Algorithms used in the Airborne Lidar Processing System (ALPS)

By David B. Nagle and C. Wayne Wright

Open-File Report 2016–1046

U.S. Department of the Interior
U.S. Geological Survey

U.S. Department of the Interior
SALLY JEWELL, Secretary

U.S. Geological Survey
Suzette M. Kimball, Director

U.S. Geological Survey, Reston, Virginia: 2016

Suggested citation:
Nagle, David B., and Wright, C. Wayne, 2016, Algorithms used in the Airborne Lidar Processing System (ALPS): U.S. Geological Survey Open-File Report 2016–1046, 45 p., http://dx.doi.org/10.3133/20161046.

ISSN 2331-1258 (online)

Acknowledgments

The algorithms discussed in this report were developed and refined by many contributors. The point projection algorithm was initially implemented by Steven Bailey. The waveform analysis algorithms were improved by contributions from Christine Kranenburg, Amar Nayegandhi, Rudy Troche, and Ed Walsh. The random consensus filter was improved by assistance from Christine Kranenburg and Amar Nayegandhi. The Interactive Data Language (IDL) gridding was initially implemented by Amar Nayegandhi. Additionally, the Airborne Lidar Processing System (ALPS) software was improved in areas outside of these algorithms by Enils Bashi, Jeremy Bracone, Paul Kinzel, Jim Lebonitte, Richard Mitchell, Lance Mosher, Jon Sellars, Charlene Sylvester, and Ed Walsh. The U.S. Geological Survey Coastal and Marine Geology Program funded the development of this report. The ALPS software was developed incrementally over the past decade with funding from and (or) in partnership with the U.S. Geological Survey, the National Aeronautics and Space Administration, the National Oceanic and Atmospheric Administration, the National Park Service, the Fish and Wildlife Service, and the U.S. Forest Service.

Contents

Figures

Tables

Conversion Factors

International System of Units to Inch/Pound

Multiply	By	To obtain
	Length	
centimeter (cm)	0.3937	inch (in.)
millimeter (mm)	0.03937	inch (in.)
meter (m)	3.281	foot (ft)
kilometer (km)	0.6214	mile (mi)
kilometer (km)	0.5400	mile, nautical (nmi)
meter (m)	1.094	yard (yd)
	Area	
square meter (m^2)	0.0002471	acre
square kilometer (km^2)	247.1	acre
square centimeter (cm^2)	0.001076	square foot (ft^2)
square meter (m^2)	10.76	square foot (ft^2)
square centimeter (cm^2)	0.1550	square inch (ft^2)
square kilometer (km^2)	0.3861	square mile (mi^2)
	Velocity	
meter per nanosecond (m/ns)	3.281	foot per nanosecond (ft/ns)
meter per second (m/s)	3.281	foot per second (ft/s)

Supplemental Information

Refractive index of air: 1.000276

Refractive index of water: 1.3330

Speed of light in space: 0.29979 meters per nanosecond (m/ns)

Speed of light through air: 0.2997 meters per nanosecond (m/ns)

Speed of light through water: 0.2249 meters per nanosecond (m/ns)

Abbreviations

AGC	Automatic Gain Control
ALPS	Airborne Lidar Processing System
DEMs	Digital Elevation Models
EAARL–A	Experimental Advanced Airborne Research Lidar, Version “A”
EAARL–B	Experimental Advanced Airborne Research Lidar, Version “B”
ENVI IDL	Environment for Visualizing Images–Interactive Data Language
GeoTIFF	Georeferencing Tagged Image File Format
GPS	Global Positioning System
IDL	Interactive Data Language
INS	Inertial Navigation System
lidar	Light Detection and Ranging
m	meter
mrad	milliradian
NAD83	North American Datum of 1983
nm	nanometer
NAVD88	North American Vertical Datum of 1988
ns	nanosecond
POSIX	Portable Operating System Interface
RANSAC	Random Sample Consensus
RCF	Random Consensus Filter
rx	return waveform
sr	steradian
tx	transmit waveform
USGS	U.S. Geological Survey
UTM	Universal Transverse Mercator
W	watt
WGS84	World Geodetic System

Algorithms used in the Airborne Lidar Processing System (ALPS)

By David B. Nagle and C. Wayne Wright

Abstract

The Airborne Lidar Processing System (ALPS) analyzes Experimental Advanced Airborne Research Lidar (EAARL) data—digitized laser-return waveforms, position, and attitude data—to derive point clouds of target surfaces. A full-waveform airborne lidar system, the EAARL seamlessly and simultaneously collects mixed environment data, including submerged, sub-aerial bare earth, and vegetation-covered topographies.

ALPS uses three waveform target-detection algorithms to determine target positions within a given waveform: centroid analysis, leading edge detection, and bottom detection using water-column backscatter modeling. The centroid analysis algorithm detects opaque hard surfaces. The leading edge algorithm detects topography beneath vegetation and shallow, submerged topography. The bottom detection algorithm uses water-column backscatter modeling for deeper submerged topography in turbid water.

This report describes slant range calculations and explains how ALPS uses laser range and orientation measurements to project measurement points into the Universal Transverse Mercator coordinate system. Parameters used for coordinate transformations in ALPS are described, as are Interactive Data Language- (IDL) based methods for gridding EAARL point cloud data to derive digital elevation models. Noise reduction in point clouds through use of a random consensus filter is explained, and detailed pseudocode, mathematical equations, and Yorick source code accompany this report.

Introduction

Light detection and ranging (lidar) is a remote sensing method that transmits short light pulses to a target and measures the return time of the associated backscattered light. Other valuable information can be determined by capturing a time record of the returning backscatter or return waveform. The speed of light in air (or for bathymetric systems, water) and the return waveform's time record can be used to calculate a slant range measurement. Slant range measurements combined with sensor position and attitude data can be used to derive three-dimensional point clouds—collections of georeferenced points—of surfaces.

Targets such as ground surfaces, water surfaces, submerged surfaces, and forest canopies appear as features within lidar waveforms (fig. 1). The characteristics of these features vary depending on many factors, including the type of surface surveyed and the design of the lidar system. Processing algorithms and software must take these factors into account when generating target point clouds. Submerged targets are particularly challenging to resolve due to complications introduced by the water column, such as surface refraction, attenuation, scattering, absorption, and bottom reflectivity (Guenther, 1985).

Many commercial airborne topo-bathymetric lidar systems use proprietary algorithms and software that prevent lidar consumers and scientists from understanding how the data are generated. This report provides a complete algorithm-level description of the processing workflows used in a modern research lidar. While the techniques documented herein likely differ from those in proprietary systems, they are general enough to be adapted for use with common lidar systems.

Experimental Advanced Airborne Research Lidar

The Experimental Advanced Airborne Research Lidar (EAARL) is a software-defined full-waveform airborne lidar system designed for the simultaneous and seamless collection of mixed environment data, including submerged topography, sub-aerial topography, and vegetation-covered topography. The design of the current version of the system, designated EAARL–B, differs from traditional bathymetric systems in four primary ways: it uses a relatively short laser pulse (0.7 nanoseconds (ns)); it has three narrow receiver fields-of-view of 2 milliradians (mrad) each and a fourth, larger field-of-view of 18 mrad; it records digitized signal backscatter full waveforms; and it relies on software instead of hardware for real-time signal processing. The laser operates at a 532-nanometer (nm) wavelength (green). An oscillating cross-track scanning mirror distributes the pulses across a 240-meter (m)-wide swath perpendicular to the airplane flight track from a 300-m altitude.

The original EAARL–A system began operation in 2001 but was superseded in 2012 by the EAARL–B system. The algorithms in this report were developed during the lifecycles of both systems. In this report, the term EAARL is used for information that applies equally to both systems and the term EAARL–B is used for system-specific information that applies only to the EAARL–B system.

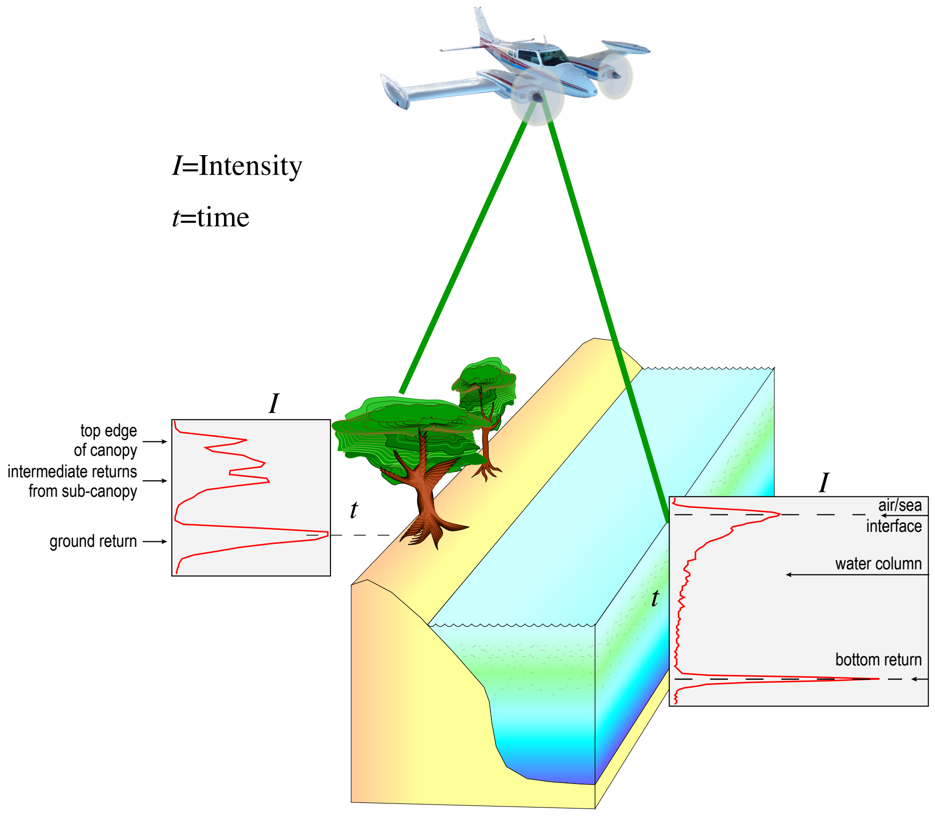


Figure 1. Example EAARL–B (Experimental Advanced Airborne Research Lidar) waveforms from vegetation and submerged environments (Wright and Brock, 2002, revised).

Airborne Lidar Processing System

Post-processing of EAARL data uses the Airborne Lidar Processing System (ALPS). The digitized laser-return waveforms are combined with the aircraft positioning and attitude data derived from a global positioning system (GPS)-aided inertial navigation system (INS) receiver to derive data products, including point clouds. The point clouds are processed into digital elevation models (DEMs). The algorithms used by the ALPS software are the focus of this report; Bonisteel and others (2009) describe how to use the software to process data.

The ALPS software is primarily built on open-source technologies and requires a Portable Operating System Interface (POSIX) environment (typically Linux). The majority of the code is written in Yorick, an interpreted programming language designed for scientific numerical processing and visualization by David H. Munro at Lawrence Livermore National Laboratory (Munro and Dubois, 1995). Yorick uses an array syntax that permits fast and powerful manipulation of large, multi-dimensional arrays of data without explicit loops.

ALPS leverages several other programming languages and tools, including C, Tcl/Tk, Makeflow, and ENVI IDL (Environment for Visualizing Images–Interactive Data Language). Yorick extensions are written in C to optimize performance at algorithmic bottlenecks. Tcl/Tk is used to provide a graphical user interface around the Yorick-based functionality. Makeflow, designed by the University of Notre Dame’s Cooperative Computing Lab, permits efficient processing of large batch jobs in parallel (Albrecht and others, 2012). ENVI IDL, designed by Exelis Visual Information Solutions, is used to generate DEMs and is the only closed-source technology in the platform.

Workflow Overview

Lidar processing workflows in ALPS begin with the processing of digitized waveforms to create a point cloud. After a point cloud is derived, post-processing steps can include filtering the point cloud to remove noise, coordinate-system transformations, and gridding production for the creation of DEMs.

The production of a usable point cloud begins with the calculation of slant range measurements. The measurement is calculated for each point using two waveforms: the transmit waveform (tx), which captures the shape of the outbound transmitted laser pulse, and the corresponding return waveform (rx), which captures the energy backscattered from the environment. The transmit waveform is analyzed to determine its center of energy. The return waveform is then analyzed to determine the location of a target. The center of energy of the transmit and the target location in the return allows calculation of the slant range time of flight to the target. The slant range time can be converted into a distance measurement using the speed of light in air, as is described in the section “Slant Range Measurement.” The analysis of the waveforms to determine the center of energy for the transmit waveform and the target location within the return waveform is detailed in the section “Waveform Analysis.”

A georeferenced point cloud is then created to represent target locations in real-world coordinates. Target ranges from the waveforms are combined with high precision and accuracy measurements—of the attitude and position of the sensor, with respect to a reference ellipsoid—to project the lidar elevation measurement to the target. This process is repeated for each pulse from each channel, resulting in a point cloud that represents target locations in real-world space. The mathematics involved are explained in the section “Point Projection.”

Uncorrelated random range noise in the point cloud is reduced by applying the Random Consensus Filter (RCF). The RCF algorithm removes outliers by detecting where the highest concentrations of points are located. This algorithm is described in the section “Random Consensus Filter (RCF).”

ALPS implements several horizontal and vertical datum transformations to provide data in a variety of coordinate systems, as is described under “Coordinate Transformations.” ALPS also provides a tool for creating a DEM from the point cloud by gridding with a triangulated mesh, which is documented under “Gridding.”

