Pylidar Documentation

Release 0.4.4

John Armston, Pete Bunting, Neil Flood, Sam Gillingham

Aug 19, 2020

Contents

CHAPTER 1

Introduction

A set of Python modules which makes it easy to write lidar processing code in Python. Based on SPDLib and built on top of RIOS it handles the details of opening and closing files, checking alignment of projection and grid, stepping through the data in small blocks, etc., allowing the programmer to concentrate on the processing involved. It is licensed under GPL 3.

See spdv4format for description of the SPD V4 file format. Supported formats are: SPD V3, SPD V4, RIEGL RXP, LAS, LVIS, ASCII and Pulsewaves (additional libraries may be required).

See the arrayvisualisation page to understand how numpy arrays are used in PyLidar.

Work funded by:

- DSITI and OEH through the Joint Remote Sensing Research Program
- University of Maryland

There is a Google Group where users can post questions.

CHAPTER 2

Examples

See processorexamples for more information on programming using PyLidar. See the following links for more information on running the command line utilities:

- commandline_translate
- commandline_index
- commandline_info
- commandline_tiles
- commandline_canopy

CHAPTER 3

Downloads

3.1 Source

Source code is available from GitHub. RIOS, Numba, Numpy and h5py are required dependencies. Additional formats require environment variables set to the root installation of other libraries before building as detailed in this table:

Type of Files	Environment Variable(s)	Link to Software
LAS/LAZ	LASTOOLS_ROOT	lastools
Riegl	RIVLIB_ROOT RIWAVELIB_ROOT RDBLIB_ROOT	RiVLIB RiWaveLIB RDBLib
ASCII .gz	ZLIB_ROOT	zlib
PulseWaves	PULSEWAVES_ROOT	pulsewaves

The related pynninterp module is used for some interpolation operations.

3.2 Test Suite

After installation, run pylidar_test to run a number of tests to check that the install is OK. You will need the latest testdata_X.tar.gz file (with the highest 'X') from the links in the wiki page. Pass the path to this file to pylidar_test with the -i option.

3.3 Conda

Conda packages are available under the 'rios' channel. Once you have installed Conda, run the following commands on the command line to install pylidar (dependencies are obtained automatically):

```
conda config --add channels conda-forge
conda config --add channels rios
conda create -n myenv pylidar
conda activate myenv
```

The related pynninterp module is used for some interpolation operations and can be installed via Conda also from the 'rios' channel:

```
conda install pynninterp
```

CHAPTER 4

Processing

4.1 userclasses

Classes that are passed to the user's function

class pylidar.userclasses.**DataContainer**(*controls*)

This is a container object used for passing as the first parameter to the user function. It contains a UserInfo object (called 'info') plus instances of LidarData and ImageData (see below). These objects will be named in the same way that the LidarFile and ImageFile were in the DataFiles object that was passed to doProcessing().

class pylidar.userclasses.**ImageData**(*mode*, *driver*)

Class that allows reading and writing to/from an image file. Passed to the user function from a field on the DataContainer object.

Calls though to the driver instance it was constructed with to do the actual work.

flush()

Now actually do the write

getData()

Returns the data for the current extent as a 3d numpy array in the same data type as the image file.

setData(*data*)

Sets the image data for the current extent. The data type of the passed in numpy array will be the data type for the newly created file.

class pylidar.userclasses.**LidarData**(*mode*, *driver*)

Class that allows reading and writing to/from a LiDAR file. Passed to the user function from a field on the DataContainer object.

Calls though to the driver instance it was constructed with to do the actual work.

static convertToStructIfNeeded(*data*, *colName*, *oldData=None*)

Converts data to a structured array if it is not. If conversion is required it uses colName and data type of data. Raises exception if conversion not possible or does not make sense.

if oldData is not None and data is non-structured, then the new data is appended onto oldData and returned. If data is structured, and error is raised.

flush()
writes data to file set via the set*() functions

getHeader()
Returns the header as a dictionary of header key/value pairs.

getHeaderTranslationDict()
Return a dictionary keyed on HEADER_* values (above) that can be used to translate dictionary field names between the formats

getHeaderValue(*name*)
Gets a particular header value with the given name

getNativeDataType(*colName*, *arrayType*)
Return the native dtype (numpy.int16 etc)that a column is stored as internally after scaling (if any) is applied. Provided so scaling can be adjusted when translating between formats.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

generic.LiDARArrayColumnError is raised if information cannot be found.

getNullValue(*colName*, *arrayType*, *scaled=True*)
Get the 'null' value for the given column.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

By default the returned value is scaled, change this with the 'scaled' parameter.

