laspy Documentation

Release 1.2.5

Grant Brown, Howard Butler

March 24, 2014

Contents

Note: The laspy documentation is currently having trouble with readthedocs, and the latest online version is therefore incomplete. Documentation for the release version 1.2.5 is available at python hosted, and the latest documentation can be built from the source at github.

Laspy is a python library for reading, modifying, and creating .LAS LIDAR files. It is available at https://github.com/grantbrown/laspy Laspy was designed for python 2.7.

Contents:

CHAPTER 1

Laspy: Tutorial

1.1 Background: What are LAS Files?

LIDAR Data

LIDAR data is analogous to RADAR with LASERs, and is short for Light Detection and Ranging. This library provides a python API to read, write, and manipulate one popular format for storing LIDAR data, the .LAS file.

LAS files are binary files packed according to several specifications.

LAS Specifications

laspy 1.0 supports LAS formats 1.0 to 1.2, and provides preliminary support for formats 1.3 and 1.4.

1.0	http://www.asprs.org/a/society/committees/standards/asprs_las_format_v10.pdf
1.1	http://www.asprs.org/a/society/committees/standards/asprs_las_format_v11.pdf
1.2	http://www.asprs.org/a/society/committees/standards/asprs_las_format_v12.pdf
1.3	http://www.asprs.org/a/society/committees/standards/LAS_1_3_r11.pdf
1.4	http://www.asprs.org/a/society/committees/standards/LAS_1_4_r11.pdf

LAS Headers

Much of the data required for laspy to know how to read the LAS file is present in the header, which laspy interacts with according to the pattern below. There are some minor departures from the raw specification for convenience, namely combining the Day+Year fields into a python `datetime.datetime` object, and placing the X Y and Z scale and offset values together.

Note: The various LAS specifications say that the Max and Min X Y Z fields store unscaled values, however LAS data in the wild appears not to follow this convention. Therefore, by default, laspy stores the scaled double precision values and updates header files accordingly on file close. This can be overridden by supplying one of several optional arguments to file.close(). First, you can simply not update the header at all, by specifying ignore_header_changes=True. Second, you can ask that laspy store the unscaled values explicitly, by specifying minmax_mode="unscaled".

If this sounds like gibberish, feel free to ignore it!

Header: Version 1.0 - 1.2

Field Name	Laspy Abbreviation	File Format[number] (length)
File Signature	file_signature	char[4] (4)
File Source Id	file_source_id	unsigned short[1] (2)
(Reserved or Global Encoding)	global_encoding	unsigned short[1] (2)
Gps Time Type	gps_time_type	Part of Global Encoding
Project Id (4 combined fields)	guid	ulong+ushort+ushort+char[8] (16)
Verion Major	version_major	unsigned char[1] (1)
Version Minor	version_minor	unsigned char[1] (1)
Version Major + Minor	version	(see above)
System Identifier	system_id	char[32] (32)
Generating Software	software_id	char[32] (32)
File Creation Day of Year	(handled internally)	unsigned short[1] (2)
File Creation Year	(handled internally)	unsigned short[1] (2)
Creation Day + Year	date	(see above)
Header Size	header_size	unsigned short[1] (2)
Offset to Point Data	data_offset	unsigned long[1] 4
Number of Variable Length Recs	(handled internally)	unsigned long[1] 4
Point Data Format Id	data_format_id	unsigned char[1] (1)
Data Record Length	data_record_length	unsigned short[1] (2)
Number of point records	records_count	unsigned long[1] (4)
Number of Points by Return Ct.	point_return_count	unsigned long[5 or 7] (20 or 28)
Scale Factor (X, Y, Z)	scale	double[3] (24)
Offset (X, Y, Z)	offset	double[3] (24)
Max (X, Y, Z)	max	double[3] (24)
Min (X, Y, Z)	min	double[3] (24)

Header: Version 1.3 (appended)

Start of waveform data record	start_waveform_data_rec	unsigned long long[1] (8)

Header: Version 1.4 (appended)

Start of first EVLR	start_first_evlr	unsigned long long[1] (8)
Number of EVLRs	(handled internally)	unsigned long[1] (4)
Number of point records	point_records_count	unsigned long long[1] (8)
Number of points by return ct.	point_return_count	unsigned long long[15] (120)

In addition, the LAS 1.4 specification replaces the point_records_count and point_return_count fields present in previous versions with legacy_point_records_count and legacy_point_return_count, which match the specification of their original counterparts.

Therefore broadly speaking, with the exception of the 1.4 legacy fields and format changes, these specifications are cumulative - each adds more potential configurations to the last, while (mostly) avoiding backwards incompatability.

Point Formats

LAS Format	Point Formats Supported
Version 1.0	0, 1
Version 1.1	0, 1
Version 1.2	0, 1, 2, 3
Version 1.3	0, 1, 2, 3, 4, 5
Version 1.4	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Note: Where there exist discrepencies between the use of point fields between LAS versions, we will assume that the more recent version is used. For example, the original 1.0 specification used a point field called *"File Marker"*, which was generally neglected. We will therefore use the more recent *"User Data"* nomenclature.

Sub-Byte Fields: Point Formats 0-5

There are two bytes per point record dedicated to sub-byte length fields, one called by laspy *flag_byte* and the other called *raw_classification*. These are detailed below, and are available from Laspy in the same way as full size dimensions:

Flag Byte

Field Name	Laspy Abbreviation	Length(in bits)
Return Number	return_num	3
Number of Returns	num_returns	3
Scan Direction Flag	scan_dir_flag	1
Edge of Flight Line	edge_flight_line	1

Classification Byte

Field Name	Laspy Abbreviation	Length(in bits)
Classification	classification	4
Synthetic	synthetic	1
Key Point	key_point	1
Withheld	withheld	1

Sub-Byte Fields: Point Formats 6-10

The new point formats introduced by LAS specification 1.4 shuffle the bit fields around a bit.

Flag Byte

Field Name	Laspy Abbreviation	Length(in bits)
Return Number	return_num	4
Number of Returns	num_returns	4

Classification Flags

Field Name	Laspy Abbreviation	Length(in bits)
synthetic	synthetic	1
key point	key point	1
withheld	withheld	1
overlap	overlap	1
Scanner Channel	scanner_channel	2
Scan Direction Flag	scan_dir_flag	1
Edge of Flight Line	edge_flight_line	1

Classification Byte

LAS 1.4 introduces a byte sized classification field, and this is interpreted as an integer. For information on the interpretation of the Classification Byte field, see the LAS specification. This dimension is accessable in laspy as simply `laspy.file.File`.classification for files which make the field available. In files without the full byte classification, this property provides the 4 bit classification field which becomes "classification_flags" in 1.4.