In practice, the typical ALPS lidar processing workflow incorporates significant manual editing. ALPS provides various manual editing tools that allow analysts to improve the quality of the point cloud, as is described under “Manual Editing.”

Slant Range Measurement

The time interval between the transmit pulse centroid and the various parts of the return waveform represent the slant range time of flight values. Slant range time of flight value is the time it takes the laser light to reach the target and then bounce back to the detector. Slant range time is directly proportional to the target’s range in a given medium of air or water. This measurement is then converted into a distance measurement using the speed of light in air. For submerged targets, this distance is corrected as described in “Water Corrections.”

The specifics of this calculation are closely tied to how EAARL–B digitizes waveforms. The waveform digitizer is started 200 ns before the laser is triggered to ensure the emitted pulse is captured and to ensure jitter in the laser trigger does not cause the system to miss the transmit pulse. The exiting transmit-pulse waveform is captured as soon as the laser pulse is emitted. Once the waveform digitizer is started, it continues digitizing each of the four return channels for 16,384 ns each. Once the digitization completes, the computer selects and transfers selected portions of each waveform to its local memory via direct memory access. There are 64 samples (1 ns per sample) extracted for the transmit pulse, and 1,500 samples are extracted for each of the four return waveforms.

In computer memory, each of the four 1,500 sample returns is further processed with real-time software to reduce the length to a maximum of 450 samples. The 64-sample transmit waveform is processed and reduced to 16 samples. At this point, the transmit and return waveforms are packetized and queued for recording to a solid-state disk. Along with the waveforms, two additional data types are collected: the timestamp corresponding to the start of the transmit waveform and the time interval between the start of the transmit waveform and the start of the return waveform. These times allow for reconstruction of the laser pulse firing timeline required for calculating the slant range time of flight (fig. 2).

Waveform Analysis

Before a point cloud is generated, the transmit and return waveforms collected by EAARL must be analyzed to derive slant range measurements. These distance measurements are later combined with the other physical measurements acquired by the system, as described in the section “Point Projection.”

The digitizer in EAARL–B samples the laser signal at 1-ns intervals and records digital counts proportional to the relative intensity of the backscattered laser light. Although the counts are uncalibrated and do not represent absolute intensity or radiance values (for example, with units of watt per steradian ($W \cdot sr^{-1}$) or watt per steradian per meter ($W \cdot sr^{-1} \cdot m^{-1}$)), they do represent changes in relative intensity in a waveform. Throughout this report, the term “intensity” refers to this uncalibrated relative intensity measurement.

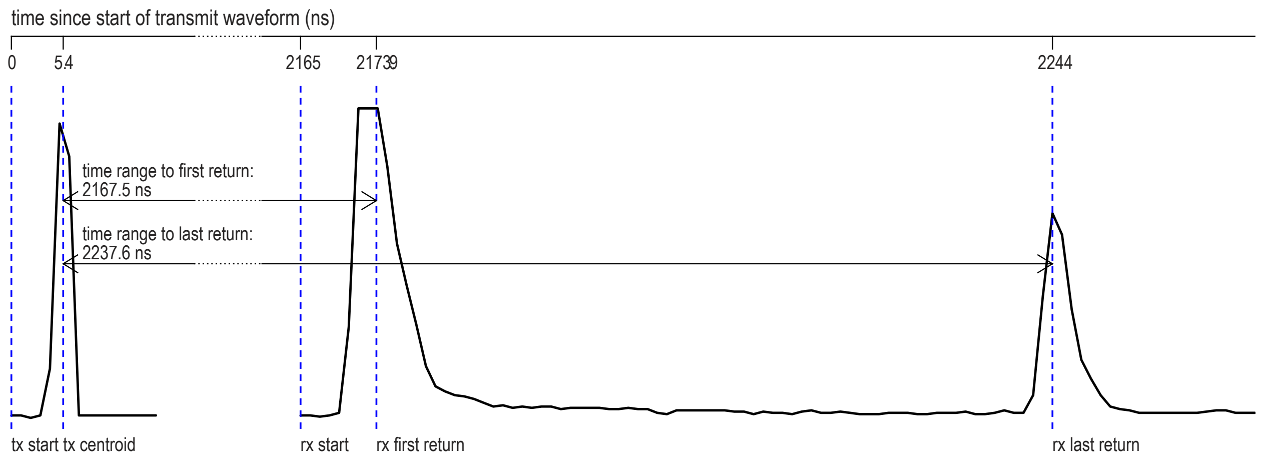


Figure 2. Sample timeline for laser pulse firing and digitization. The transmit waveform (tx) centroid occurs 5.4 nanoseconds (ns) after the start of the recorded transmit waveform. The recorded digitized return waveform (rx) starts at 2165 ns. The first return peak is at 2173.9 ns, which corresponds to a slant range time of 2167.5 ns from the centroid of the transmitted pulse. The last return peak is at 2244 ns, which corresponds to a slant range time of 2237.6 ns.

The transmit and return waveforms have decidedly different characteristics and processing algorithms. The transmit waveform captures the shape of the emitted laser pulse, and the energy center of the pulse is derived by using a simple centroid calculation on the transmit waveform. This process is documented in the "Centroid Analysis" section below. The return waveform represents the energy distribution over time of light scattered back to the sensor. Different environments result in waveforms with different characteristics, so ALPS provides three algorithms for use in waveform analysis. Each algorithm is designed for use with specific environments, and they are described in the sections "Centroid Analysis," "Leading Edge Detection," and "Bottom Detection Using Water-Column Backscatter Modeling." All three algorithms are used to determine a position within the waveform that corresponds to a target. Common targets are the ground (either submerged or subaerial), water surfaces, and the top of vegetation canopies. Centroid analysis is only suited for processing first surfaces (such as unvegetated, level ground; water surfaces; or the top of vegetation canopies). Leading edge detection is suited for topography underneath vegetation and for shallow, submerged topography. Bottom detection using water-column backscatter modeling is suited to submerged topography, especially in deeper waters.

Centroid Analysis

Centroid analysis is a fast algorithm best used for determining the center of energy in a pulse with a well-defined peak. The laser's transmit and return waveforms are time-series of relative light-intensity values. The centroid is calculated against the area under the curve (fig. 3). The algorithm has three components:

1. The waveform is adjusted to remove background energy by subtracting the intensity value of the first sample from all samples.
2. For first return analysis, the return waveform is truncated to 12 samples to restrict the portion of the waveform analyzed. (Transmit waveforms are not truncated.)
3. The centroid is calculated.

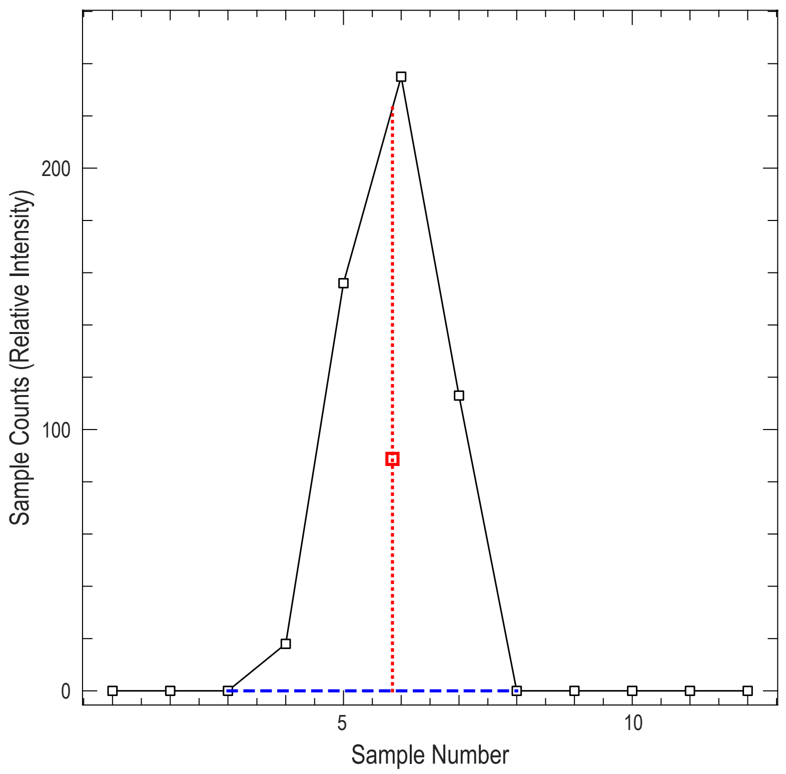


Figure 3. A waveform and its centroid. The solid black line is the waveform and the dashed blue line shows where the rest of the region is bounded. The black squares indicate the discrete intensity values. The red square is the centroid, and the vertical dotted red line shows the position used in centroid analysis.

Transmit and return waveforms digitized by EAARL contain a significant level of background energy from the system and the environment. All analysis techniques remove this background energy, but centroid analysis is especially sensitive to background energy because it skews the centroid. The waveform in figure 3 already had the background energy removed. Figure 4 shows the same waveform before the removal of its background energy and the impact the energy has on the centroid. Bias is removed by subtracting the intensity value of the first sample in a waveform from all samples in the waveform (under the assumption that the first sample is at or near the baseline of the waveform).

For first return analysis, the next step is truncation. Identifying the centroid of transmit waveforms is straightforward because they represent the shape of a single pulse. The processing of return waveforms is more complicated than the processing of transmit waveforms because the energy from multiple targets can be convolved in a return waveform. However, the first return in the received waveform typically has a much higher energy level than other returns and is always located in the first 12 samples of the waveform due to the EAARL–B system design. The EAARL–B real-time software writes out the waveform such that it has 3–5 samples before impact to permit background bias estimation. Initially, a transmitted pulse is approximately 0.9-ns wide, but it is stretched by system components (photomultiplier tube, cables, electronics, digitizer) and the environment. EAARL–B has a sampling interval of 1 ns, so 12 samples provide 3–5 ns for the background energy and 7–9 ns for the return pulse from the first surface. Thus, first returns are derived by restricting centroid analysis to the first 12 samples of the waveform to ensure that subsequent returns do not significantly impact the centroid calculation.

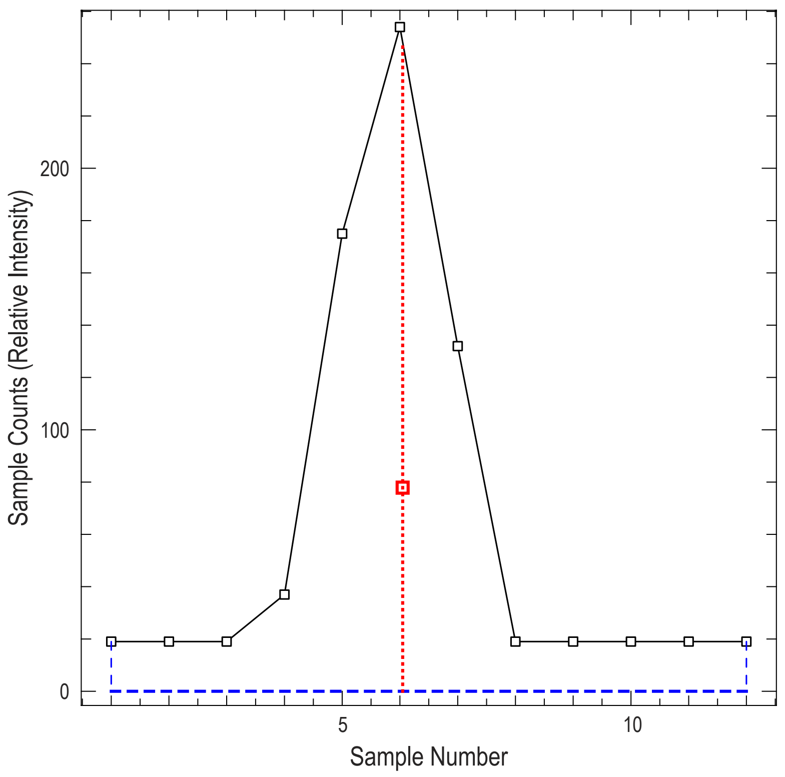


Figure 4. A sample waveform and its centroid illustrating the effect of unremoved background energy. The solid black line is the waveform, and the dashed blue lines show where the rest of the region is bounded. The black squares indicate the discrete intensity values. The red square is the centroid, and the vertical dotted red line shows the position used in centroid analysis.

The final step is the centroid calculation, which is defined mathematically as

$$C = \frac{\sum_{i=1}^{n} i \cdot wf_i}{\sum_{i=1}^{n} wf_i} \qquad (1)$$

where	C	is the floating-point position of the centroid in the waveform
	n	is the number of samples in *wf*
	i	varies from 1 to n
	wf	is a sequence of waveform sample intensity values

Leading Edge Detection

Leading edge detection is primarily used for processing topography under vegetation; however, it is also used to process shallow, submerged topography. Leading edge detection attempts to discriminate between convolved signals in the waveform, allowing for the determination of a last return even in situations where additional signal returns might be above the topographic surface, as from vegetation or the water column. The algorithm has three components:

1. The waveform is adjusted to remove background energy using the same approach described for centroid analysis (meaning the value of the first sample is subtracted from all samples in the waveform).
2. The waveform is optionally smoothed as described in "Smooth Waveform" to mitigate the effect of noise.
3. The leading edge is determined as described in "Leading Edge."