Raises generic.LiDARArrayColumnError if information cannot be found for the column.

getPoints(*colNames=None*)
Returns the points for the extent/range of the current block as a structured array. The fields on this array are defined by the driver being used.

colNames can be a name or list of column names to return. By default all columns are returned.

getPointsByBins(*extent=None*, *colNames=None*, *indexByPulse=False*, *returnPulseIndex=False*)
Returns the points for the extent of the current block as a 3 dimensional structured masked array. Only valid for spatial processing. The fields on this array are defined by the driver being used.

First axis is the points in each bin, second axis is the rows, third is the columns.

Some bins have more points that others so the mask is set to True when data not valid.

The extent/binning for the read data can be overriden by passing in a basedriver.Extent instance.

colNames can be a name or list of column names to return. By default all columns are returned.

Set indexByPulse to True to bin points by the pulse index location rather than point location.

Set returnPulseIndex to True to also return a 3 dimensional masked array containing the indexes into the 1d array returned by getPulses().

getPointsByPulse(*colNames=None*)
Returns the points as a 2d structured masked array. The first axis is the same length as the pulse array but the second axis contains the points for each pulse. The mask will be set to True where no valid data since some pulses will have more points than others.

colNames can be a name or list of column names to return. By default all columns are returned.

getPulses(*colNames=None*, *pulseIndex=None*)
Returns the pulses for the extent/range of the current block as a structured array. The fields on this array are defined by the driver being used.

colNames can be a name or list of column names to return. By default all columns are returned.

pulseIndex is an optional masked 3d array of indices to remap the 1d pulse array to a 3D point by bin array. pulseIndex is returned from getPointsByBins with returnPulseIndex=True.

getPulsesByBins(*extent=None*, *colNames=None*)
Returns the pulses for the extent of the current block as a 3 dimensional structured masked array. Only valid for spatial processing. The fields on this array are defined by the driver being used.

First axis is the pulses in each bin, second axis is the rows, third is the columns.

Some bins have more pulses that others so the mask is set to True when data not valid.

The extent/binning for the read data can be overriden by passing in a basedriver.Extent instance.

colNames can be a name or list of column names to return. By default all columns are returned.

getReceived()
Returns a masked 3d radiance array. The first axis is the waveform bins, the second axis will be the waveform number and the third axis axis will be the same length as the pulses.

Because some pulses will have a longer waveform than others a masked array is returned.

getScaling(*colName*, *arrayType*)
Returns the scaling (gain, offset) for the given column name

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

getScalingColumns(*arrayType*)
Return a list of column names that require scaling to be set on write.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

getTransmitted()
Returns a masked 3d radiance array. The first axis is the waveform bins, the second axis will be the waveform number and the third axis axis will be the same length as the pulses.

Because some pulses will have a longer waveform than others a masked array is returned.

getWaveformInfo()
Returns a 2d masked structured array with information about the waveforms. First axis will be the waveform number, second will be same length as the pulses

rebinPtsByHeight(*pointsByBin*, *bins*, *heightArray=None*, *heightField='Z'*)
pointsByBin 3d ragged (masked) structured array of points. (nrows, ncols, npts) bins Height bins into which to stratify points

Set heightArray to a masked array of values used to vertically stratify the points. This allows columns not in pointsByBin to be used.

Set heightField to specify which pointsByBin column name to use for height values. Only used if heightArray is None.

Return: 4d re-binned copy of pointsByBin

setHeader(*headerDict*)
Sets header values as a dictionary of header key/value pairs.

setHeaderValue(*name*, *value*)
Sets a particular header value with the given name

setHeaderValues(***kwargs*)
Overloaded version to support key word args instead

setNativeDataType(*colName*, *arrayType*, *dtype*)
Set the native dtype (numpy.int16 etc)that a column is stored as internally after scaling (if any) is applied.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

generic.LiDARArrayColumnError is raised if information cannot be found.

setNullValue(*colName*, *arrayType*, *value*, *scaled=True*)
Set the 'null' value for the given column.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

By default the value is treated as the scaled value, but this can be changed with the 'scaled' parameter.

generic.LiDARArrayColumnError is raised if this cannot be set for the column.

setPoints(*points*, *colName=None*)
Write the points to a file. If passed a structured array, the same field names are expected as those read with the same driver.