Point Format Specifications The five possible point formats are detailed below:

Point Format 0

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Byte)	raw_classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Point Source Id	pt_src_id	unsigned short[1] (2)

Point Format 1

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Byte)	raw_classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)

Point Format 2

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Byte)	raw_classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Point Source Id	pt_src_id	unsigned short[1] (2)
Red	red	unsigned short[1] (2)
Green	green	unsigned short[1] (2)
Blue	blue	unsigned short[1] (2)

Point Format 3

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Byte)	raw_classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)
Red	red	unsigned short[1] (2)
Green	green	unsigned short[1] (2)
Blue	blue	unsigned short[1] (2)

Point Format 4

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Byte)	raw_classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)
Wave Packet Descriptor Index	wavefm_packet_desc_index	unsigned char[1] (1)
Byte Offset to Waveform Data	byte_offset_to_waveform_data	unsigned long long[1] (8)
Waveform Packet Size	waveform_packet_size	unsigned long[1] (4)
Return Point Waveform Location	return_pt_waveform_loc	float[1] (4)
X(t)	x_t	float[1] (4)
Y(t)	y_t	float[1] (4)
Z(t)	z_t	float[1] (4)

Point Format 5

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Byte)	raw_classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)
Red	red	unsigned short[1] (2)
Green	green	unsigned short[1] (2)
Blue	blue	unsigned short[1] (2)
Wave Packet Descriptor Index	wavefm_packet_desc_index	unsigned char[1] (1)
Byte Offset to Waveform Data	byte_offset_to_waveform_data	unsigned long long[1] (8)
Waveform Packet Size	waveform_packet_size	unsigned long[1] (4)
Return Point Waveform Location	return_pt_waveform_loc	float[1] (4)
X(t)	x_t	float[1] (4)
Y(t)	y_t	float[1] (4)
Z(t)	z_t	float[1] (4)

Point Format 6

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Flags)	classification_flags	unsigned byte[1] (1)
classification	classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Scan Angle	scan_angle	short[1] (2)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)

Point Format 7

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Flags)	classification_flags	unsigned byte[1] (1)
classification	classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Scan Angle	scan_angle	short[1] (2)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)
Red	red	unsigned short[1] (2)
Green	green	unsigned short[1] (2)
Blue	blue	unsigned short[1] (2)

Point Format 8

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Flags)	classification_flags	unsigned byte[1] (1)
classification	classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Scan Angle	scan_angle	short[1] (2)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)
Red	red	unsigned short[1] (2)
Green	green	unsigned short[1] (2)
Blue	blue	unsigned short[1] (2)
Near Infared	nir	unsigned short[1] (2)

Point Format 9

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Flags)	classification_flags	unsigned byte[1] (1)
classification	classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Scan Angle	scan_angle	short[1] (2)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)
Wave Packet Descriptor Index	wavefm_packet_desc_index	unsigned char[1] (1)
Byte Offset to Waveform Data	byte_offset_to_waveform_data	unsigned long long[1] (8)
Waveform Packet Size	waveform_packet_size	unsigned long[1] (4)
Return Point Waveform Location	return_pt_waveform_loc	float[1] (4)
X(t)	x_t	float[1] (4)
Y(t)	y_t	float[1] (4)
Z(t)	z_t	float[1] (4)

Point Format 10

Field Name	Laspy Abbreviation	File Format[number] (length)
X	X (x for scaled)	long[1] (4)
Y	Y (y for scaled)	long[1] (4)
Z	Z (z for scaled)	long[1] (4)
Intensity	intensity	unsigned short[1] (2)
(Flag Byte)	flag_byte	unsigned byte[1] (1)
(Classification Flags)	classification_flags	unsigned byte[1] (1)
classification	classification	unsigned byte[1] (1)
User Data	user_data	unsigned char[1] (1)
Scan Angle	scan_angle	short[1] (2)
Point Source Id	pt_src_id	unsigned short[1] (2)
GPS Time	gps_time	double[1] (8)
Red	red	unsigned short[1] (2)
Green	green	unsigned short[1] (2)
Blue	blue	unsigned short[1] (2)
Near Infared	nir	unsigned short[1] (2)
Wave Packet Descriptor Index	wavefm_packet_desc_index	unsigned char[1] (1)
Byte Offset to Waveform Data	byte_offset_to_waveform_data	unsigned long long[1] (8)
Waveform Packet Size	waveform_packet_size	unsigned long[1] (4)
Return Point Waveform Location	return_pt_waveform_loc	float[1] (4)
X(t)	x_t	float[1] (4)
Y(t)	y_t	float[1] (4)
Z(t)	z_t	float[1] (4)

Variable Length Records, Extended Variable Length Records

Each LAS file can also contain a number of variable length records, or VLRs. These can be used to store specific georeferencing information, or user/software specific data. When laspy recognizes a specific type of VLR, it attempts to parse the VLR_body data, and provides a simple API to interact with these fields.

The LAS 1.3 specification also adds the concept of an extended VLR. In 1.3, waveform data is stored at the end of the file in a variable length record which can contain more data than the original VLR, due to the larger data type for the "rec_len_after_header" field. This EVLR is known as the Waveform Data Packet Record.

In the LAS 1.4 specification, more than one EVLR may be present. The headers to these newer formats therefore provide the byte offset to the Waveform Data Packet Record, as well as the byte offset to the first EVLR record. These numbers may be the same or different.

To summarize in tabular form, LAS files follow the following structure:

`laspy.header.VLR` Attributes:

Name	Format in File	Length
reserved	Unsigned Short	2
user_id	Character	16
record_id	Unsigned Short	2
rec_len_after_header	Unsigned Short	2
description	Character	32
VLR_body	Raw Bytes	rec_len_after_header

Additionally, when laspy is able to parse a VLR_body, it provides an attribute called parsed_body, which gives a numpy array of the VLR_body members. Also, several methods comprise the VLR_body api: parse_data, pack_data, and body_summary. The first unpacks data to parsed_body from VLR_body, while the second packs data from parsed_body to VLR_body. The last prints a simple summary of the parsed VLR body.