Smooth Waveform

The smooth waveform algorithm (table 1) mitigates the effect of noise in the waveform, reducing the likelihood of false positives in leading edge determination. The smooth waveform algorithm is described by the following function and parameters:

$$\mathit{smoothed} = \text{smooth\_waveform}(\mathit{wf}, \mathit{factor}) \qquad (2)$$

smoothed	sequence of smoothed waveform intensity values
wf	sequence of waveform sample intensity values
factor	number of neighbor samples to use on each side of a sample for averaging

Table 1. Smooth Waveform Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let $nwf$ be the number of samples in $wf$
Let $smoothed$ be a copy of $wf$
For each sample in $wf$ that has at least $factor$ adjacent neighbors on each side:
    Set the corresponding sample in $smoothed$ to the average of that sample and its $factor$ neighbors on each side
For each sample in $wf$ that has fewer than $factor$ adjacent neighbors on either side:
    Set $pts$ to the smallest count of neighbors on either side of that sample
    Set the corresponding sample in $smoothed$ to the average of that sample and its $pts$ neighbors on each side
Return $smoothed$
```

The algorithm applies a moving-average function to the sample values in a window that slides through *wf*. Each output sample is the average of the corresponding input sample and its *factor* neighbors on either side, with all samples weighted equally (via a boxcar filter). If there are fewer neighbors than are called for (as happens near the start and end of the waveform), then *factor* is reduced for those samples to conform to the available neighborhood. This smoothing step is optional and often omitted; inclusion and parameters are determined experimentally by the analyst processing the data.

Figure 5 illustrates how the algorithm works in practice. The figure shows a close-up of a 120-sample range in the middle of a waveform. The red line is the original waveform and has one clear peak as well as an excess of jitter due to noise. The blue line is the result of the smooth waveform algorithm (using *factor*=2) and shows that the peak is retained, but the noise jitter is reduced significantly.

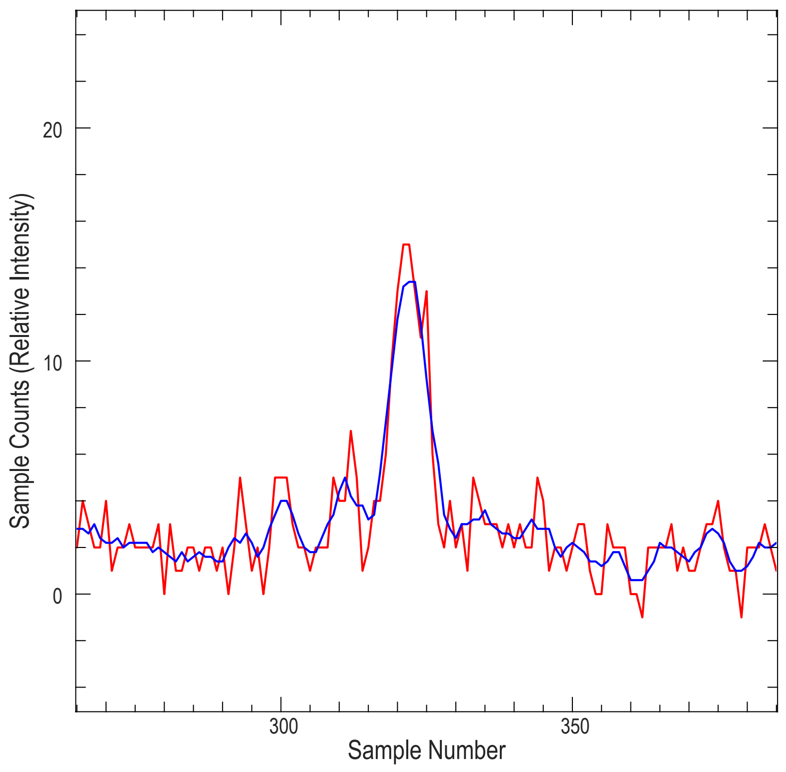


Figure 5. A close-up of a 120-sample range of a noisy waveform (in red) and the result of smoothing it with the smooth waveform algorithm (in blue). Algorithm parameter is *factor*=2.

Leading Edge

The leading edge algorithm (table 2) uses a first-difference approach to analyze the waveform. This approach often permits the differentiation of convolved signals. The leading edge algorithm is described by the following function and parameters:

$$position = \text{leading\_edge}(wf, thresh, noiseadj) \tag{3}$$

position	position in the waveform of the final peak
wf	sequence of waveform sample intensity values
thresh	threshold value the peak must exceed for screening out noise
noiseadj	Boolean value specifying whether a noise adjustment should be applied (false = no adjustment; true = adjustment)

Table 2. Leading Edge Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let wfd1 be the discrete first derivative of wf
Let edges be the indexes into wfd1 corresponding to each point where the first derivative crosses above thresh
If edges is empty, then return null result
Let last_edge be the last element in edges
Let max_return_length be 18 samples
If last_edge + max_return_length exceeds the size of wf, then reduce max_return_length to wf - last_edge
If max_return_length is less than 5, then return null result (assumed to be noise)
If noiseadj is true:
    If any of the values of wfd1 in the range with indices last_edge to last_edge+3 falls below 0:
        Advance last_edge to the first such value (assumed to be noise, so advance beyond it)
        Reduce max_return_length again if necessary to avoid exceeding the size of wf, as above
If any of the values of wfd1 in the range with indices last_edge+1 to last_edge+max_return_length falls below 0:
    Return the index of the first such value
Otherwise:
    Return null result
```

The discrete first derivative of the waveform (calculated using forward differences) is used to locate the leading edges of local peaks, which are the locations where the derivative is significantly positive (as determined by *thresh*). The waveform is then traversed after the last leading edge to locate the first location where the waveform changes from ascending to descending (forming a local peak) and that location is interpreted as corresponding with the last return (fig. 6).

When *noiseadj* is enabled the samples immediately after the detected peak are analyzed to compensate for the effect of noise. Sometimes a leading edge is followed by a small dip that is then followed by another leading edge. If the dip is below *thresh*, then the second leading edge is masked by the preceding leading edge. The *noiseadj* check detects this situation and advances the algorithm past the dip so it can find the peak after the latter leading edge. Without *noiseadj* enabled, the local peak of the dip would be interpreted as the last return.

Bottom Detection Using Water-Column Backscatter Modeling

Bottom detection using water-column backscatter modeling is the most complex waveform analysis workflow and is used exclusively on submerged topography. The algorithm compensates for water column backscatter and absorption by modeling them and subtracting the model from the waveform signal. A first derivative approach is used to locate the final peak, then heuristics are used to validate the peak. The components of the algorithm are described more fully in the following sections, but below is a summary of the steps:

1. The waveform is adjusted to remove background energy. Unlike centroid analysis and leading edge detection, background energy for this algorithm is removed by finding the minimum value in the first fifteen samples and subtracting that minimum from all samples. The algorithms used are sensitive to noise, and using the first fifteen samples to derive the background energy makes the process more resilient against bias.
2. The waveform is optionally smoothed as described in “Smooth Waveform” under “Leading Edge Detection” (by using a moving average).

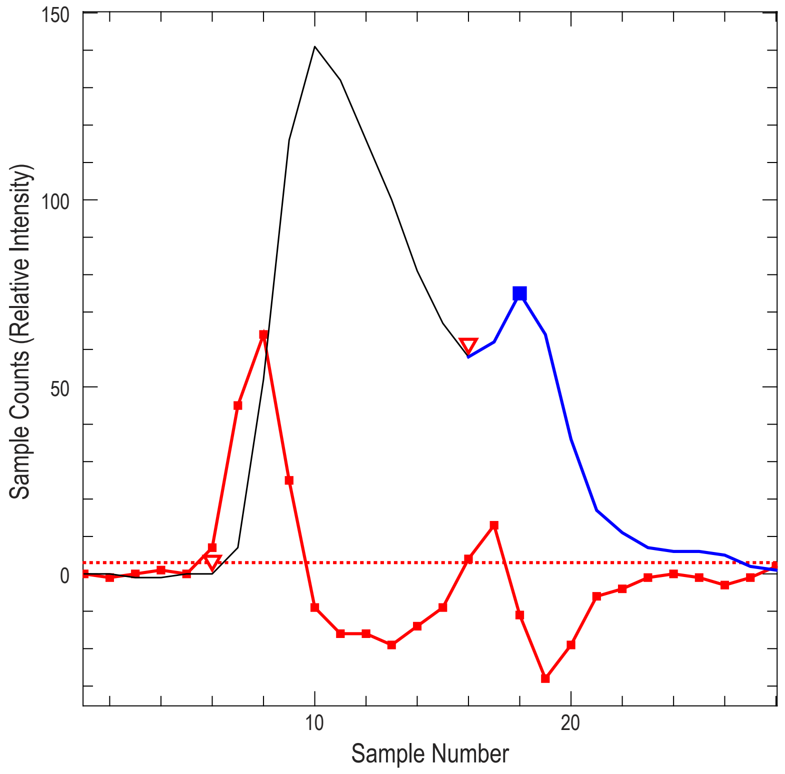


Figure 6. Leading edge algorithm. The solid black and blue lines represent the waveform. The solid red line is the first derivative of the waveform with discrete values indicated by red squares. The dotted red line is the threshold. Leading edges are marked with red triangles. The blue line highlights the section searched for a peak (following the final leading edge), and the peak found is marked with a blue square. Algorithm parameters were *thresh*=4, *noiseadj*=false.

3. The water surface is detected using a first derivative approach as described in “Detect Surface” below. This surface location is only used for determining where to start applying the signal decay compensation and where to begin normalization of the modeled water column and laser attenuation.

4. The waveform is adjusted to compensate for signal decay in the water column using manually calibrated parameters. The decay is modeled in one of two ways: as an exponential curve or as a log-normal curve. An automatic gain control (AGC) function is applied after the decay curve to modulate the amplification of the signal as a function of depth to help suppress wild excursions in the upper part of the water column. These two algorithms are described below in “Compensate for Exponential Decay” and “Compensate for Log-Normal Decay.”

5. The bottom is detected as described in “Detect Bottom” below. Noise in the tail is temporarily removed (as described in “Remove Noisy Tail”). The final peak is then detected using a manually-calibrated threshold crossing and the first derivative (as described in “Extract Peaks with First Derivative”).

6. The peak is checked to see if it is saturated. If the peak is saturated, the location of the peak is tuned to compensate for the saturation as described in “Saturation Check.”

7. The peak is validated using heuristics that analyze the shape of the waveform at the detected bottom location as described in “Validate Bottom.”

The bottom peak location is then used for the bottom location, if it passes validation, and is paired with a water surface location as derived by the centroid algorithm.

Detect Surface

The detect surface algorithm (table 3) provides an approximate surface location that is used solely for applying the decay curve and is optimized for that purpose. This algorithm is described by the following function and parameters:

$$position = \text{detect\_surface}(wf, maxint, sfc\_last, wantlen) \tag{4}$$

position	position in waveform where surface peak is located
wf	sequence of waveform sample intensity values
maxint	intensity value that represents saturation
sfc_last	farthest distance into the waveform to look for a saturated surface
wantlen	number of samples to analyze in the waveform if no saturation is found

Table 3. Detect Surface Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let saturated be an index list showing which samples in wf are equal to maxint
Let numsat be the number of items in saturated
Let nwf be the number of samples in wf
If numsat is greater than 1 and the first element in saturated is less than or equal to sfc_last:
    Let surface be the last value in the series of consecutive integers at the beginning of saturated
Otherwise:
    If nwf is greater than wantlen + 8:
        Let surface be the index of the maximum value of the first wantlen samples in wf
    Otherwise:
        Let surface be the smaller of wantlen and nwf
Return surface
```

This procedure does not necessarily determine the exact location of the surface. The decay algorithms seek to fit a model to the submerged backscatter and only require the trailing edge of the surface return. In the case of a saturated waveform, the procedure places the surface at the end of the saturated section; the surface is actually earlier in the waveform, but the model would be skewed by the inclusion of the saturated section, so it is omitted. Non-saturated waveforms use the maximal peak. Very short waveforms (those shorter than *wantlen* + 8) use a dummy value.

Compensate for Exponential Decay

The exponential decay compensation algorithm (table 4) fits an exponential decay model to the backscatter in a shallow-water, narrow-beam-channel waveform. The algorithm is described by the following function and parameters:

wf_decay = compensate_decay_exp(*wf*, *maxint*, *laser*, *water*, *surface*, *agc_factor*) (5)

wf_decay	output sequence of waveform intensity values that have been adjusted to compensate for decay
wf	sequence of waveform sample intensity values
maxint	maximum intensity
laser	modeling factor for the inherent decay of the laser
water	modeling factor for decay due to water column absorption and backscatter
surface	location to use for the surface
agc_factor	automatic gain control factor

Table 4. Compensate for Exponential Decay Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let nwf be the number of samples in wf
Let C_H2O be the speed of light through water in m/ns (approximately 0.22490)
Let attdepth be the first nwf elements in the series 0, 1*C_H2O*.5, 2*C_H2O*.5, 3*C_H2O*.5, …

Let laser_decay be exp(laser * attdepth) * maxint
Let secondary_decay be exp(water * attdepth) * maxint

Let decay be laser_decay + 0.25 * secondary_decay
Let agc be 1 - exp(agc_factor * attdepth)

Let shift be surface - 1
If shift is greater than 0:
    Drop the last shift values of decay
    Prepend a list of shift copies of maxint to decay
    Replace indices shift and shift+1 in decay with maxint
    Drop the last shift values of agc
    Prepend a list of shift copies of 0 to agc
    Replace index shift in agc with 0

Let bias be (1 - agc) * -5
Let wf_decay be (wf - decay) * agc + bias

Return wf_decay
```

A significant amount of light is reflected back by particulate material suspended in the water column. This backscatter contributes a gradually decreasing intensity level to the waveform that is convolved with the bottom return response, potentially masking it. The algorithm compensates for the decreasing intensity level by applying three cumulative adjustments to the waveform (fig. 7). The first and second adjustments use an exponential model to account for the characteristics of the laser and water column, respectively. The third adjustment applies an AGC correction.