If the array is non-structured (ie you passed a colNames as a string to getPoints()) you need to pass the same string as the colName parameter.

Pass either a 1d array (like that read from getPoints()) or a 3d masked array (like that read from getPointsByBins()).

setPulses(*pulses*, *colName=None*)
Write the pulses to a file. If passed a structured array, the same field names are expected as those read with the same driver.

If the array is non-structured (ie you passed a colNames as a string to getPulses()) you need to pass the same string as the colName parameter.

Pass either a 1d array (like that read from getPulses()) or a 3d masked array (like that read from getPulsesByBins()).

setReceived(*received*)
Set the received waveform for each pulse as a masked 3d integer array.

setScaling(*colName*, *arrayType*, *gain*, *offset*)
Set the scaling for the given column name

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

setTransmitted(*transmitted*)
Set the transmitted waveform for each pulse as a masked 3d integer array.

setWaveformInfo(*info*, *colName=None*)
Set the waveform info as a masked 2d array.

If passed a structured array, the same field names are expected as those read with the same driver.

If the array is non-structured (ie you passed a colNames as a string to getWaveformInfo()) you need to pass the same string as the colName parameter.

translateFieldNames(*otherLidarData*, *array*, *arrayType*)
Translates the field names in an array from another format (specified by passing the other data object) for use in writing to the format for this driver. The array is passed in and updated directly (no copy made). The array is returned.

arrayType is one of the ARRAY_TYPE_* values defined in lidarprocessor.py.

class pylidar.userclasses.**UserInfo**(*controls*)

The 'DataContainer' object (below) contains an 'info' field which is an instance of this class. The user function can use these methods to obtain information on the current processing state and region.

Equivalent to the RIOS 'info' object.

getBlockCoordArrays()

Return a tuple of the world coordinates for every pixel in the current block. Each array has the same shape as the current block. Return value is a tuple:

```
(xBlock, yBlock)
```

where the values in xBlock are the X coordinates of the centre of each pixel, and similarly for yBlock.

The coordinates returned are for the pixel centres. This is slightly inconsistent with usual GDAL usage, but more likely to be what one wants.

getControls()

Return the instance of the controls object used for processing

getExtent()

Get the extent of the current block being procesed. This is only valid when spatial processing is enabled. Otherwise use getRange()

This is an instance of .basedriver.Extent.

getPixGrid()

Return the current pixgrid. This defines the current total processing extent, resolution and projection.

Is an instance of rios.pixelgrid.PixelGridDefn.

getRange()

Get the range of pulses being processed. This is only vaid when spatial processing is disabled. When doing spatial processing, use getExtent().

isFirstBlock()

Returns True if this is the first block to be processed

isLastBlock()

Returns True if this is the last block to be processed

setExtent(*extent*)

For internal use. Used by the processor to set the current state.

setPixGrid(*pixGrid*)

For internal use. Used by the processor to set the current state.

setRange(*range*)

For internal use. Used by the processor to set the current state.

- genindex
- modindex
- search

4.2 LiDAR Processor

Classes that are passed to the doProcessing function. And the doProcessing function itself

pylidar.lidarprocessor.**ARRAY_TYPE_POINTS = 0**

For use in userclass.LidarData.translateFieldNames() and LiDARFile.getTranslationDict()

pylidar.lidarprocessor.**ARRAY_TYPE_PULSES = 1**
For use in userclass.LidarData.translateFieldNames() and LiDARFile.getTranslationDict()

pylidar.lidarprocessor.**ARRAY_TYPE_WAVEFORMS = 2**
For use in userclass.LidarData.translateFieldNames() and LiDARFile.getTranslationDict()

pylidar.lidarprocessor.**BOUNDS_FROM_REFERENCE = <Mock name='mock.imageio.BOUNDS_FROM_REFEREI**
to be passed to Controls.setFootprint()

pylidar.lidarprocessor.**CLASSIFICATION_BRANCH = 102**
Extended classification codes

pylidar.lidarprocessor.**CLASSIFICATION_BRIDGE = 12**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_BUILDING = 6**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_CREATED = 0**
Classification codes from the LAS spec. Drivers perform automatic translation to/from their internal codes for recognised values.