`laspy.header.EVLR` Attributes:

Name	Format in File	Length
reserved	Unsigned Short	2
user_id	Character	16
record_id	Unsigned Short	2
rec_len_after_header	Unsigned LongLong	8
description	Character	32
VLR_body	Raw Bytes	rec_len_after_header

#	Formats 1.0 through 1.2
1	Header
2	VLR(s)
3	Point Records

#	Format 1.3
1	Header
2	VLR(s)
3	Point Records
4	Waveform Data Packet Record

#	Format 1.4
1	Header
2	VLR(s)
3	Point Records
4	EVLR(s), including Waveform Data if present

1.2 Laspy Tools

Laspy comes with several command line utilities which use the library. When laspy is installed with setup.py, these scripts are built and installed by setuptools, and shold become available to the command line envionment you're using. The tools include lascopy, lasexplorer, lasverify, lasvalidate, and finally lasviewer. The first four are full command line utilities and should function out of the box after a successful laspy install. Lasviewer is an OpenGL point cloud viewer using laspy to provide LAS data, and requires OpenGL 3.0+, PyOpenGL, and GLUT.

lascopy

overview

Lascopy is a general purpose LAS file version and point format conversion tool, and is able to read and write between all valid combinations of these two values. If the output data format has fewer dimensions than the input data set, the output file will not include them.

usage For help, simply type:

```
lascopy -h
```

In general, lascopy is called as:

```
lascopy ./path/to/input/file ./path/to/output/file <output point format> <output file format>
```

lascopy also accepts the optional logical arguments, -b and -u.

Specifying -b=True will cause lascopy to attempt to copy sub-byte field data to the output file in the event that there is a discrepency in how this data is stored between the two point formats (i.e., if you need to convert between point formats 5 and below and those greater than 5).

Specifying -u=True indicates that lascopy should update the point return count histogram before closing the output file. This is usually not neccesary, but when downscaling from point formats greater than 5 to point formats below 5, there is excess point return data which can not fit into the header.

Both of these options are associated with substantial overhead, and as a result they are set to False by default.

example

Suppose you want to convert ./file_1.LAS to point format 8 from 0, and to file version 1.4 from 1.0. Further, suppose that you want to make sure sub-byte fields are populated and that the histogram is updated. You would call lascopy as:

```
lascopy ./file_1.LAS ./file_2.LAS 8 1.4 -u=True -b=True
```

Note: Even if -b=True is specified, lascopy may not be able to store all sub-byte data in the output format. This can occur when the input data has a larger number of bits per field. For example, las point formats 6-10 reserve four bits for return number, while formats 0-5 reserve only three. In cases such as this, a larger value may be stored in the input file than can be written to the output file. If this occurs, a warning is printed and the output file receives zeros for the offending fields.

lasexplorer

Overview

Lasexplorer is the simplest utility described here. It provides a basic entry point to start an interactive shell to try out the laspy API. It accepts one mandatory argument, the path to the file to be read. When run, the script reads in the requested file, and calls the resulting laspy file object inFile. Unless suppressed, it also prints a brief summary of the LAS file supplied.

Usage

The basic use case is simply:

```
lasexplorer ./path/to/las/file
```

The shell defaults to read mode, which will prevent you from accidentally breaking any data files. If you want the ability to break stuff, however, you're free to specify the optional mode argument, and set it to read/write:

```
lasexplorer ./path/to/las/file --mode=rw
```

The shell doesn't provide the ability to open write mode files from the command line, because this action requires a valid header object. If you'd like to experiment with write mode, however, you can easily instantiate pure write mode files once the shell is active:

```
new_write_mode_file = File("Path_to_file.las", mode = "w",
                           header = inFile.header)
```

This is fine for learning how to use the API, but any substantial work is better done with a dedicated script (see tutorial for details on scripting with laspy).

If you'd like to supress the printed summary, simply specify -q=True:

lasvalidate

overview

Lasvalidate is a simple validation tool for las files. Currently, it runs three tests though this may be expanded. First, it checks if all points fall inside the bounding box specified by file.header.max and file.header.min. Second, it checks that the bounding box is precise, that is, that the max and min values specified by the header are equal to the max and min values prensent in the point data within a given tolerance. Finally, it checks that the X and Y range data makes sense. Lasvalidate produces a log file to indicate problems.

usage

Lasvalidate is called as:

```
lasvalidate ./path/to/las/file
```

Optionally, the user can specify –log=/path/to/logfile and –tol=tolerance, where –log specifies where the log will be written, and –tol determines the tolerance for comparisons of actual and header max/min data. By default, the logfile is written to ./lasvalidate.log, and the tolerance is set to 0.01

lasverify

overview

Lasverify is a LAS file diff tool; that is, it compares two LAS files based on header fields, point dimensons, and VLR/EVLR records. Header discrepencies are displayed, while differences in VLRs, EVLRs and point dimensons are simply indicated.

usage

In general, lasverify is called as:

```
lasverify ./path/to/file/1 ./path/to/file/2
```

> There is one additional argument,-b, which is similar in function to its counterpart in lascopy. Specifying -b=True will cause lasverify to dig into the sub-byte fields of both files to compare them individually in the case of a format mismatch, which occurrs when comparing files of point format less than six with those greater than five. Specifying -b=True when no such mismatch exists has no effect.

lasviewer

overview

Lasviewer is an OpenGL point cloud visualizer for laspy. Upon successful OpenGL initialization, the user is shown a resizable OpenGL window, which should depict the point cloud associated with the input file from a birds-eye view. The user can then navigate around the point cloud using their keyboard.

usage

Lasviewer is simple to call:

```
lasviewer ./path/to/las/file
```

By default, lasviewer will first attempt to display the point cloud in RGB color, though if color informaton is not present in the file, greyscale is used. In this case, the image is shaded according to the intensity dimension. One can also specify the mode explicitly:

Default Color Modes

```
# Display the intensity shaded map
lasviewer ./path/to/las/file --mode=intensity
# Display a heatmap based on the z dimension.
lasviewer ./path/to/las/file --mode=elevation
# Display the rgb data (if present in the file)
lasviewer ./path/to/las/file --mode=rgb
```

The elevation mode creates a three color heatmap (blue-green-red) for the Z dimension, and colors the point cloud accordingly. The RGB mode uses color data present in the LAS file to provide a true-color representation of the point cloud. If either of these modes fails for whatever reason, lasviewer will attempt to fall back to the intensity mode.