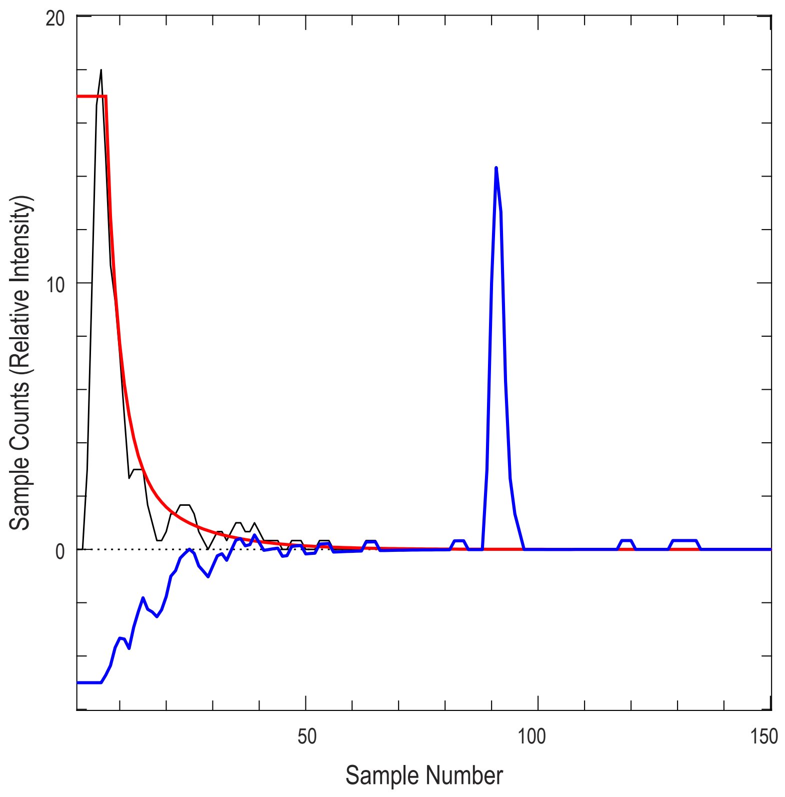


Figure 7. Submerged topography waveform as adjusted using the exponential decay model. The solid black line is the waveform prior to adjustment. The solid red line is the combined decay model (laser and water) that was fit to the waveform. The solid blue line is the waveform as adjusted by the algorithm. The dotted red line indicates the zero sample count level. Algorithm parameters were *laser*=-2.9, *water*=-0.7, *agc_factor*=-1.0.

Compensate for Log-Normal Decay

The log-normal decay compensation algorithm (table 5) fits a log-normal model to the backscatter in the waveform. The algorithm is described by the following function and parameters:

wf_decay = compensate_decay_lognorm(*wf*, *mean*, *stdev*, *mean*, *xshift*, *xscale*, *tiepoint*, *agc_factor*) (6)

wf_decay	output sequence of waveform sample intensity values that have been adjusted to compensate for decay
wf	sequence of waveform sample intensity values
mean	mean value for the log-normal curve
stdev	standard deviation for the log-normal curve
xshift	x-axis shifting factor
xscale	x-axis scaling factor
tiepoint	position in the waveform that should intersect with the log-normal curve
agc_factor	automatic gain control factor

Table 5. Compensate for Log-Normal Decay Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let nwf be the number of samples in wf
If nwf < tiepoint:
    Return a list of nwf copies of 0

Let C_H2O be the speed of light through water in m/ns (approximately 0.22490)
Let attdepth be the first nwf elements in the series 0, 1*C_H2O*.5, 2*C_H2O*.5, 3*C_H2O*.5, …

Let x be the first nwf positive integers
Let x be (x - xshift) / xscale

Let decay be log_normal(x, mean, stdev)
Let factor be the value of wf at index tiepoint divided by the value of decay at index tiepoint
Let decay be decay * factor

Let agc be 1 - exp(agc_factor * attdepth)
Let shift be surface - 1
If shift is greater than 0:
    Drop the last shift values of agc
    Prepend a list of shift copies of 0 to agc
    Replace index shift in agc with 0

Let bias be (1 - agc) * -5
Let wf_decay be (wf - decay) * agc + bias

Return wf_decay
```

As with the previous section, the backscatter in the water column is modeled and removed from the waveform (fig. 8). This algorithm fits a single log-normal curve instead of two exponential curves. The values of the log-normal curve are scaled to intersect with the waveform at the location of *tiepoint* in the waveform. As with the exponential decay model, an additional adjustment applies an AGC correction. The log-normal function is the best model for the EAARL–B deep water channel due to the nature of the optical design of the channel.

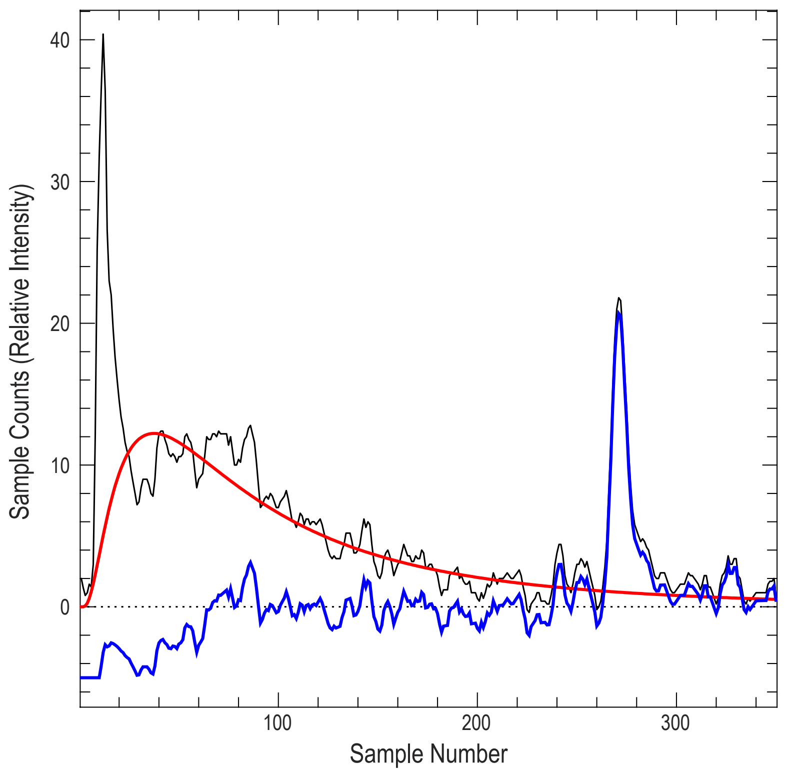
The algorithm makes use of a helper function log_normal. The log_normal function implements the equation for the probability density function of the log-normal distribution (Norstad, 2011):

$$\text{log\_normal}(x,\mu,\sigma)=\frac{1}{x\sigma\sqrt{2\pi}}e^{\frac{(\ln x-\mu)^2}{2\sigma^2}} \tag{7}$$

where x is a standard normal variable

μ is the mean of the distribution

σ is the standard deviation of the distribution



Figute 8. Submerged topography waveform as adjusted using the log-normal decay model. The solid black line is the waveform prior to the algorithm. The solid red line is the decay model that was fit to the waveform. The solid blue line is the waveform as adjusted by the algorithm. The dotted red line indicates the zero sample count level. Algorithm parameters were *mean*=1.7, *stdev*=0.9, *agc_factor*=-0.2, *xshift*=1, *xscale*=15, *tiepoint*=40.

Detect Bottom

The bottom-detection algorithm (table 6) uses a first-derivative-peak extraction approach over a subsection of the waveform to determine the bottom location. The algorithm is described by the following function and parameters:

$$bottom = \text{detect\_bottom}(wf, first, last, thresh) \qquad (8)$$

bottom	position in waveform where bottom is located
wf	sequence of waveform sample intensity values
first	starting position in the waveform to look for a peak
last	ending position in the waveform to look for a peak
thresh	noise threshold

Table 6. Detect Bottom Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let nwf be the number of samples in wf
Let offset be first - 1
Let last be the smaller of last or nwf

Let wfaoi be the values in wf from index first to index last
Let last be offset + remove_noisy_tail(wfaoi, thresh)
If last - first is less than 4:
    Return -1

Let wfaoi be the values in wf from index first to index last
Let peaks be extract_peaks_first_deriv(wfaoi, thresh)
If peaks is empty:
    Return -1

Let bottom be offset plus the value of the last element in peaks
Return bottom
```

Only the specified range of the waveform is analyzed for a possible bottom: the *first* and *last* parameters must be tuned based on the characteristics of the area being processed. The tail portion of that region is truncated to remove noise if noise is present. Then the remaining range is subjected to the first derivative analysis. The algorithms used for the noisy tail removal and the peak detection are described in the following subsections, "Remove Noisy Tail" and "Extract Peaks with First Derivative."

Remove Noisy Tail

The remove noisy tail algorithm (table 7) trims off the tail portion of the waveform where all samples fall below *thresh*, a noise threshold. This simple approach helps prevent minor noise fluctuations from appearing as false bottom signals. The algorithm is described by the following function and parameters:

$$last = \text{remove\_noisy\_tail}(wf, thresh) \qquad (9)$$

last	last non-noise position of the waveform
wf	sequence of waveform sample intensity values
thresh	noise threshold

Table 7. Remove Noisy Tail Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let nwf be the number of samples in wf
Let wfmin be the smallest value in wf
Let w be the indices into wf where $wf > wfmin + thresh$
If w is empty:
    Return nwf
Let end be 1 plus the value of the last element in w
Return the smaller of nwf or end
```

Extract Peaks with First Derivative

The peak extraction algorithm (table 8) locates peaks using the discrete first derivative of the waveform. The algorithm is described by the following function and parameters.

$$positions = \text{extract\_peaks\_first\_deriv}(wf, thresh) \tag{10}$$

positions	sequence of positions in waveform where peaks were found
wf	sequence of waveform sample intensity values
thresh	noise threshold

Table 8. Extract Peaks with First Derivative Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let wfd1 be the discrete first derivative of wf
Let wfd1 be wfd1 - 0.05 (to avoid false positives due to the effect of AGC)

Let wfd1_sign be sign(wfd1) (where the sign function returns -1, 0, or 1 for values that are negative, zero, or positive)
Let wfd1_sign_d1 be the discrete first derivative of wfd1_sign
Let peaks be the indices into wfd1_sign_d1 with value equal to 2
Let peaks be peaks + 1 (needed to correct indices to reference wf due to successive derivative operations)

Let wfpeaks be the values of wf at the indices in peaks
Let idx be the indices into wfpeaks with values greater than or equal to thresh
If idx is empty:
    Return an empty list
Let peaks be the values in peaks corresponding to idx
Return peaks
```

Peaks are located by looking for zero-crossings of the discrete first derivative. The peaks whose magnitudes match or exceed *thresh* are returned as candidate target locations (fig. 9).

Saturation Check

The saturation check algorithm (table 9) compensates for the effect of saturated waveforms. The algorithm is described by the following function and parameters:

$$bottom\_new = \text{saturation\_check}(saturated, bottom) \tag{11}$$

bottom_new	revised bottom location
saturated	sequence of Boolean values indicating which samples in the original waveform were saturated (true = saturated; false = not saturated)
bottom	original bottom location

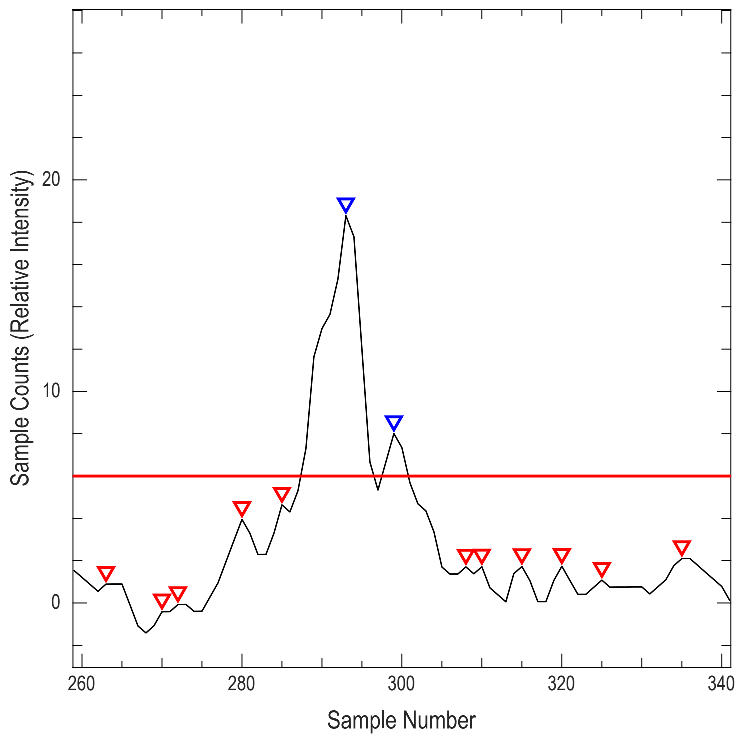


Figure 9. Peaks as extracted using the discrete first derivative. The black line is the waveform. The red line is the threshold. Red triangles are peaks that were rejected because they are below the threshold. Blue triangles are peaks that passed because they are above the threshold. Algorithm parameter used was *thresh*=6.