pylidar.lidarprocessor.**CLASSIFICATION_FOLIAGE = 101**
Extended classification codes

pylidar.lidarprocessor.**CLASSIFICATION_GROUND = 2**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_HIGHPOINT = 8**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_HIGHVEGE = 5**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_INSULATOR = 16**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_LOWPOINT = 7**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_LOWVEGE = 3**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_MEDVEGE = 4**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_RAIL = 10**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_ROAD = 11**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_TRANSTOWER = 15**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_TRUNK = 100**
Extended classification codes

pylidar.lidarprocessor.**CLASSIFICATION_UNCLASSIFIED = 1**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_WATER = 9**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_WIRECOND = 14**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CLASSIFICATION_WIREGUARD = 13**
Classification codes from the LAS spec.

pylidar.lidarprocessor.**CREATE = 2**
to be passed to ImageData and LidarData class constructors

class pylidar.lidarprocessor.**Controls**
The controls object. This is passed to the doProcessing function and contains methods for controling the behaviour of the processing.

setFootprint(*footprint*)
Set the footprint of the processing area. This should be either INTERSECTION, UNION or BOUNDS_FROM_REFERENCE.

Note: setting spatial processing to True now deprecated. Consider updating your code.

setMessageHandler(*messageHandler*)
Set the message handler function to use for printing messages regarding things discovered during the processing. The default behaviour is to print all messages.

Can pass in silentMessageFn which will print nothing, or your own function that takes a message string and a level (one of the MESSAGE_* constants).

setOverlap(*overlap*)
Sets the overlap between each window. In bins.

Note: setting spatial processing to True now deprecated. Consider updating your code.

setProgress(*progress*)
Set the progress instance to use. Usually one of rios.cuiprogress.* Default is silent progress

setReferenceImage(*referenceImage*)
The path to a reference GDAL image to use when the footprint is set to BOUNDS_FROM_REFERENCE. Set only one of this or referencePixgrid not both.

Note: setting spatial processing to True now deprecated. Consider updating your code.

setReferencePixgrid(*referencePixgrid*)
The instance of rios.pixelgrid.PixelGridDefn to use as a reference when footprint is set to BOUNDS_FROM_REFERENCE. Set only one of this or referenceImage, not both.

Note: setting spatial processing to True now deprecated. Consider updating your code.

setReferenceResolution(*resolution*)
Overrides the resolution that the processing happens with. Overrides either of the setReferenceImage or setReferencePixgrid calls or the default reference.

Note: setting spatial processing to True now deprecated. Consider updating your code.

setSnapGrid(*snap*)
Snap the output grid to be multiples of the resolution. This is only needed when ReferenceResolution is not set. True or False.

Note: setting spatial processing to True now deprecated. Consider updating your code.

setSpatialProcessing(*spatial*)
Set whether to do processing in a spatial manner. If set to True and if one of more LiDAR inputs do not support spatial indexing will be reset to False and warning printed.

Note: setting spatial processing to True now deprecated. Consider updating your code.

setWindowSize(*size*)
Size of the window in bins/pixels that the processing is to be performed in. Same in the X and Y direction. If doing non spatial processing 'size*size' pulses are read in at each iteration.

pylidar.lidarprocessor.**DEFAULT_WINDOW_SIZE = 256**
Size of the default window size in bins

class pylidar.lidarprocessor.**DataFiles**
Container class that has all instances of LidarFile and ImageFile inserted into it as the names they are to be used inside the users function.

pylidar.lidarprocessor.**INTERSECTION = <Mock name='mock.imageio.INTERSECTION' id='139696478**
to be passed to Controls.setFootprint()

class pylidar.lidarprocessor.**ImageFile**(*fname*, *mode*)

setRasterDriver(*driverName*)
Set GDAL driver short name to use for output format.

setRasterDriverOptions(*options*)
Set a list of strings in driver specific format. See GDAL documentation.