Custom Color Modes

You can use heatmap and greyscale color modes to display any numeric dimension offered by a las file, and the syntax is no more complicated. For example, lets say we're interested in gps_time in order to see which parts of a a LAS file were recorded first:

```
lasviewer ./path/to/las/file --mode=heatmap --dim=gps_time

lasviewer ./path/to/las/file --mode=greyscale --dim = gps_time
```

A Cool Trick

With laspy, you don't acutally need to use the lasviewer tool to visualize LAS files. In fact, the lasviewer tool is really just a wrapper for the File.visualize method, which accepts mode and dimension arguments. Thus, for example, if you wanted to visualize a file with an exaggerated Z scale, you could use the lasexplorer tool, rescale z, and then call .visualize()

First open a file in read/write mode with lasexplorer:

```
lasexplorer ./path/to/las/file --mode=rw
```

Now you can re-scale Z, and call visualize:

```
inFile.z *= 2.5
inFile.visualize(mode="elevation")
```

Navigation/Controls

There are currently no menus or help options once the viewer is running, so the following controls will prove useful:

Key	Function
w	Look up
s	Look down
a	Look left
d	Look right
shift-w	Move forwards
shift-s	Move backwards
shift-a	Move left
shift-d	Move right
q	Roll counterclockwise
e	Roll clockwise
+	Increase movement/look granularity
-	Decrease movement/look granularity
x	Snap to x axis
y	Snap to y azis
z	Snap to z axis
r	Reset the view location and angle

There is currently no mouse support, though the bindings are in place for future development.

1.3 Getting Started

Dependencies

Apart from the python standard library, we require Numpy, available at http://numpy.scipy.org.

Installation

If you feel comfortable manually installing python libraries, feel free to do so - the module is readily importable from ./laspy/, so adding this directory to your sys.path should suffice. Otherwise, the easiest way to get set up is to use setuptools or distribute.

Distribute is available at: http://pypi.python.org/pypi/distribute.

Once you have that installed, navigate to the root laspy directory where setup.py is located, and do:

```
$ python setup.py build
$ python setup.py install
```

If you encounter permissions errors at this point (and if you're using a unix environment) you may need to run the above commands as root, e.g.

```
$ sudo python setup.py build
```

Once you successfully build and install the library, run the test suite to make sure everything is working:

```
$ python setup.py test
```

Importing laspy

Previously, laspy documentation exclusively used relative imports. However, as outlined in PEP 328, there is a growing consensus in the Python community that absolute imports are preferable to relative imports. Moving forward, this convention will be adopted by laspy. If you wish to use absolute imports, you can:

```
import laspy
infile = laspy.file.File("./laspytest/data/simple.las", mode="r")
# ...and so on
```

Opening .LAS Files

The first step for getting started with laspy is to open a `laspy.file.File` object in read mode. As the file *"simple.las"* is included in the repository, the tutorial will refer to this data set. We will also assume that you're running python from the root laspy directory; if you run from somewhere else you'll need to change the path to simple.las.

The following short script does just this:

```
import numpy as np
import laspy
inFile = laspy.file.File("./laspytest/data/simple.las", mode = "r")
```

When a file is opened in read mode, laspy first reads the header, processes any VLR and EVLR records, and then maps the point records with numpy. If no errors are produced when calling the File constructor, you're ready to read data!

Reading Data

Now you're ready to read data from the file. This can be header information, point data, or the contents of various VLR records. In general, point dimensions are accessible as properties of the main file object, and header attributes are accessible via the header property of the main file object. Refer to the background section of the tutorial for a reference of laspy dimension and field names.

```
# Grab all of the points from the file.
point_records = inFile.points

# Grab just the X dimension from the file, and scale it.

def scaled_x_dimension(las_file):
    x_dimension = las_file.X
    scale = las_file.header.scale[0]
    offset = las_file.header.offset[0]
    return(x_dimension*scale + offset)

scaled_x = scaled_x_dimension(inFile)
```

Note: Laspy can actually scale the x, y, and z dimensions for you. Upper case dimensions (*las_file.X,*

> *las_file.Y, las_file.Z*) give the raw integer dimensions, while lower case dimensions (*las_file.x, las_file.y, las_file.z*) give the scaled value. Both methods support assignment as well, although due to rounding error assignment using the scaled dimensions is not reccomended.

Again, the `laspy.file.File` object *inFile* has a reference to the `laspy.header.Header` object, which handles the getting and setting of information stored in the laspy header record of *simple.las*. Notice also that the *scale* and *offset* values returned are actually lists of [*x scale, y scale, z scale*] and [*x offset, y offset, z offset*] respectively.

LAS files differ in what data is available, and you may want to check out what the contents of your file are. Laspy includes several methods to document the file specification, based on the `laspy.util.Format` objects which are used to parse the file.

```
# Find out what the point format looks like.
pointformat = inFile.point_format
for spec in inFile.point_format:
    print(spec.name)

#Like XML or etree objects instead?
a_mess_of_xml = pointformat.xml()
an_etree_object = pointformat.etree()

#It looks like we have color data in this file, so we can grab:
blue = inFile.blue

#Lets take a look at the header also.
headerformat = inFile.header.header_format
for spec in headerformat:
    print(spec.name)
```

Many tasks require finding a subset of a larger data set. Luckily, numpy makes this very easy. For example, suppose we're interested in finding out whether a file has accurate min and max values for the X, Y, and Z dimensions.

```
import laspy
import numpy as np

inFile = laspy.file.File("/path/to/lasfile", mode = "r")
# Some notes on the code below:
# 1. inFile.header.max returns a list: [max x, max y, max z]
# 2. np.logical_or is a numpy method which performs an element-wise "or"
#    comparison on the arrays given to it. In this case, we're interested
#    in points where a XYZ value is less than the minimum, or greater than
#    the maximum.
# 3. np.where is another numpy method which returns an array containing
#    the indexes of the "True" elements of an input array.

# Get arrays which indicate invalid X, Y, or Z values.
X_invalid = np.logical_or((inFile.header.min[0] > inFile.x),
                          (inFile.header.max[0] < inFile.x))
Y_invalid = np.logical_or((inFile.header.min[1] > inFile.y),
                          (inFile.header.max[1] < inFile.y))
Z_invalid = np.logical_or((inFile.header.min[2] > inFile.z),
                          (inFile.header.max[2] < inFile.z))
bad_indices = np.where(np.logical_or(X_invalid, Y_invalid, Z_invalid))

print(bad_indices)
```