Table 9. Saturation Check Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let nwf be the number of samples in wf
Let is_sat be the value of saturated at index bottom

If is_sat is false and the value of saturated at index bottom-1 is true:
    (This means that AGC pushed the peak off the saturated section by a sample)
    Let is_sat be true
    Decrement bottom by 1

Let sat0 and sat1 each be bottom
While sat0 is greater than 1 and the value of saturated at index sat0-1 is true:
    Decrement sat0 by 1
While sat1 is less than nwf and the value of saturated at index sat1+1 is true:
    Increment sat1 by 1

Return (sat0+sat1)/2 truncated to an integer
```

The EAARL–B digitizer has a fixed signal range in which it can detect. If the signal intensity saturates the digitizer by exceeding that range, then the recorded signal values are at their maximum. Consequently, a strong peak may have its top flattened. If the bottom occurs in a saturated portion of the waveform, the bottom should be placed at the center of the saturated segment to approximate better the true location of the peak. This algorithm detects whether the saturation condition has occurred and updates the bottom accordingly. If no saturation is detected, then no change occurs.

Validate Bottom

The bottom validation algorithm (table 10) uses pulse heuristics to accept or reject the bottom based on the shape of the peak. The algorithm is described by the following function and parameters:

$$valid = \text{bottom\_validate}(wf, bottom, thresh, first, last, lw\_dist, rw\_dist, lw\_factor, rw\_factor) \quad (12)$$

valid	Boolean value indicating whether the bottom is valid (true = valid; false = invalid)
wf	sequence of waveform sample intensity values
bottom	position in the waveform detected as the bottom
thresh	threshold that the intensity value of a valid peak has to exceed
first	starting position in the waveform for valid peaks
last	ending position in the waveform for valid peaks
lw_dist	left pulse wing distance threshold
rw_dist	right pulse wing distance threshold
lw_factor	left pulse wing scaling factor
rw_factor	right pulse wing scaling factor

Table 10. Validate Bottom Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let lw_idx be bottom – lw_dist
Let rw_idx be bottom + rw_dist
Let bottom_intensity be the value of wf at index bottom

If bottom_intensity <= thresh or last < rw_idx:
    Return false

If lw_idx < first or rw_idx > last:
    Return false

Let lw_thresh be lw_factor * bottom_intensity
Let rw_thresh be rw_factor * bottom_intensity

Let lw_intensity be the value of wf at index lw_idx
Let rw_intensity be the value of wf at index rw_idx

If lw_intensity > lw_thresh or rw_intensity > rw_thresh:
    Return false

Return true
```

The bottom validation algorithm makes sure that the detected bottom is in the correct range of the waveform (specified by *first* and *last*) and exceeds a minimum intensity threshold (specified by *thresh*). The shape of the peak is then analyzed to ensure it fits within the bounds defined by the four pulse wing parameters (*lw_dist*, *rw_dist*, *lw_factor*, and *rw_factor*). Pulse wings are the sections of the waveform immediately before or after the peak.

Figures 10 and 11 illustrate how bottom validation works. A detected bottom must have an intensity above the red *thresh* line and the curve of the waveform must be below both pulse wing thresholds (magenta markers) to pass validation. Figure 10 shows a bottom detected using *thresh*=6 that fails validation because its left pulse wing threshold is beneath the curve of the waveform. Figure 11 shows a bottom detected using *thresh*=9 that passes validation because both pulse wing thresholds are above the curve of the waveform and the intensity of the bottom is above *thresh*.

Point Projection

The exact location of a georeferenced point representing a target surface detected by the system is derived from a combination of many data sources:

- The distance from the mirror to the target, as determined by waveform analysis (fig. 2).
- Known reference position data from one or more GPS base stations (fig. 12).
- Position and attitude (rotational) measurements for the aircraft, acquired by a NovaTel SPAN–CPT unit (fig. 13).
- Manual calibration measurements for the positional offsets between the SPAN–CPT unit and the GPS antenna and between the SPAN–CPT unit and the mirror (fig. 13).
- Manual calibration measurements for the rotational offsets between the SPAN–CPT unit and the mirror and between the SPAN–CPT unit and the laser.
- The current angle of the oscillating mirror, provided by the mirror instrumentation (fig. 12).

The NovAtel SPAN–CPT unit is a GPS-assisted INS consisting of a kinematic dual-frequency GPS receiver tightly coupled with an inertial measurement unit composed of a laser fiber optic gyro and a set of precision microelectromechanical system accelerometers operating in the X, Y, and Z planes. The unit provides position and attitude measurements. Attitude consists of roll, pitch, and yaw: roll is the rotation about the longitudinal or y-axis of the airplane, pitch is the rotation about the lateral or x-axis of the airplane, and yaw is the rotation about the vertical or z-axis of the airplane. GPS base station data are submitted to the Online Positioning User Service to derive accurate positions for the base stations (National Geodetic Survey, 2014). Commercial GPS and INS post-processing software (NovAtel Inertial Explorer) is used to improve the quality of the position and attitude measurements using the GPS base station data and the offset between the SPAN–CPT unit and the onboard GPS antenna. This process provides trajectory data consisting of high-quality position and attitude data for the location and orientation of the SPAN–CPT unit.

Within ALPS, the time measurements from waveform analysis (laser range, position in transmit waveform, and position in return waveform) are combined and converted to meters (using the speed of light through air) to determine the magnitude of the range vector between the mirror and the target. The aircraft position, the equipment offset vectors, and the various rotations are then applied to the range vector to derive the real-world coordinate location of the target in Universal Transverse Mercator

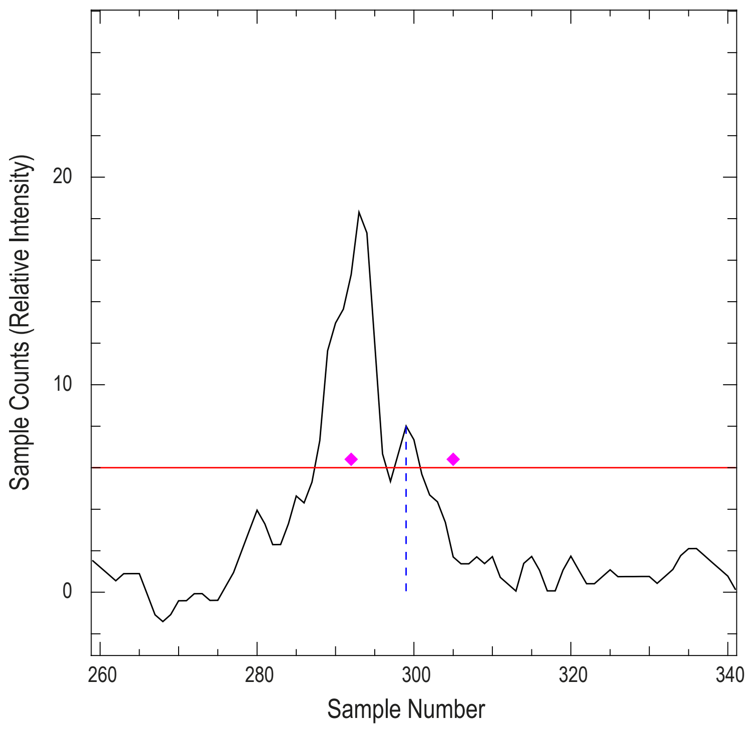


Figure 10. Bottom return that fails bottom validation. The waveform is in black, and the threshold is in red. The dashed blue line indicates the bottom that was detected. The magenta markers represent the pulse wing thresholds. The right pulse wing threshold passes, but the left pulse wing threshold fails as it is beneath the curve of the waveform. Algorithm parameters used were *lw_dist*=3, *lw_factor*=0.7, *rw_dist*=4, *rw_factor*=0.7, *first*=15, *last*=220, *thresh*=6.

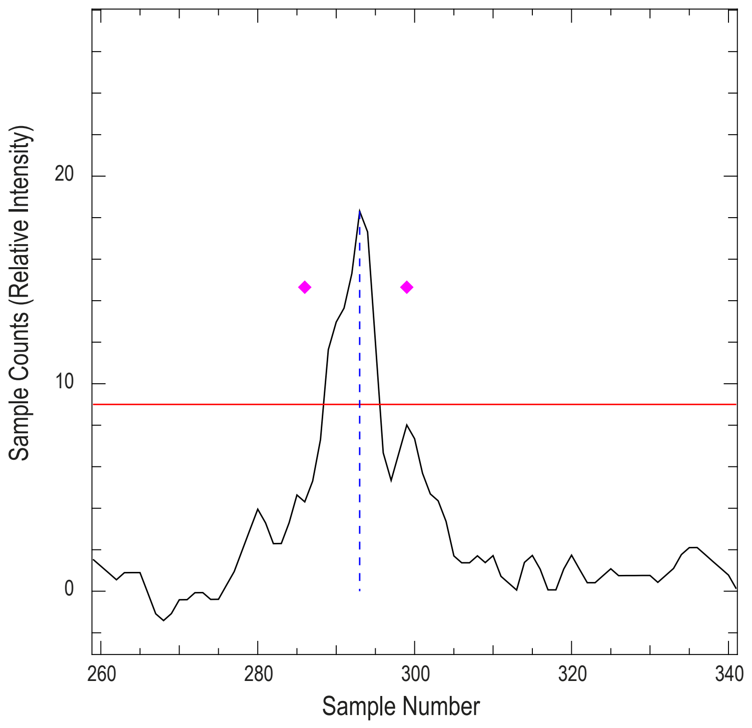


Figure 11. Bottom return that passes bottom validation. This figure shows the same waveform as depicted in figure 10 but used *thresh*=9 instead of *thresh*=6. Both of the magenta pulse wing thresholds are above the curve of the waveform.

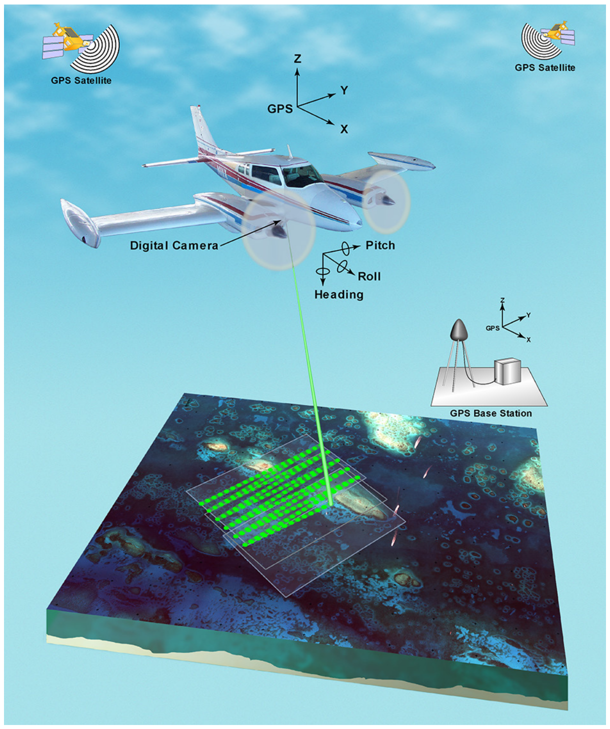


Figure 12. Experimental Advanced Airborne Research Lidar (EAARL) system in flight with oscillating scan pattern, plus external data sources.

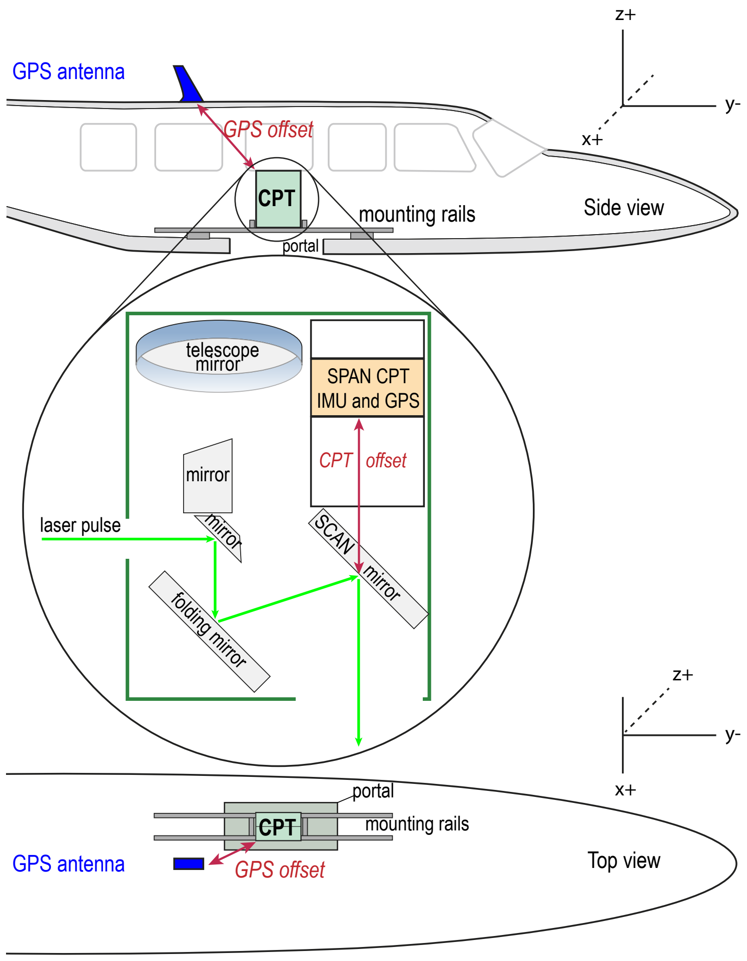


Figure 13. Experimental Advanced Airborne Research Lidar (EAARL) system components as located on an airplane.