setRasterIgnore(*ignore*)
Set the ignore value for calculating statistics

class pylidar.lidarprocessor.**LidarFile**(*fname*, *mode*)
Create an instance of this to process a LiDAR file. Set it to a field within your instance of DataFiles. The mode is one of: READ, UPDATE or CREATE.

setLiDARDriver(*driverName*)
Set the name of the Lidar driver to use for creaton

setLiDARDriverOption(*key*, *value*)
Set a key and value that the specific driver understands

setWriteSpatialIndex(*writeSpatialIndex*)
Set whether to write spatial index or not on creation or update. Ignored for reading.

pylidar.lidarprocessor.**MESSAGE_DEBUG = 2**
to be passed to message handler function set with Controls.setMessageHandler

pylidar.lidarprocessor.**MESSAGE_INFORMATION = 1**
to be passed to message handler function set with Controls.setMessageHandler

pylidar.lidarprocessor.**MESSAGE_WARNING = 0**
to be passed to message handler function set with Controls.setMessageHandler

class pylidar.lidarprocessor.**OtherArgs**
Container class that has any arbitary information that the user function requires. Set in the same form as DataFiles above, but no conversion of the contents happens.

pylidar.lidarprocessor.**READ = 0**
to be passed to ImageData and LidarData class constructors

pylidar.lidarprocessor.**UNION = <Mock name='mock.imageio.UNION' id='139696478360976'>**
to be passed to Controls.setFootprint()

pylidar.lidarprocessor.**UPDATE = 1**
to be passed to ImageData and LidarData class constructors

pylidar.lidarprocessor.**defaultMessageFn**(*message*, *level*)
Default message printer. Prints all messages regardless of level.

Change with Controls.setMessageHandler

pylidar.lidarprocessor.**doProcessing**(*userFunc*, *dataFiles*, *otherArgs=None*, *controls=None*)
Main function in PyLidar. Calls function userFunc with each block of data. dataFiles to be an instance of DataFiles with fields of instances of LidarFile and ImageFile. The names of the fields are re-used in the object passed to userFunc that contains the actual data.

If otherArgs (an instance of OtherArgs) is not None, this is passed as the second param to userFunc.

If controls (an instance of Controls) is not None then these controls are used for changing the behaviour of reading and writing.

pylidar.lidarprocessor.**findCommonPixelGridRegion**(*gridList*, *refGrid*, *combine=<Mock name='mock.imageio.INTERSECTION' id='139696478328592'>*)
Returns a PixelGridDefn for the combination of all the grids in the given gridList. The output grid is in the same coordinate system as the reference grid.

This is adapted from the original in RIOS. This version does not attempt to reproject between coordinate systems. Firstly, because many LiDAR files do not seem to have the projection set. Secondly, we don't support reprojection anyway - unlike RIOS.

The combine parameter controls whether UNION, INTERSECTION or BOUNDS_FROM_REFERENCE is performed.

pylidar.lidarprocessor.**getWorkingPixGrid**(*controls*, *userContainer*, *gridList*, *driverList*)
Calculates the working pixel grid and informs the drivers and userContainer.

pylidar.lidarprocessor.**openFiles**(*dataFiles*, *userContainer*, *controls*)
Open all the files required by doProcessing

pylidar.lidarprocessor.**setDefaultDrivers**()
Adapted from RIOS Sets some default values into global variables, defining what defaults we should use for GDAL and LiDAR drivers. On any given output file these can be over-ridden, and can be over-ridden globally using the environment variables (for GDAL):

- $PYLIDAR_DFLT_RASTERDRIVER
- $PYLIDAR_DFLT_RASTERDRIVEROPTIONS

(And for LiDAR):

- $PYLIDAR_DFLT_LIDARDRIVER

If PYLIDAR_DFLT_RASTERDRIVER is set, then it should be a gdal short driver name If PYLIDAR_DFLT_RASTERDRIVEROPTIONS is set, it should be a space-separated list of driver creation options, e.g. "COMPRESS=LZW TILED=YES", and should be appropriate for the selected GDAL driver. This can also be 'None' in which case an empty list of creation options is passed to the driver.

If not otherwise supplied, the default is to use what RIOS is set to. This defaults to the HFA driver with compression.