Now lets do something a bit more complicated. Say we're interested in grabbing only the points from a file which are within a certain distance of the first point.

```
# Grab the scaled x, y, and z dimensions and stick them together
# in an nx3 numpy array

coords = np.vstack((inFile.x, inFile.y, inFile.z)).transpose()

# Pull off the first point
first_point = coords[0,:]

# Calculate the euclidean distance from all points to the first point

distances = np.sum((coords - first_point)**2, axis = 1)

# Create an array of indicators for whether or not a point is less than
# 500000 units away from the first point

keep_points = distances < 500000

# Grab an array of all points which meet this threshold

points_kept = inFile.points[keep_points]

print("We're keeping %i points out of %i total"%(len(points_kept), len(inFile)))
```

As you can see, having the data in numpy arrays is very convenient. Even better, it allows one to dump the data directly into any package with numpy/python bindings. For example, if you're interested in calculating the nearest neighbors of a set of points, you might want to use a highly optimized package like FLANN (http://people.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN)

Here's an example doing just this:

```
import laspy
import pyflann as pf
import numpy as np

# Open a file in read mode:
inFile = laspy.file.File("./laspytest/data/simple.las")
# Grab a numpy dataset of our clustering dimensions:
dataset = np.vstack([inFile.X, inFile.Y, inFile.Z]).transpose()

# Find the nearest 5 neighbors of point 100.

neighbors = flann.nn(dataset, dataset[100,], num_neighbors = 5)
print("Five nearest neighbors of point 100: ")
print(neighbors[0])
print("Distances: ")
print(neighbors[1])
```

Alternatively, one could use the built in KD-Tree functionality of scipy to do nearest neighbor queries:

```
import laspy
import scipy
#from scipy.spatial.kdtree import KDTree
import numpy as np

# Open a file in read mode:
inFile = laspy.file.File("./laspytest/data/simple.las")
# Grab a numpy dataset of our clustering dimensions:
dataset = np.vstack([inFile.X, inFile.Y, inFile.Z]).transpose()
# Build the KD Tree
```

```
tree = scipy.spatial.kdtree(data)
# This should do the same as the FLANN example above, though it might
# be a little slower.
tree.query(dataset[100,], k = 5)
```

For another example, lets say we're interested only in the last return from each pulse in order to do ground detection. We can easily figure out which points are the last return by finding out for which points return_num is equal to num_returns.

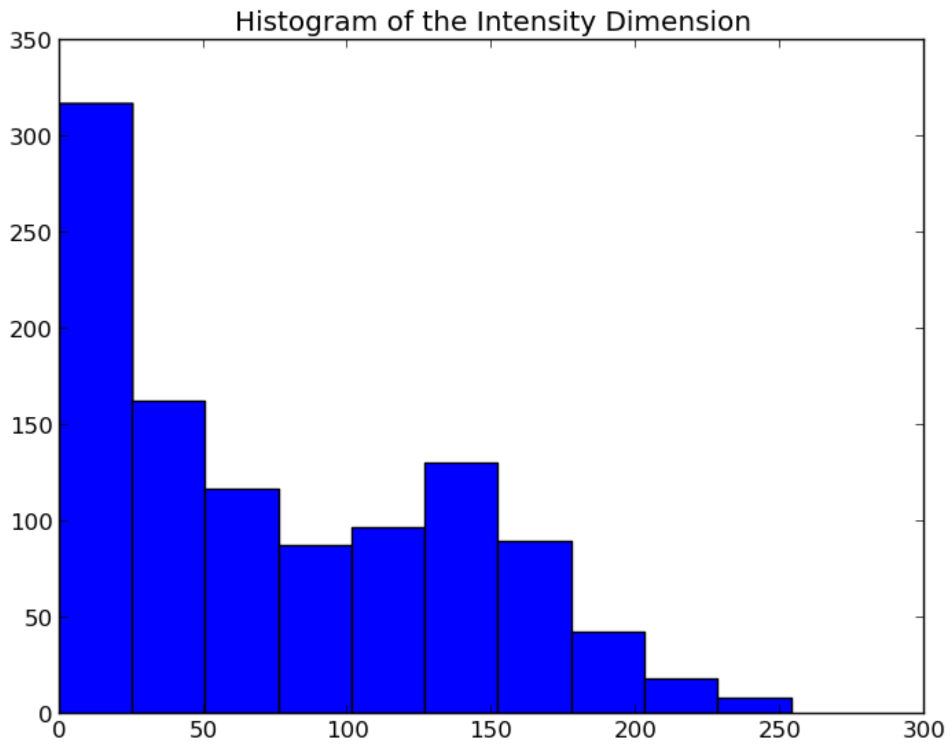
Note: Unpacking a bit field like num_returns can be much slower than a whole byte, because the whole byte must be read by numpy and then converted in pure python.

```
# Grab the return_num and num_returns dimensions
num_returns = inFile.num_returns
return_num = inFile.return_num
ground_points = inFile.points[num_returns == return_num]

print("%i points out of %i were ground points." % (len(ground_points),
        len(inFile)))
```

Since the data are simply returned as numpy arrays, we can use all sorts of analysis and plotting tools. For example, if you have matplotlib installed, you could quickly make a histogram of the intensity dimension:

```
import matplotlib.pyplot as plt
plt.hist(inFile.intensity)
plt.title("Histogram of the Intensity Dimension")
plt.show()
```



Writing Data

Once you've found your data subsets of interest, you probably want to store them somewhere. How about in new .LAS files?