(UTM) coordinates. If the target is submerged, then the target location must also be adjusted to compensate for refraction. The following sections detail the math and algorithms involved in transforming the data into a final real-world point location, as summarized below:

1. Rotation matrices are derived from the angles of rotation used when transforming frames of reference: from aircraft to real-world (using INS attitude data), from mirror to aircraft (using fixed mounting bias angles for the mirror assembly with respect to the airplane), and from laser to aircraft (using fixed mounting bias angles for the laser with respect to the airplane).
2. The location of the mirror is transformed from the aircraft's frame of reference to the real-world frame of reference using the aircraft-to-real-world rotation matrix, the position of the GPS system, and the known offset vector from the GPS system to the mirror.
3. The incidence direction vector for the laser is transformed from the laser's frame of reference to the aircraft's frame of reference using the laser-to-aircraft rotation matrix. The vector is then further transformed to the real-world frame of reference using the aircraft-to-real-world rotation matrix.
4. The normal direction vector of the mirror is transformed from the mirror's frame of reference to the aircraft's frame of reference using the mirror-to-aircraft rotation matrix. The vector is then is further transformed to the real-world frame of reference using the aircraft-to-real-world rotation matrix.
5. The reflection-direction vector of the laser is calculated by the law of specular reflection using the laser-incidence-direction vector and the mirror-normal-direction vector.
6. The target location is derived using the known mirror position, the known range from the mirror to the target, and the known laser-reflection-direction vector.
7. For submerged targets, the water-correction algorithm is applied to account for the different refractive indices for air and water.

Rotation Matrices

Rotations in EAARL are recorded and applied as a series of extrinsic Tait–Bryan angles in z-x-y order. These rotations are applied using rotation matrices (Wittenburg, 2008). Rotation matrices about the x, y, and z axes are defined as—

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{13}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{14}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

where θ is the angle of rotation about the specified axis
These matrices are then combined using matrix multiplication to derive a final rotation matrix:

$$R(\alpha,\beta,\gamma) = R_z(\gamma)R_x(\alpha)R_y(\beta) \tag{16}$$

where α is the angle of rotation about the *x* axis

β is the angle of rotation about the *y* axis

γ is the angle of rotation about the *z* axis

In total, three sets of rotation matrices are used in deriving the target point location:

- The roll, pitch, and yaw of the aircraft (provided by the INS unit) are used to transform from the aircraft's frame of reference to the real-world frame of reference.
- The fixed-mounting-bias angles of the mirror assembly, with respect to the airplane, combined with the scan angle of the oscillating mirror, as reported by the mirror instrumentation, are used to transform from the mirror's frame of reference to the aircraft's local frame of reference. The scan angle of the mirror is simply added to the y-axis mounting value.
- The fixed-mounting-bias angles of the laser, with respect to the airplane, are used to transform from the laser's frame of reference to the aircraft's local frame of reference.

Derivation of Mirror Location

The mirror location (representing the location on the scanning mirror where the laser hits to reflect) is known to be at a fixed location relative to the GPS system. To render that location in real-world coordinates, a three-dimensional conformal transformation, consisting of a rotation and a translation, must be applied. The equation for these transformations is—

$$\mathbf{p_m} = \mathbf{R_{ar}}\mathbf{s_{gm}} + \mathbf{p_g} \tag{17}$$

where $\mathbf{p_m}$ is the position vector of the mirror in real-world coordinates

$\mathbf{R_{ar}}$ is the rotation matrix to transform from the aircraft's frame of reference to the real-world frame of reference

$\mathbf{s_{gm}}$ is the offset vector from the GPS system to the mirror, in the aircraft's frame of reference

$\mathbf{p_g}$ is the position vector of the GPS system in real-world coordinates

Incidence Vector for Laser

In the laser's frame of reference, the laser beam is projected perpendicularly below an imaginary plane formed by the *x* and *z* axes of the frame of reference. The incidence vector of the laser is thus represented by a unit vector that is -1 unit in the *y* direction. This unit vector is changed to the real-world frame of reference by applying a sequence of two rotational transformations, placing it first into the aircraft's frame of reference and then into the real-world frame of reference. The equation for the real-world incidence vector for the laser is

$$\hat{\mathbf{d}}_{\mathbf{i}} = \mathbf{R}_{\mathbf{ar}}\mathbf{R}_{\mathbf{la}} \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} \tag{18}$$

where $\hat{\mathbf{d}}_{\mathbf{i}}$ is the incidence direction vector of the laser in the real-world frame of reference

$\mathbf{R}_{\mathbf{ar}}$ is the rotation matrix to transform from the aircraft's frame of reference to the real-world frame of reference

$\mathbf{R}_{\mathbf{la}}$ is the rotation matrix to transform from the laser's frame of reference to the aircraft's frame of reference

Normal Vector for Mirror

In the mirror's frame of reference, the mirror is treated as lying on the plane formed by the *x* and *y* axes of the frame of reference. The normal vector of the mirror is thus represented by a unit vector that is 1 unit in the *z* direction. This unit vector is changed to the real-world frame of reference by applying a sequence of two rotational transformations: first it is put into the aircraft's frame of reference, and then it is put into the real-world frame of reference. The equation for the real-world normal vector for the laser is

$$\hat{\mathbf{d}}_{\mathbf{n}} = \mathbf{R}_{\mathbf{ar}}\mathbf{R}_{\mathbf{ma}} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{19}$$

where $\hat{\mathbf{d}}_{\mathbf{n}}$ is the surface normal direction vector of the mirror in the real-world frame of reference

$\mathbf{R}_{\mathbf{ar}}$ is the rotation matrix to transform from the aircraft's frame of reference to the real-world frame of reference

$\mathbf{R}_{\mathbf{ma}}$ is the rotation matrix to transform from the mirror's frame of reference to the aircraft's frame of reference

Reflection Vector for Laser

The previous two sections derive a known incidence vector for the laser and a known surface normal vector for the mirror. The law of specular reflection is used to derive a unit vector of specular reflection from these two values, which provides the direction of the reflected laser that leaves the airplane (Comninos, 2006). The equation is

$$\hat{\mathbf{d}}_{\mathbf{s}} = 2(\hat{\mathbf{d}}_{\mathbf{i}} \cdot \hat{\mathbf{d}}_{\mathbf{n}})\hat{\mathbf{d}}_{\mathbf{n}} - \hat{\mathbf{d}}_{\mathbf{i}} \tag{20}$$

where $\hat{\mathbf{d}}_{\mathbf{s}}$ is the direction vector of specular reflection in the real-world frame of reference

$\hat{\mathbf{d}}_{\mathbf{i}}$ is the incidence direction vector of the laser in the real-world frame of reference

$\hat{\mathbf{d}}_{\mathbf{n}}$ is the surface normal direction vector of the mirror in the real-world frame of reference

Derivation of Target Location

Combining the resulting values of the previous sections, the target location is derived using the equation

$$\mathbf{p_t} = \hat{\mathbf{d}}_\mathbf{s} r_{mt} + \mathbf{p_m} \tag{21}$$

where $\mathbf{p_t}$ is the position vector of the target location in real-world coordinates

$\hat{\mathbf{d}}_\mathbf{s}$ is the direction vector of specular reflection in the real-world frame of reference

r_{mt} is the range (distance) from the mirror to the target

$\mathbf{p_m}$ is the position vector of the mirror in real-world coordinates

Water Corrections

The water correction algorithm (table 11) adjusts submerged returns for the difference in refractive index between the air and the water. Since light travels more slowly in water than in air, the distance measurement between the surface and the target must be recalculated using the speed of light in water instead of the speed of light in air, resulting in a smaller value. A submerged target must also be corrected for the angle of refraction using Snell's law (Feynman and others, 1964). The algorithm is described by the following function and parameters:

$$cx, cy, cz = \text{water\_correction}(sx, sy, sz, bx, by, bz) \tag{22}$$

cx, cy, cz	corrected position coordinates of the bottom
sx, sy, sz	position coordinates of the water surface
bx, by, bz	uncorrected position coordinates of the bottom

Table 11. Water correction Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let $K\_H2O$ be the refractive index for water at 20 degrees Celsius (approximately 1.333)
Let $C\_H2O$ be the speed of light through water in m/ns (approximately 0.22490)
Let $C\_AIR$ be the speed of light through air in m/ns (approximately 0.29971)

Let $dx$ be $bx - sx$
Let $dy$ be $by - sy$
Let $dz$ be $bz - sz$

Let $dist$ be the total displacement: the square root of the sum of the squares of $dx$, $dy$, and $dz$
Let $horiz$ be the horizontal displacement: the square root of the sums of the squares of $dx$ and $dy$
Let $height$ be the vertical displacement: $dz$
Let $hsign$ be sign($dz$) (where sign returns -1 where $dz$ is negative, 0 where $dz$ is 0, and 1 where $dz$ is positive)

Let $theta$ be arctan($dy$, $dx$)

If any values in $dist$ are 0, change them to $10^{-10}$ (to prevent divide by zero errors)
```

Table 11. Water correction Algorithm—Continued

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let $phi\_air$ be arcos($height$ / $dist$)
Let $phi\_water$ be arcsin(sin($phi\_air$)/$K\_H2O$)

Let $dist$ be $dist * C\_H2O / C\_AIR$

Let $height$ be cos($phi\_water$) * $dist$
Let $horiz$ be sin($phi\_water$) * $dist$

Let $cdx$ be $horiz$ * cos($theta$)
Let $cdy$ be $horiz$ * sin($theta$)
Let $cdz$ be $height * hsign$

Let $cx$ be $sx + cdx$
Let $cy$ be $sy + cdy$
Let $cz$ be $sz + cdz$

Return $cx$, $cy$, $cz$
```

Random Consensus Filter

The point cloud data are filtered using the random consensus filter (RCF), which is modeled after the concepts of the random sample consensus (RANSAC) paradigm described by Fischler and Bolles (1981). The RCF algorithm was first implemented for use on EAARL data by Wright (2002) and was first described by Nayegandhi and others (2009). The basic building block for the filtering algorithms is the one-dimensional RCF, which identifies the region of highest concentration in a one-dimensional list of values. For three-dimensional data, the gridded RCF algorithm partitions points into grid cells and then applies the one-dimensional RCF on elevation values. The multi-gridded RCF algorithm leverages the gridded RCF algorithm multiple times with offset partitioning to avoid sub-optimal results near grid lines.

One-Dimensional RCF

The basic one-dimensional RCF algorithm (table 12) is used to reduce the presence of noise. The algorithm is described by the following function and parameters:

$$low = \mathrm{rcf}(jury, width) \tag{23}$$

low	lower bound of the range that contains the highest concentration of points
jury	list of numerical values in an arbitrary order
width	width of the search window

Table 12. One-dimensional Random Consensus Filter (RCF) Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Sort jury into ascending order
Let best_vote be 0
Let best_index be 0
For each index i into jury:
    Let lower be the value corresponding to i in jury
    Let upper be lower + width
    Let j be the index of the last value in jury that is less than upper
    Let vote be $j – i + 1$
    If $vote > best\_vote$:
        Let best_vote be vote
        Let best_index be i
Return the value corresponding to best_index in jury
```

The one-dimensional RCF algorithm operates on a list of values (*jury*) by identifying the range (with the size specified by *width*) that contains the highest concentration of points. The algorithm iterates over each unique value in *jury* and determines how many values are greater than or equal to that value and also less than that value plus *width*. The value that has the highest count is the winning *low*. If there are multiple values that have the same high count, then the algorithm returns the largest value.

Gridded RCF

The gridded RCF algorithm (table 13) adapts the one-dimensional RCF for use on three-dimensional point cloud data. The algorithm is described by the following function and parameters:

$$pass = \text{gridded\_rcf}(x, y, z, width, buf, n) \tag{24}$$

pass	sequence of Boolean values indicating which values passed the filter (true = pass; false = fail)
x, *y*, *z*	sequences of point coordinate values
width	width of the vertical search window
buf	size of the horizontal buffer (grid cell)
n	minimum number of winners

Table 13. Gridded Random Consensus Filter (RCF) Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let xgrid be x/buf truncated to integers
Let ygrid be y/buf truncated to integers

Let pass be a Boolean sequence matching the length of x with all values initialized to false

For each distinct value xgi in xgrid:
    For each distinct value ygi in ygrid:
        Let idx be the indices into xgrid and ygrid where xgrid == xgi and ygrid == ygi
        If idx is null, skip to next iteration
        Let jury be the values in z corresponding to idx
        Let low be the result of rcf(jury, width)
        Let w be the indices into jury where jury >= low and jury < low + width
        Let good be the values in idx that correspond to the indices in w
        If good contains at least n values:
            Update pass so that entries corresponding to the indices in good are set to true
Return pass
```

Points are partitioned into horizontal grid cells whose size is specified by *buf*. The elevation values (*z*) in each cell are then passed through the one-dimensional RCF. The points whose elevations pass the one-dimensional RCF in each grid cell pass the filter, provided there are enough points that pass to be significant (specified by *n*). The *width* and *buf* parameters must be manually chosen based on the characteristics of the terrain. On terrain with significant sloping, the filter requires looser settings (increased *width* and/or reduced *buf*) to avoid removal of valid data, but it then allows more noise to pass through. Figure 14 illustrates how the gridded RCF filter operates over a cross-section of a point cloud.

