</td>
</tr>
<tr>
<td>( t_{nua} )</td>
<td>ns</td>
<td>4 \text{DataHdr.tc}</td>
</tr>
<tr>
<td>( t_{pP} )</td>
<td>ns</td>
<td>\text{EmPulse waveform processing (Gaussian pulse estimation or other)}</td>
</tr>
<tr>
<td>( t_{RecWave, A} )</td>
<td>ns</td>
<td>\text{RecWave waveform processing (research in progress)}</td>
</tr>
</tbody>
</table>

**SENSOR RADIOMETRY AND RANGE ESTIMATION**

**Radiometric Information Decoding**

The waveform data samples are stored in a simple 1-byte unsigned integer format, with 1 ns sampling interval. The instrument implements a dynamic range compression scheme through a nonlinear amplifier which reduces the file size by a factor 2. The less sensitive channel (Lo) is almost logarithmic, while the other (Hi) has a more linear behavior and is actually linear on \([0,120]\). To recover the original values the simplest way is to use the conversion arrays provided in the file header. This is valid for Gaussian pulse amplitudes, assuming the peaks are well separated, and should not be applied to raw wave sample values.
\[ A_{\text{corr}} = F_{-1}^{\text{Lo}}[A] \quad \text{or} \quad A_{\text{corr}} = F_{-1}^{\text{Hi}}[A] \]

The corrected amplitude ranges are \([0,840]\) for Lo and \([0,65565]\) for Hi. An intensity calibration needs to be done (Shan, 2008) to transform these amplitudes and form intensity images.

**Ranging using Simple Corrections**

Due to various fixed time delays in the system, a range offset has to be added to the distance \(c_{1/2} \Delta t\), however this is not sufficient to obtain accurate results. Indeed, the transimpedance amplifier used in the receiver electronics exhibits sufficient nonlinearity to introduce a significant delay that depends on the amplitude.

The simple solution is to use the provided look-up tables \(\Delta r\) with the underlying assumption is that the peaks are well-separated; thus one can run a Gaussian fit, and use the estimated amplitude to correct the range as if the entire Gaussian had been shifted. This is the solution that seems to be implemented in RiAnalyze (but not in the Ri-WaveLib library as it requires an amplitude to be known).

\[
r = r_0^\text{Lo} + \Delta r^\text{Lo}[A] + c_{1/2} \Delta t \quad \text{or} \quad r = r_0^\text{Hi} + \Delta r^\text{Hi}[A] + c_{1/2} \Delta t
\]

(5)

**Accurate Radiometric Modeling and its Applications**

The simple processing just mentioned is based on assumptions that can be easily broken when considering ground peaks contaminated by low vegetation echoes. In such cases, classical Gaussian fitting can be replaced by a robust technique such as the one introduced by (Jalobeanu, 2011). In this work we had to account for ringing, not only for contaminating peaks from real echoes. The ringing effect due to the amplifier is obvious (except when the signal to noise ratio is very low) and false peaks could be extracted (see Fig. 3 for an illustration).

A solution consists of accurately modeling the overall impulse response (IR) of the amplifier. The easiest approach is to first neglect the nonlinearities and consider an amplitude-independent function made of a sum of Gaussians. Instead of performing a full deconvolution with the IR that requires regularization to avoid noise amplification, but which produces unwanted artifacts, we apply a partial deconvolution using only the peak positions and amplitudes from the IR, not trying to remove the effect of the Gaussian blur. This effectively removes the ringing and suppresses the false detections. In practice we use the discrete kernel \([1 \ 0 \ 0 \ 0.03 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0.04]\) and a Gaussian of full width at half maximum (FWHM) of 4.3 ns.

The IR can be made amplitude-dependent, including the delay \(\Delta r\) function of the amplitude, for each channel. To be complete, the model should also include a nonlinear gain function \(f\) whose inverse would correspond to the \(F_i\) arrays. The IR model can be expressed as \(h(t,x) = f(x) g(t-\tau(x))\) where \(\tau\) is the signal-dependent delay function and \(g\) the sum-of-Gaussians IR of the previous paragraph. A proper inversion, without trying to remove the Gaussian blur, would produce more consistent results than Gaussian decomposition and would not necessary be more complex. The decomposition of a waveform using a sum of Gaussians or other functions is indeed a computationally intensive task (Wagner, 2006; Chauve, 2007). Complex waveforms could be processed to remove the nonlinear instrumental artifacts; after ground peak subtraction the processed waveform would be very useful in forestry applications.