When creating a new .LAS file using the write mode of `laspy.file.File`, we need to provide a `laspy.header.Header` instance, or a `laspy.header.HeaderManager` instance. We could instantiate a new instance without much input, but it will make potentially untrue assumptions about the point and file format. Luckily, we have a HeaderManager (which has a header) ready to go:

```
outFile1 = File("./laspytest/data/close_points.las", mode = "w",
                header = inFile.header)
outFile1.points = points_kept
outFile1.close()

outFile2 = File("./laspytest/data/ground_points.las", mode = "w",
                header = inFile.header)
outFile2.points = ground_points
outFile2.close()
```

For another example, let's return to the bounding box script above. Let's say we want to keep only points which fit within the given bounding box, and store them to a new file:

```
import laspy
import numpy as np

inFile = laspy.file.File("/path/to/lasfile", mode = "r")

# Get arrays which indicate VALID X, Y, or Z values.

X_invalid = np.logical_and((inFile.header.min[0] <= inFile.x),
                          (inFile.header.max[0] >= inFile.x))
Y_invalid = np.logical_and((inFile.header.min[1] <= inFile.y),
                          (inFile.header.max[1] >= inFile.y))
Z_invalid = np.logical_and((inFile.header.min[2] <= inFile.z),
                          (inFile.header.max[2] >= inFile.z))
good_indices = np.where(np.logical_and(X_invalid, Y_invalid, Z_invalid))
good_points = inFile.points[good_indices]

output_file = File("/path/to/output/lasfile", mode = "w", header = inFile.header)
output_file.points = good_points
output_file.close()
```

That covers the basics of read and write mode. If, however, you'd like to modify a las file in place, you can open it in read-write mode, as follows:

```
import laspy
inFile = laspy.file.File("./laspytest/data/close_points.las", mode = "rw")

# Let's say the X offset is incorrect:
old_location_offset = inFile.header.offset
old_location_offset[0] += 100
inFile.header.offset = old_location_offset

# Lets also say our Y and Z axes are flipped.
Z = inFile.Z
Y = inFile.Y
inFile.Y = Z
inFile.Z = Y

# Enough changes, let's go ahead and close the file:
```

```
inFile.close()
```

Variable Length Records

Variable length records, or VLRs, are available in laspy as file.header.vlrs. This property will return a list of `laspy.header.VLR` instances, each of which has a header which defines the type and size of their record. There are two fields which together determine the type of VLR: user_id and record_id. For a summary of what these fields might mean, refer to the "Defined Variable Length Records" section of the LAS specification. These fields are not required to be known values, however unless they are standard record types, laspy will simply treat the body of the VLR as dumb bytes.

To create a VLR, you really only need to know user_id, record_id, and the data you want to store in VLR_body (For a fuller discussion of what a VLR is, see the background section). The rest of the attributes are filled with null bytes or calculated according to your input, but if you'd like to specify the reserved or description fields you can do so with additional arguments.

Note: If you are creating a known type of VLR, you will still need to fill the VLR_body with enough bytes to fit the data you need before manipulating it in human readable form via parsed_body. This part of laspy is still very much under development, so feedback on how it should function would be greatly appreciated.

```
# Import the :obj:`laspy.header.VLR` class.

import laspy

inFile = laspy.file.File("./laspytest/data/close_points.las", mode = "rw")
# Instantiate a new VLR.
new_vlr = laspy.header.VLR(user_id = "The User ID", record_id = 1,
              VLR_body = "\x00" * 1000)
# The \x00 represents what's called a "null byte"
# Do the same thing without keyword args
new_vlr = laspy.header.VLR("The User ID", 1, "\x00" * 1000)
# Do the same thing, but add a description field.
new_vlr = laspy.header.VLR("The User ID",1, "\x00" * 1000,
                description = "A description goes here.")

# Append our new vlr to the current list. As the above dataset is derived
# from simple.las which has no VLRS, this will be an empty list.
old_vlrs = inFile.header.vlrs
old_vlrs.append(new_vlr)
inFile.header.vlrs = old_vlrs
inFile.close()
```

Putting it all together.

Here is a collection of the code on this page, copypaste ready:

```
import numpy as np
import laspy
inFile = laspy.file.File("./laspytest/data/simple.las", mode = "r")
# Grab all of the points from the file.
point_records = inFile.points

# Grab just the X dimension from the file, and scale it.
def scaled_x_dimension(las_file):
    x_dimension = las_file.X
    scale = las_file.header.scale[0]
    offset = las_file.header.offset[0]
    return(x_dimension*scale + offset)
```

```
scaled_x = scaled_x_dimension(inFile)

# Find out what the point format looks like.
print("Examining Point Format: ")
pointformat = inFile.point_format
for spec in inFile.point_format:
    print(spec.name)

#Like XML or etree objects instead?
print("Grabbing xml...")
a_mess_of_xml = pointformat.xml()
an_etree_object = pointformat.etree()

#It looks like we have color data in this file, so we can grab:
blue = inFile.blue

#Lets take a look at the header also.
print("Examining Header Format:")
headerformat = inFile.header.header_format
for spec in headerformat:
    print(spec.name)

print("Find close points...")
# Grab the scaled x, y, and z dimensions and stick them together
# in an nx3 numpy array

coords = np.vstack((inFile.x, inFile.y, inFile.z)).transpose()

# Pull off the first point
first_point = coords[0,:]

# Calculate the euclidean distance from all points to the first point

distances = np.sum((coords - first_point)**2, axis = 1)

# Create an array of indicators for whether or not a point is less than
# 500000 units away from the first point

keep_points = distances < 500000

# Grab an array of all points which meet this threshold

points_kept = inFile.points[keep_points]

print("We're keeping %i points out of %i total"%(len(points_kept), len(inFile)))


print("Find ground points...")
# Grab the return_num and num_returns dimensions
num_returns = inFile.num_returns
return_num = inFile.return_num
ground_points = inFile.points[num_returns == return_num]

print("%i points out of %i were ground points." % (len(ground_points),
        len(inFile)))


print("Writing output files...")
```

```
outFile1 = File("./laspytest/data/close_points.las", mode = "w",
                header = inFile.header)
outFile1.points = points_kept
outFile1.close()

outFile2 = File("./laspytest/data/ground_points.las", mode = "w",
                header = inFile.header)
outFile2.points = ground_points
outFile2.close()


print("Trying out read/write mode.")
inFile = File("./laspytest/data/close_points.las", mode = "rw")

# Let's say the X offset is incorrect:
old_location_offset = inFile.header.offset
old_location_offset[0] += 100
inFile.header.offset = old_location_offset

# Lets also say our Y and Z axes are flipped.
Z = inFile.Z
Y = inFile.Y
inFile.Y = Z
inFile.Z = Y

# Enough changes, let's go ahead and close the file:
inFile.close()


print("Trying out VLRs...")

inFile = File("./laspytest/data/close_points.las", mode = "rw")
# Instantiate a new VLR.
new_vlr = laspy.header.VLR(user_id = "The User ID", record_id = 1,
              laspy.header.VLR_body = "\x00" * 1000)
# Do the same thing without keyword args
new_vlr = laspy.header.VLR("The User ID", 1, "\x00" * 1000)
# Do the same thing, but add a description field.
new_vlr = laspy.header.VLR("The User ID",1, "\x00" * 1000,
                description = "A description goes here.")

# Append our new vlr to the current list. As the above dataset is derived
# from simple.las which has no VLRS, this will be an empty list.
old_vlrs = inFile.header.vlrs
old_vlrs.append(new_vlr)
inFile.header.vlrs = old_vlrs
inFile.close()
```