Multi-Gridded RCF

The multi-gridded RCF algorithm (table 14) improves upon the gridded RCF algorithm by running it multiple times at slight spatial offsets. The algorithm is described by the following function and parameters:

$$pass = \text{multi\_gridded\_rcf}(x, y, z, width, buf, n, factor) \tag{25}$$

pass	sequence of Boolean values indicating which values passed the filter (true = pass; false = fail)
x, *y*, *z*	sequences of point coordinate values
width	width of the vertical search window
buf	size of the horizontal buffer (grid cell)
n	minimum number of winners
factor	number of times to shift in each horizontal direction

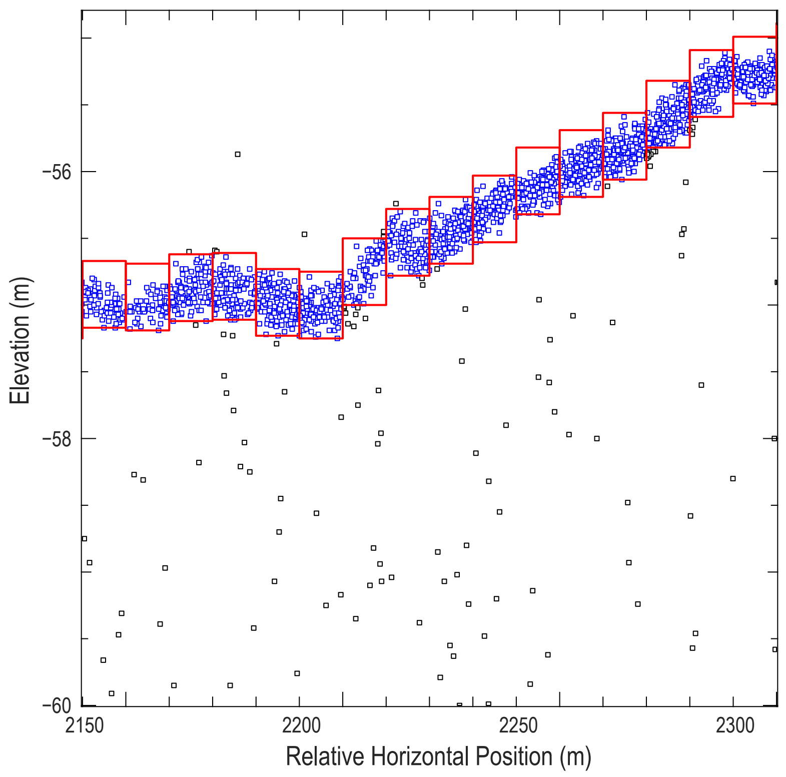


Figure 14. Example of points that pass (blue dots) and fail (black dots) the gridded Random Consensus Filter (RCF) algorithm using parameters *width*=0.5, *buf*=10, *n*=3. The figure represents a 10-meter cross-section of a point cloud (corresponding to a single row of grid cells). The red boxes represent 10-by-10-by-0.5-meter volumes containing the points that passed the filter.

Table 14. Multi-Gridded Random Consensus Filter (RCF) Algorithm

A sample Yorick implementation of this algorithm is available in Appendix A.

```
Let pass be a Boolean sequence matching the length of x with all values initialized to false
For each integer i in the range 0 to factor - 1:
    For each integer j in the range 0 to factor - 1:
        Let xshift be buf * (i / factor)
        Let yshift be buf * (j / factor)
        Let cur be the result of gridded_rcf(x+xshift, y+yshift, z, width, buf, n)
        Let good be the indices into cur where cur is true
        Update pass so that entries corresponding to the indices in good are set to true
Return pass
```

The gridded RCF algorithm works well for points near the center of the grid cell because they have good context in all horizontal directions. However, the algorithm sometimes fails near the edges of the grid cell because it lacks the nearby context on the other side of the grid line and instead compares to values on the far side of the cell. The multi-gridded RCF algorithm mitigates this deficiency by shifting grid lines so that each point has an opportunity to be closer to the grid cell’s center. The algorithm thus allows the underlying gridded-RCF parameters to be tightened to remove more noise while reducing the elimination of valid data near grid lines.

Coordinate Transformations

Geographic data can be projected into numerous coordinate systems and referenced to numerous datums. The ALPS software has basic support for three types of transformations: Universal Transverse Mercator (UTM) to/from geographic coordinates; World Geodetic System 84 (WGS84) to/from the North American Datum of 1983 (NAD83); and NAD83 ellipsoid heights to/from the North American Vertical Datum of 1988 (NAVD88).

The conversion between UTM and geographic coordinates uses math formulas presented by Snyder (1987). When working with data in WGS84, the ellipsoid parameters are a semi-major axis of 6378137 and a semi-minor axis of 6356752.3142. When working with data in NAD83, the ellipsoid parameters are a semi-major axis of 6378137 and a semi-minor axis of 6356752.3141.

The conversion between WGS84 and NAD83 uses a 7-parameter Helmert transformation as described by Land Information New Zealand (2007). The *x*,*y*,*z* rotations used are (-0.0275, -0.0101, -0.0114), the *x*,*y*,*z* translation factors used are (0.9738, -1.9453, -0.5486), and the scale factor used is 0.

Conversion between NAD83 ellipsoid heights and NAVD88 geoid heights is performed by adding elevation offsets derived from a 2-dimensional parabola-fit interpolation against geoid model files provided by the National Geodetic Survey (2015). ALPS requires the user to decide which geoid model to use and currently supports models GEOID96, GEOID99, GEOID03, GEOID06, GEOID09, GEOID12, GEOID12A, and GEOID12B.

The U.S. National Imagery and Mapping Agency notes that "an important distinction is needed between the definition of a coordinate system and the practical realization of a reference frame" (U.S. National Imagery and Mapping Agency, 2004, p. 2-2). Multiple realizations have been published for WGS84 and NAD83 because of changing data from reference stations due to tectonic movement and improved data collection methods (National Geodetic Survey, 2010). ALPS does not support working with different realizations of WGS84 or NAD83 because the magnitude of difference between realizations is below the accuracy level of EAARL. Although that assumption is held for EAARL–A, analysis by Wright and others (in press) suggests that the accuracy for EAARL–B has improved enough that realizations may need to be accounted for in the future.

Gridding

Point cloud data is gridded in ALPS by invoking IDL (Exelis Visual Information Solutions) to create a DEM saved in the Georeferencing Tagged Image File Format (GeoTIFF) file format. Most of the work is performed by standard IDL functions. The process has the following steps:

1. Use IDL's triangulate function to generate a triangulated irregular mesh based on the Delaunay triangulation method.
2. Remove triangles from the mesh whose area exceeds a specified threshold.
3. Remove triangles from the mesh whose longest side exceeds a specified threshold.
4. Use IDL's trigrid function to grid the data.
5. Use IDL's write_tiff function to save the data as a GeoTIFF.

Manual Editing

The ALPS processing workflow also incorporates significant manual editing of the lidar point cloud. The RCF algorithm does not remove all noise, and in some cases it may remove good data that must then be restored. To further improve the quality of the point cloud, analysts can supplement the RCF filter with manual editing using interactive tools built into ALPS, as documented by Bonisteel and others (2009).

These manual editing tools are also used for other tasks. For example, ALPS does not have any built-in methods for determining which waveform analysis algorithms to use for a given area. When processing an area containing a mixture of topographic and submerged features, both algorithms are used over the entire region. Analysts must manually edit the two datasets based on their delineation of the land-water interface. The datasets can then be merged to create a seamless topobathymetric product (Tyler and others, 2007). Submerged regions often have significant algorithm parameter variation depending on factors such as water depth and turbidity, even during a single flight. For such areas, analysts must use the manual editing tools to selectively reprocess sub-regions using alternate parameter configurations.

References Cited

Albrecht, Michael, Donnelly, Patrick, Bui, Peter, and Thain, Douglas, 2012, Makeflow—A portable abstraction for data intensive computing on clusters, clouds, and grids: 1st ACM SIGMOD Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET 2012), 1st, Scottsdale, Ariz., 2012, accessed April 9, 2015, at http://dx.doi.org/10.1145/2443416.2443417.

Bonisteel, J.M., Nayegandhi, Amar, Wright, C.W., Brock, J.C., and Nagle, David, 2009, Experimental Advanced Airborne Research Lidar (EAARL) data processing manual: U.S. Geological Survey Open-File Report, 2009-1078; 38 p. [Also available at http://pubs.er.usgs.gov/publication/ofr20091078.]

Comninos, Peter, 2006, Physically based lighting and shading models and rendering algorithms, *chap. in* Mathematical and computer programming techniques for computer graphics: London, Springer-Verlag, 547 p., accessed April 16, 2015, at http://dx.doi.org/10.1007/978-1-84628-292-8

Feynman, R.P., Leighton, R.B., and Sands, Matthew, 1964, Reflection and refraction, *chap. 26*, *sec. 2 in* The Feynman lectures on physics, volume 1: Reading, Mass., Addison-Wesley, p. 26-2–26-3.

Fischler, M. A., and Bolles, J. C., 1981, Random sample consensus—A paradigm for model fitting with applications to image analysis and automated cartography: Communications of the ACM, v. 24, no. 6, p. 381–395, accessed October 29, 2014, at http://dx.doi.org/10.1145/358669.358692.

Guenther, G.C., 1985, Airborne laser hydrography—System design and performance factors: National Ocean Service, National Oceanographic and Atmospheric Administration Professional Paper Series no. 1; 385 p., accessed April 1, 2015, at http://www.dtic.mil/dtic/tr/fulltext/u2/a488936.pdf.

Land Information New Zealand, 2007, Standard for New Zealand Geodetic Datum 2000: Wellington, New Zealand, Office of the Surveyor General, New Zealand Government, 18 p. [Also available at http://www.linz.govt.nz/regulatory/25000.]

Munro, D.H., and Dubois, P.F., 1995, Using the Yorick Interpreted Language: Computers in Physics, v. 9, no. 6, p. 609–615, accessed February 9, 2016, at http://dx.doi.org/10.1063/1.4823451.

National Geodetic Survey, 2010, Transformations between NAD83 and WGS84: Washington, D.C., National Atmospheric and Oceanographic Administration, 4 p., accessed September 18, 2015, at http://www.ngs.noaa.gov/CORS/Articles/WGS84NAD83.pdf.

National Geodetic Survey, 2014, OPUS—Online positioning user service: National Geodetic Survey, National Oceanographic and Atmospheric Administration Web page, accessed February 9, 2016, at http://www.ngs.noaa.gov/OPUS/.

National Geodetic Survey, 2015, The NGS geoid page: National Geodetic Survey, National Oceanographic and Atmospheric Administration Web page, accessed February 9, 2016, at http://www.ngs.noaa.gov/GEOID/.

Nayegandhi, A., Brock, J. C., and Wright, C.W., 2009, Small-footprint, waveform-resolving lidar estimation of submerged and sub-canopy topography in coastal environments: International Journal of Remote Sensing, v. 30, no. 4, p. 861–878, accessed October 24, 2014, at http://dx.doi.org/10.1080/01431160802395227.

Norstad, John, 2011, The Normal and lognormal distributions: John Norstad's home page [Engineer, Northwestern University, retired], p. 2, http://www.norstad.org/finance/normdist.pdf.

Snyder, J. P., 1987, Map Projections—A Working Manual: U.S. Geological Survey Professional Paper 1395, p. 58–64, 1 pl. [Also available at http://pubs.er.usgs.gov/publication/pp1395.]

Tyler, D., Zawada, D.G., Nayegandhi, A., Brock, J.C., Crane, M.P., Yates, K.K., and Smith, K.E.L., 2007, Topobathymetric data for Tampa Bay, Florida: U.S. Geological Survey Open-File Report 2007-1051 (revised August 2, 2012). [Also available at http://pubs.er.usgs.gov/publication/ofr20071051.]

U.S. National Imagery and Mapping Agency, 2004, WGS 84 coordinate system—Realization, sec. 2.2 *in* Department of Defense World Geodetic System 1984—Its definition and relationships with local geodetic systems (3rd ed., Amendment 1): Bethesda, Md., Geodesy and Geophysics Department, National Imagery and Mapping Agency, report no. TR8350.2, 3rd release, p. 2-2–2-7, accessed February 12, 2016, at http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf.

Wittenburg, Jens, 2008, Bryan Angles, *sect*. 2.1.2 *in* Dynamics of Multibody Systems: Berlin, Germany, Springer-Verlag, p. 12–14, accessed April 16, 2015, at http://dx.doi.org/10.1007/978-3-540-73914-2.

Wright, C.W., 2002, Airborne Lidar Processing System—ALPS [computer software] (revision c26adc74ae3f); contributions from Nayeghandi, Amar, and others: Wallops Island, Va., National Aeronautics and Space Administration. [Software information is available at http://coastal.er.usgs.gov/lsrm/tech/tech2-alps.html.]

Wright, C.W., and Brock, J.C., 2002, EAARL—A lidar for mapping shallow coral reefs and other coastal environments, *in* Proceedings, Seventh International Conference on Remote Sensing for Marine and Coastal Environments, 20–22 May 2002, Miami, Florida, USA: International Conference on Remote Sensing for Marine and Coastal Environments, 7th, Miami, Fla., 2002, Veridian International Conferences, Ann Arbor, Mich., 2002, CD-ROM.

Wright, C.W., Kranenburg, C., Battista, T.A., and Parrish, C., in press, Depth calibration and validation of the Experimental Advanced Airborne Research Lidar, EAARL-B, *in* Advances in Topobathymetric Mapping, Models, and Applications: Journal of Coastal Research, Special issue no. 76, unpaged.

Appendix A.   Source Code

This appendix provides Yorick source code implementations for many of the algorithms described in the report. This code was derived from the implementations used in ALPS (revision a1b3d73abd2a).