**Figure 3.** Accumulated normalized waves showing the ringing effect. Left: emitted pulses, right: received waves (single echo) and proposed impulse response (IR) model.
SENSOR GEOMETRY

Angle Decoding

The two coarse encoder values have to be added to form a consistent value; the same is valid for the fine encoder values. Then we apply the respective factors $K$ and $K_f$, and add the global offset $\theta_0$. For some reason a 180º jump occurs when one of the coarse encoders becomes bigger than the other, and has to be corrected.

$$\theta_{\text{full}} = (\theta_0 + K (C_1 + C_2) + K_f (C_{1f} + C_{2f}) + 180 (C_{1f} > C_{2f})) \mod 360$$  \hspace{1cm} (6)

This angle parametrizes the rotation of the full pyramid-shaped mirror and we have to extract the current facet angle $\theta$ and number $F$. We have a value in [15º,75º] (the origin is arbitrary, this choice simplifies subsequent geometric computations, and we just add 45º to agree with the RiAnalyze output files). For one facet we have then:

$$\theta = \theta_{\text{full}} \mod 90 \quad \text{and} \quad F = \text{int}(\theta_{\text{full}}/ 90)$$  \hspace{1cm} (7)

Sensor Own Coordinate System (SOCS) Geometric Model

The most probable model for the scanning device is a rotating square pyramid mirror, illustrated by Fig. 4. The pulse is shot in the direction of the y axis at about 2 cm from this axis; the pyramid rotates around the y axis which allows scanning along almost perfectly straight lines.

The facets are planar and parametrized by a normal vector $\mathbf{n}[F]$ and a distance from the origin $d[F]$. We first apply a rotation of angle $\theta$ around the axis vector $\mathbf{a}$ using the Rodrigues formula to get the rotated facet normal:

$$\mathbf{n}_{\theta} = \text{Rotation}(\mathbf{n}[F], \mathbf{a}, \theta) \quad \text{where} \quad \text{Rotation}(\mathbf{v}, \mathbf{a}, \theta) = \cos \theta \mathbf{v} + (1-\cos \theta) (\mathbf{a} \cdot \mathbf{v}) \mathbf{a} + \sin \theta \mathbf{a} \times \mathbf{v}$$  \hspace{1cm} (8)

The look direction is obtained by reflecting the laser beam direction vector $\mathbf{l}$ on the facet with rotated normal $\mathbf{n}_{\theta}$:

$$\mathbf{u} = \text{Mirror}(\mathbf{l}, \mathbf{n}_{\theta}) \quad \text{where} \quad \text{Mirror}(\mathbf{v}, \mathbf{n}) = \mathbf{v} - 2 (\mathbf{v} \cdot \mathbf{n}) \mathbf{n}$$  \hspace{1cm} (9)

The actual origin to consider is the virtual image of the laser source $\mathbf{L}$ (apparent position of the reflected point $\mathbf{L}$) which requires to find the intersection $\mathbf{M}$ of the beam with the mirror surface. After some geometry we get:

$$\mathbf{p}_0 = \mathbf{L} + d_{LM} (\mathbf{l} - \mathbf{u}) \quad \text{where} \quad d_{LM} = (d[F] - \mathbf{L} \cdot \mathbf{n}) / \mathbf{L} \cdot \mathbf{n}$$  \hspace{1cm} (10)

We did not use the CorrOffset vector from the header, since it is equal to zero for our sensor.

In the SOCS frame the position of a point $\mathbf{p}$ corresponding to a range $r$ is then:

$$\mathbf{p} = \mathbf{p}_0 + r \mathbf{u}$$  \hspace{1cm} (11)

We have finished decoding the binary data. Then the LiDAR equation can be applied to georeference the point (Wehr, 1999; Shan, 2008) as a starting point for DEM generation; if range uncertainties are estimated during the waveform processing, they can be propagated, opening the way to probabilistic DEM inference.
Figure 4. Determination of the origin $p_0$ and the laser direction $u$ in the sensor coordinate system assuming the scanner mechanism is a rotating square pyramid mirror (parameters given in the header block).

CONCLUSION

Proprietary formats are an impediment to algorithmic research. The widespread use of black boxes in professional data processing gives the impression that the problems are solved and there is no room for improvement. Moreover, the lack of freedom in the choice of processing tools is an issue for quality assessment, and we believe that alternate methods should be available at least to help assess the reliability of the mainstream ones.

A first step in this direction is being made by companies such as Riegl who eventually decided to release a library to read their sdf file format; however at the time of writing, this library is not functional enough to be used at full scale, and the format specification is still unknown. We reverse engineered the files to be able to start doing research on large LiDAR dataset processing using new methodologies, without having to wait for library updates, and we already made significant progress. Our findings and the related source code are published on the project website (AutoProbaDTM). Sample data can also be provided upon request.

ACKNOWLEDGEMENTS

The present work, the data acquisition and the field campaigns were funded by the Foundation for Science and Technology (FCT) of Portugal in the framework of the AutoProbaDTM project (PTDC/EIA-CCO/102669/2008, FCOMP-01-0124-FEDER-010039).

REFERENCES

AutoProbaDTM research project website - sites.google.com/site/autoprobadtm.