1.4 The more complicated stuff

Using laspy's public api from `laspy.file.File` and `laspy.header.HeaderManager` objects will get you a long way, but sometimes it's neccesary to dig a little deeper. For example, if you would like to build a 1.1 version file from a 1.2 version file, there is no automatic function to do this for you. Life becomes easier when we dig into some of laspy's internal functionality:

```
import laspy
import copy

# Open an input file in read mode.
inFile = laspy.file.File("./laspytest/data/simple.las",mode= "r")

# Call copy on the HeaderManager object to get a more portable Header instance.
# This means we don't  have to modify the header on the read mode inFile.
new_header = copy.copy(inFile.header)
# Update the fields we want to change, the header format and data_format_id
new_header.format = 1.1
new_header.pt_dat_format_id = 0

# Now we can create a new output file with our modified header.
# Note that we need to give the file the VLRs manually, because the low level
# header doesn't know about them, while the header manager does.
outFile = laspy.file.File("./laspytest/data/output.las",
                    mode= "w",
                    vlrs = inFile.header.vlrs,
                    header = new_header)

# Iterate over all of the available point format specifications, attepmt to
# copy them to the new file. If we fail, print a message.

# Take note of the get_dimension and set_dimension functions. These are
# useful for automating dimension oriented tasks, because they just require
# the spec name to do the lookup.

for spec in inFile.reader.point_format:
    print("Copying dimension: " + spec.name)
    in_spec = inFile.reader.get_dimension(spec.name)
    try:
        outFile.writer.set_dimension(spec.name, in_spec)
    except(util.LaspyException):
        print("Couldn't set dimension: " + spec.name +
                " with file format " + str(outFile.header.version) +
                ", and point_format " + str(outFile.header.data_format_id))

# Close the file

outFile.close()
```

1.5 New Format Features: LAS Versions 1.3 and 1.4

There is not a great deal of LAS version 1.3 and 1.4 test data around, so laspy's implementation of these formats is neccesarily preliminary. Nevertheless, based on the test data we've been able to find, all the basic features should work.

The dimensions which are newly available from point formats 4-10 are accessed in the same way as those from formats 0-3. The tutorial background section has an extensive list of what laspy calls each of these dimensions, but here we'll provide some quick examples. More examples can be found by looking at the ./test/test_laspy.py file included in the source.

Additionally, LAS version 1.4 provides the ability to specify an "extra bytes" variable length record (or EVLR) which can dynamically add additional dimensins. Laspy now parses such records, and provides the specified dimensions accordingly.

The names of these new dimensions are constructed by using the name field specified in the VLR record, and replacing null bytes with python empty strings, spaces with underscores, and upper case letters with lower case letters. For example, the field

"Pulse Width\X00\X00\X00\X00\X00\X00\X00\X00\X00"

would become simply: "pulse_width"

In order to maintain backwards compatability, laspy also provides access to these dimensions via `laspy.file.extra_bytes`, which provides raw access to the extra bytes in point records (present when data_record_length is greater than the default for a given point format).

Opening Files

Opening 1.3 and 1.4 files works exactly the same:

```
import numpy as np
import laspy

inFile_v13 = laspy.file.File("./laspytest/data/simple1_3.las", mode = "r")
inFile_v14 = laspy.file.File("./laspytest/data/simple1_4.las", mode = "r")
```

Reading Data - New Dimensions

```
#By checking the data_format_ids, we can see what new
# dimensions are present. Our 1.3 file has data format 4,
# and our 1.4 file has data format 7.
inFile_v13.header.data_format_id
inFile_v14.header.data_format_id

#Grab some dimensions as usual:
v_13 points = inFile_v13.points
x_t = inFile_v13.x_t

v_14_points = inFile_v14.points
v_14_classification = inFile_v14.classification

# Note that classification means different things depending on
# the file version. Before v1.4, classification was part of the
# classification byte. This is called classification flags in 1.4,
# and classification refers to a new whole-byte field. For a
# discussion, see tutorial:background.
```

Writing Data + EVLRS

EVLRS work very much the same way as traditional VLRs, though they are stored in a different part of the file.

```
import laspy
outFile_14 = laspy.file.File("./laspytest/data/output_14.las", mode = "w",
                header = inFile_v14.header)
new_evlr = laspy.header.EVLR(user_id = 10, record_id = 2,
                VLR_body = "Lots of data can go here.")
#outFile_14 has the same, single EVLR as inFile
old_evlrs = outFile_14.header.evlrs
old_evlrs.append(new_evlr)
outFile_14.header.evlrs = old_evlrs
outFile_14.close()
```

Extra Bytes

The extra bytes in a point record can now be described by a particular type of VLR. From the LAS 1.4 specification, a VLR which describes new dimensions should have the following header information:

User ID: LASF_Spec

Record ID: 4

Record Length after Header: n x 192 bytes

where n is the number of new dimensions that the VLR will define. The actual dimension specification goes in the body of the VLR, and has the following structure:

Note: Laspy coerces the no_data, max and min fields to have double precision format. If this is a problem for your application, let us know.