Some of the algorithms reference hard-coded constants. These constants are defined as follows.

```
// Speed of light in space, in meters per nanosecond
C = 0.299792458;

// Refractive index for air (oxygen, as a gas)
K_AIR = 1.000276;

// Refractive index for water (at approx. 20 degrees C)
K_H2O = 1.333;

// Speed of light through air, in m/ns
C_AIR = C/K_AIR;

// Speed of light through water, in m/ns
C_H2O = C/K_H2O;

// Factor for converting degrees to radians
DEG2RAD = pi / 180;
```

Centroid

```
func centroid(wf) {
  sum_power = wf(sum);
  if(!sum_power) return -1;

  weighted_idx = wf * indgen(numberof(wf));
  weighted_sum = weighted_idx(sum);

  return weighted_sum / double(sum_power);
}
```

Smooth Waveform

```
func smooth_waveform(wf, factor) {
  nwf = numberof(wf);
  smoothed = array(double, numberof(wf));
  // factor is points on either side; bin is total size of neighborhood
  bin = factor * 2 + 1;

  // For each point that has a big enough neighborhood, sum its neighbors
  // then divide by the binsize.
```

```
    if(bin <= nwf) {
      lo = factor + 1;
      hi = nwf - factor;
      for(i = 1; i <= bin; i++) {
        smoothed(lo:hi) += wf(i:i-bin);
      }
      smoothed /= bin;
    }

    // Smooth out the remaining leading/trailing points using the largest
    // balanced neighborhood possible.
    count = min(factor, (nwf+1)/2);
    for(i = 0; i < count; i++) {
      pts = 2 * i + 1;
      smoothed(i+1) = wf(:pts)(avg);
      smoothed(-i) = wf(1-pts:)(avg);
    }

    return smoothed;
  }
```

Leading Edge

```
  func leading_edge(wf, thresh, noiseadj) {
    nwf = numberof(wf);

    // Discrete first derivative of waveform
    wfd1 = wf(dif);

    // Find the starting points where the first derivative exceeds the
    // thresholds. These are the locations of the pulse edges.
    edges = where((wfd1 >= thresh)(dif) == 1);

    if(!numberof(edges)) return -1;

    // Determine the length of the section of the waveform that represents the
    // last return (starting from the last edge). Assume 18ns to be the longest
    // duration for a complete last return. But truncate based on length of
    // waveform.
    max_ret_len = min(18, nwf - edges(0) - 1);

    // Assume less than 5ns to be a noise pulse
    if(max_ret_len < 5) return -1;

    // Noise adjustment. Advance beyond any minor fluctuations that occur in
    // next 3 samples.
    if(noiseadj) {
      noise = where(wfd1(edges(0):edges(0)+3) < 0);
      if(numberof(noise)) {
        edges(0) = edges(0) + noise(1);
        max_ret_len = min(max_ret_len, nwf - edges(0) - 1);
      }
    }

    // Find where the bottom return pulse changes direction after its trailing
    // edge.
    rng = edges(0)+1:edges(0)+max_ret_len;
    wneg = where(wfd1(rng) < 0);

    if(!numberof(wneg)) return -1;

    return edges(0) + wneg(1);
  }
```

Detect Surface

```
func detect_surface(wf, maxint, sfc_last, wantlen) {
  nwf = numberof(wf);
  saturated = where(wf == maxint);
  numsat = numberof(saturated);

  if(numsat > 1 && saturated(1) <= sfc_last) {
    if(saturated(dif)(max) == 1) {
      surface = saturated(0);
    } else {
      surface = saturated(where(saturated(dif) > 1))(1);
    }
  } else {
    if(nwf > wantlen + 8) {
      surface = wf(1:wantlen)(mxx);
    } else {
      surface = min(wantlen, nwf);
    }
  }

  return surface;
}
```

Compensate for Exponential Decay

```
func compensate_decay_exp(wf, maxint, laser, water, surface, agc_factor) {
  nwf = numberof(wf);
  attdepth = indgen(0:nwf-1) * C_H2O * .5;

  laser_decay = exp(laser * attdepth) * maxint;
  secondary_decay = exp(water * attdepth) * maxint;

  shift = 0 - surface + 1;
  decay = laser_decay + secondary_decay * .25;
  agc = 1 - exp(agc_factor * attdepth);

  decay(surface:) = decay(:shift);
  decay(:surface+1) = maxint;

  agc(surface:) = agc(:shift);
  agc(:surface) = 0;

  bias = (1 - agc) * -5;
  wf_decay = (wf - decay) * agc + bias;

  return wf_decay;
}
```

Compensate for Log-Normal Decay

```
func compensate_decay_lognorm(wf, mean, stdev, xshift, xscale, tiepoint,
agc_factor, surface) {
  nwf = numberof(wf);
  if(nwf < tiepoint) {
    return wf * 0;
  }

  x = (indgen(1:nwf) - xshift) / double(xscale);
  decay = log_normal(x, mean, stdev);
  decay *= (wf(tiepoint) / decay(tiepoint));

  attdepth = indgen(0:nwf-1) * C_H2O * .5;
  agc = 1 - exp(agc_factor * attdepth);
  agc(surface:) = agc(:0-surface+1);
  agc(:surface) = 0;

  bias = (1 - agc) * -5;
  wf_decay = (wf - decay) * agc + bias;

  return wf_decay;
}
```

Log Normal

```
func log_normal(x, mean, stdev) {
  variance = stdev^2;

  // The exp function will not work on values <= 0, so force them to 0 and
  // only calculate where values are positive.
  y = array(0., dimsof(x));
  w = where(x > 0);
  if(numberof(w)) {
    y(w) = exp(-(log(x(w))-mean)^2/(2*variance)) / (x(w)*sqrt(2*pi*variance));
  }

  return y;
}
```

Detect Bottom

```
func detect_bottom(wf, first, last, thresh) {
  offset = first - 1;
  last = min(last, numberof(wf));
  last = offset + remove_noisy_tail(wf(first:last), thresh);

  // waveform too short after removing noisy tail
  if(last - first < 4) return -1;

  peaks = extract_peaks_first_deriv(wf(first:last), thresh);
  // no significant inflection in backscattered waveform after decay
  if(!numberof(peaks)) return -1;

  bottom = peaks(0) + offset;
  return bottom;
}
```

Remove Noisy Tail

```
func remove_noisy_tail(wf, thresh) {
  w = where(wf > wf(min) + thresh);
  if(!numberof(w)) return numberof(wf);
  return min(numberof(wf), w(0)+1);
}
```

Extract Peaks with First Derivative

```
func extract_peaks_first_deriv(wf, thresh) {
  wfd1 = wf(dif);

  // By subtracting a small amount from the first derivative, we force the
  // algorithm to ignore very tiny positive slopes. This often occurs where
  // two successive samples have the same value in the raw waveform, but AGC
  // boosts the second ever-so-slightly more than the first.
  wfd1 -= 0.05;

  wfd1_sign = sign(wfd1);
  wfd1_sign_d1 = wfd1_sign(dif);

  peaks = where(wfd1_sign_d1 == -2);
  if(!numberof(peaks)) return [];

  // Fix indexes caused by successive dif operations
  peaks += 1;

  idx = where(wf(peaks) >= thresh);
  if(!numberof(idx)) return [];
  peaks = peaks(idx);

  return peaks;
}
```

Saturation Check

```
func saturation_check(saturated, bottom) {
  is_sat = saturated(bottom);

  // occasionally, AGC pushes the peak off the saturated section by a sample
  if(!is_sat && saturated(bottom-1)) {
    is_sat = 1;
    bottom--;
  }

  sat0 = sat1 = bottom;
  while(sat0 > 1 && saturated(sat0-1)) sat0--;
  while(sat1 < numberof(saturated) && saturated(sat1+1)) sat1++;
  return long(0.5*(sat0+sat1));
}
```

Validate Bottom

```
func bottom_validate(wf, bottom, thresh, first, last, lw_dist, rw_dist,
lw_factor, rw_factor) {
  lw_idx = bottom - lw_dist;
  rw_idx = bottom + rw_dist;

  if(wf(bottom) <= thresh || last < rw_idx) {
    // Below threshold
    return 0;
  }

  if(lw_idx < first || rw_idx > last) {
    // Too close to edge gate
    return 0;
  }

  lw_thresh = lw_factor * wf(bottom);
  rw_thresh = rw_factor * wf(bottom);

  if(wf(lw_idx) > lw_thresh || wf(rw_idx) > rw_thresh) {
    // Bad pulse shape
    return 0;
  }

  return 1;
}
```

Point Projection

```
func project_point(arZ, arX, arY, maZ, maX, maY, laZ, laX, laY, d, g, mag,
&p) {
  // Parameters:
  // arZ, arX, arY - aircraft rotations vs real world
  // maZ, maX, maY - mirror rotations vs aircraft
  // laZ, laX, laY - laser rotations vs aircraft
  // d - displacement vector from GPS to mirror
  // g - GPS coordinates for GPS unit
  // mag - magnitude of the laser vector from mirror to target (distance)
  // p - output coordinate for target point coordinates

  Rar = rotation_matrix(arZ, arX, arY);
  Rla = rotation_matrix(laZ, laX, laY);
  Rma = rotation_matrix(maZ, maX, maY);
  m = calc_mirror(Rar, d, g);

  I = calc_incidence(Rar, Rla);
  N = calc_normal(Rar, Rma);
  S = calc_reflection(I, N);

  p = calc_target(S, mag, m);
}
```

Rotation Matrix

```
func rotation_matrix(Z, X, Y) {
  z = Z * DEG2RAD;
  x = X * DEG2RAD;
  y = Y * DEG2RAD;

  Rx = Ry = Rz = array(double, 3, 3);

  Rx(1,) = [1, 0, 0];
  Rx(2,) = [0, cos(x), -sin(x)];
  Rx(3,) = [0, sin(x), cos(x)];

  Ry(1,) = [cos(y), 0, sin(y)];
  Ry(2,) = [0, 1, 0];
  Ry(3,) = [-sin(y), 0, cos(y)];

  Rz(1,) = [cos(z), -sin(z), 0];
  Rz(2,) = [sin(z), cos(z), 0];
  Rz(3,) = [0, 0, 1];

  return Rz(,+) * (Rx(,+) * Ry(+,))(+,);
}
```

Calculate Mirror Location

```
func calc_mirror(Rar, d, g) {
  return Rar(,+) * d(+,) + g;
}
```

Calculate Incidence Vector

```
func calc_incidence(Rar, Rla) {
  return Rar(,+) * (Rla(,+) * [0,-1,0](+))(+,);
}
```

Calculate Normal Vector

```
func calc_normal(Rar, Rma) {
  return Rar(,+) * (Rma(,+) * [0,0,1](+))(+,);
}
```

Calculate Reflection Vector

```
func calc_reflection(I, N) {
  return 2 * (I*N)(sum) * N - I;
}
```

Calculate Target Location

```
func calc_target(S, d, m) {
  return S * d + m;
}
```

Water Corrections

```
func water_correction(sx, sy, sz, bx, by, bz, &cx, &cy, &cz) {
  // Calculate uncorrected deltas
  dx = bx - sx;
  dy = by - sy;
  dz = bz - sz;

  // Calculate total displacement
  dist = sqrt(dx^2 + dy^2 + dz^2);

  // Calculate horizontal displacement
  horiz = sqrt(dx^2 + dy^2);

  // Calculate vertical displacement and note which direction it is in
  height = dz;
  hsign = sign(dz);

  // Calculate angle of x and y components of horizontal displacement
  theta = atan(dy, dx);

  // To avoid divide-by-zero issues, replace zero distances with very small
  // distances
  w = where(!dist);
  if(numberof(w)) dist(w) = 1e-10;

  // Calculate angle of incidence for laser intercepting water surface
  phi_air = acos(height/dist);
```

```
    // Use Snell's law to determine angle in water
    phi_water = asin(sin(phi_air)/K_H2O);

    // Adjust distance for the change in speed of light
    dist *= (C_H2O / C_AIR);

    // Calculate adjusted height and horizontal displacement
    height = cos(phi_water) * dist;
    horiz = sin(phi_water) * dist;

    // Calculate corrected deltas
    cdx = horiz * cos(theta);
    cdy = horiz * sin(theta);
    cdz = height * hsign;

    // Add corrected deltas to surface to derive corrected bottom
    cx = sx + cdx;
    cy = sy + cdy;
    cz = sz + cdz;
  }
```

One-Dimensional RCF

```
  func rcf(jury, width) {
    jury = jury(sort(jury));
    jurysize = numberof(jury);

    bestvote = bestidx = 0;
    for(i = 1, j = 1; j <= jurysize; i++) {
      upper = jury(i) + width;
      while(j <= jurysize && jury(j) < upper) j++;
      vote = j - i;
      if(vote >= bestvote) {
        bestvote = vote;
        bestidx = i;
      }
    }

    return jury(bestidx);
  }
```

Gridded RCF

```
func gridded_rcf(x, y, z, width, buf, n) {
  xgrid = long(x/buf);
  ygrid = long(y/buf);

  xgrid_min = xgrid(min);
  xgrid_max = xgrid(max);
  ygrid_min = ygrid(min);
  ygrid_max = ygrid(max);

  pass = array(0, dimsof(x));

  for(xgi = xgrid_min; xgi <= xgrid_max; xgi++) {
    for(ygi = ygrid_min; ygi <= ygrid_max; ygi++) {
      idx = where(xgrid == xgi & ygrid == ygi);
      if(!numberof(idx)) continue;

      low = rcf(z(idx), width);

      w = where(z(idx) >= low & z(idx) < low + width);
      if(numberof(w) < n) continue

      pass(idx(w)) = 1;
    }
  }

  return pass;
}
```

Multi-Gridded RCF

```
func multi_gridded_rcf(x, y, z, width, buf, n, factor) {
  pass = array(0, dimsof(x));

  for(i = 0; i < factor; i++) {
    for(j = 0; j < factor; j++) {
      xshift = buf * (i / double(factor));
      yshift = buf * (j / double(factor));
      cur = gridded_rcf(x+xshift, y+yshift, z, width, buf, n);
      pass = pass | cur;
    }
  }

  return pass;
}
```

Nagle and Wright—Algorithms used in the Airborne Lidar Processing System (ALPS)—Open-File Report 2016–1046

ISSN 2331-1258 (online)
http://dx.doi.org/10.3133/ofr20161046