Extra Bytes Struct

Name	Format[number] (Total Bytes)
reserved	unsigned char[2] (2)
data_type	unsigned char[1] (1)
options	unsigned char[1] (1)
name	char[32] (32)
unused	char[4] (4)
no_data	double[3] (24)
min	double[3] (24)
max	double[3] (24)
scale	scale[3] (24)
offset	offset[3] (24)
description	char[32] (24)

Data Type Description

Value	Meaning	Size
0	Raw Extra Bytes	Value of “options”
1	unsigned char	1 byte
2	Char	1 byte
3	unsigned short	2 bytes
4	Short	2 bytes
5	unsigned long	4 bytes
6	Long	4 bytes
7	unsigned long long	8 bytes
8	long long	8 bytes
9	Float	4 bytes
10	Double	8 bytes
11	unsigned char[2]	2 byte
12	char[2]	2 byte
13	unsigned short[2]	4 bytes
14	short[2]	4 bytes
15	unsigned long[2]	8 bytes
16	long[2]	8 bytes
17	unsigned long long[2]	16 bytes
18	long long[2]	16 bytes
19	float[2]	8 bytes
20	double[2]	16 bytes
21	unsigned char[3]	3 byte
22	char[3]	3 byte
23	unsigned short[3]	6 bytes
24	short[3]	6 bytes

Continued on next page

Table 1.1 – continued from previous page

Value	Meaning	Size
25	unsigned long[3]	12 bytes
26	long[3]	12 bytes
27	unsigned long long[3]	24 bytes
28	long long[3]	24 bytes
29	float[3]	12 bytes
30	double[3]	24 bytes

Adding Extra Dimensions - The laspy way.

One can easily create new dimensions using the above data type table and a laspy file object. In fact, it is not even neccesary to use a 1.4 file in this process, however other software will likely not know to use the new 1.4 features in a previous file version. Most readers should, however, be able to treat the extra dimensions as extra bytes. Here's the easy way to specify new dimensions:

```
import laspy

# Set up our input and output files.
inFile = laspy.file.File("./laspytest/data/simple.las", mode = "r")
outFile = laspy.file.File("./laspytest/data/output.las", mode = "w",
            header = inFile.header)
# Define our new dimension. Note, this must be done before giving
# the output file point records.
outFile.define_new_dimension(name = "my_special_dimension",
                        data_type = 5, description = "Test Dimension")

# Lets go ahead and copy all the existing data from inFile:
for dimension in inFile.point_format:
    dat = inFile.reader.get_dimension(dimension.name)
    outFile.writer.set_dimension(dimension.name, dat)

# Now lets put data in our new dimension
# (though we could have done this first)

# Note that the data type 5 refers to a long integer
outFile.my_special_dimension = range(len(inFile))
```

Adding Extra Dimensions - The long way.

If you want to see what's happening when you create new dimensions at a level much closer to the raw specification, laspy lets you create the requisite components manually.

```
import laspy
import copy

inFile = laspy.file.File("./laspytest/data/simple.las", mode = "r")

# We need to build the body of our dimension VLRs, and to do this we
# will use a class called ExtraBytesStruct. All we really need to tell
# it at this point is the name of our dimension and the data type.

extra_dimension_spec_1 = laspy.header.ExtraBytesStruct(name = "My Super Special Dimension",
                                                        data_type = 5)
extra_dimension_spec_2 = laspy.header.ExtraBytesStruct(name = "Another Special Dimension",
                                                        data_type = 5)
vlr_body = (extra_dimension_spec_1.to_byte_string() +
           extra_dimension_spec_2.to_byte_string())
```

```
# Now we can create the VLR. Note the user_id and record_id choices.
# These values are how the LAS specification determines that this is an
# extra bytes record. The description is just good practice.
extra_dim_vlr = laspy.header.VLR(user_id = "LASF_Spec",
                                 record_id = 4,
                                 description = "Testing Extra Bytes.",
                                 VLR_body = vlr_body)


# Now let's put together the header for our new file. We need to increase
# data_record_length to fit our new dimensions. See the data_type table
# for details. We also need to change the file version
new_header = copy.copy(inFile.header)
new_header.data_record_length += 8
new_header.format = 1.4

# Now we can create the file and give it our VLR.
new_file = laspy.file.File("./laspytest/data/new_14_file.las", mode = "w",
                header = new_header, vlrs = [extra_dim_vlr])

# Let's copy the existing data:
for dimension in inFile.point_format:
    dim = inFile._reader.get_dimension(dimension.name)
    new_file._writer.set_dimension(dimension.name, dim)

# We should be able to acces our new dimensions based on the
# Naming convention described above. Let's put some dummy data in them.
new_file.my_super_special_dimension = [0]*len(new_file)
new_file.another_special_dimension = [10]*len(new_file)

# If we would rather grab a raw byte representation of all the extra
# dimensions, we can use:

raw_bytes = new_file.extra_bytes

# This might be useful if we wanted to later take this data and put it
# back into an older file version which doesn't support extra dimensions.
```

API Documentation

2.1 File

The File class is the core laspy tool. It provides access to point records and dimensions via the `laspy.base.reader` and `laspy.base.writer` classes, and holds a reference to a `laspy.header.HeaderManager` instance to provide header reading and modifying capability. It is also iterable and sliceable.

Dimensions: In addition to grabbing whole point records, it is possible to obtain and set individual dimensions as well. The dimensions available to a particular file depend on the point format, a summary of which is available via the File.point_format.xml() method. Dimensions might be used as follows:

```
# Flip the X and Y Dimensions
>>> FileObject = file.File("./path_to_file", mode = "rw")
>>> X = FileObject.X
>>> Y = FileObject.Y
>>> FileObject.X = Y
>>> FileObject.Y = X
>>> FileObject.close()

# Print out a list of available point dimensions:
>>> for dim in FileObject.point_format:
>>>     print(i.name)
# Alternately, grab descriptive xml:
>>> FileObject.point_format.xml()
```

2.2 Header

Header Module

The laspy header module holds the low level `laspy.header.Header` class, which both stores header information during a laspy session, and also provides a container for moving header data around. Most of the header API is located in the `laspy.header.HeaderManager` class, which holds a `laspy.header.Header` instance. This is accessed from a `laspy.file.File` object as `laspy.file.File`.header

Additionally, this module holds the VLR and EVLR classes, for regular and extended variable length records as defined in the various LAS specifications.

Finally, this module provides ExtraBytesStruct, which is a frontend for defining additional dimensions in the LAS file via an Extra Bytes type VLR. See the tutorial for an example of this (the LAS specification is also a helpful reference.)

2.3 Base

Base Module Basics:

Most of the functionality of laspy is exposed on the file and header modules, however it will sometimes be convenient to dig into the base as well. The two workhorses of the base class are Writer and Reader, and they are both subclasses of FileManager, which handles a lot of the file initialization logic, as well as file reading capability.

2.4 Util

The laspy util module holds useful data structures and functions needed in multiple locations, but not belonging unambiguously to File, Reader/Writer, or Header

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*