If PYLIDAR_DFLT_LIDARDRIVER is set, then is should be a LiDAR driver name If not otherwise supplied, the default is to use the SPDV4 driver.

pylidar.lidarprocessor.**silentMessageFn**(*message*, *level*)
Alternate message printer - does nothing.

- genindex
- modindex
- search

4.3 Array Utilities

Utility functions for use with pylidar.

pylidar.toolbox.arrayutils.**addFieldToStructArray**(*oldArray*, *newName*, *newType*, *newData=0*)

Creates a new array with all the data from oldArray, but with a new field as specified by newName and newType. newType should be one of the numpy types (numpy.uint16 etc). newData should either be a constant value, or an array of the correct shape and the new field will be initialised to this value.

- genindex
- modindex
- search

4.4 Toolbox

toolbox module. A bunch of useful tools for LiDAR Processing using pylidar.

4.4.1 Ground Filters

grdfilter module. A algorithms for filter ground returns using pylidar.

Classify Ground Returns

Functions to classify the ground returns within the point cloud.

pylidar.toolbox.grdfilters.classGrdReturns.**classifyGroundReturns**(*ptBinVals*, *grdSurf*, *thres*)

A function to classify the ground return points within a distance from a ground surface. * ptBinVals is a binned list of points * grdSurf ground surface with the same dimensions as the binned points * thres is the threshold for defining whether a point is ground or not.

Progressive Morphology Filter

Functions used to implement the progressive morphological filter algorithm (Zhang et al., 2003) to generate a raster surface which can be used to classify the ground returns.

Zhang, K., Chen, S., Whitman, D., Shyu, M., Yan, J., & Zhang, C. (2003). A progressive morphological filter for removing nonground measurements from airborne LIDAR data. IEEE Transactions on Geoscience and Remote Sensing, 41(4), 872-882.

pylidar.toolbox.grdfilters.pmf.**applyPMF**(*dataArr*, *noDataMask*, *binGeoSize*, *initWinSize=1*, *maxWinSize=12*, *winSizeInc=1*, *slope=0.3*, *dh0=0.3*, *dhmax=5*, *expWinSizes=False*)

Apply the progressive morphology filter (PMF) to the input data array (dataArr) filtering the surface to remove regions which are not ground.

- dataArr is a numpy array, usually defined as minimum Z LiDAR return within bin, on which the filter is to be applied.
- noDataMask is a numpy array specifying the cells in the input dataArr which do not have data (i.e., there were no points in the bin)

- binGeoSize is the geographic (i.e., in metres) size of each square bin (i.e., 1 m)
- initWinSize is the initial window size (Default = 1)
- maxWinSize is the maximum window size (Default = 12)
- winSizeInc is the increment for the window size (Default = 1)
- slope is the slope within the scene (Default = 0.3)
- dh0 is the initial height difference threshold for differentiating ground returns (Default = 0.3)
- dhmax is the maximum height difference threshold for differentiating ground returns (Default = 5)
- expWinSizes is a boolean specifying whether the windows sizes should increase exponentially or not (Default = False)

Returns: PMF Filtered array with the same data type as the input.

pylidar.toolbox.grdfilters.pmf.**disk**(*radius*, *dtype=<type 'numpy.uint8'>*)
Generates a flat, disk-shaped structuring element. A pixel is within the neighborhood if the euclidean distance between it and the origin is no greater than radius. Parameters:

- radius : int The radius of the disk-shaped structuring element.

Other Parameters:

- dtype : data-type The data type of the structuring element.

Returns:

- selem : ndarray The structuring element where elements of the neighborhood are 1 and 0 otherwise.

pylidar.toolbox.grdfilters.pmf.**doNearestNeighbourInterp**(*data*, *noDataMask*, *m*, *n*)
Function to do nearest neighbout interpolation of filling in no data area

pylidar.toolbox.grdfilters.pmf.**doOpening**(*iarray*, *maxWindowSize*, *winSize1*, *c*, *s*, *dh0*, *dhmax*)
A function to perform a series of iterative opening operations on the data array with increasing window sizes.

pylidar.toolbox.grdfilters.pmf.**elevationDiffTreshold**(*c*, *wk*, *wk1*, *s*, *dh0*, *dhmax*)
Function to determine the elevation difference threshold based on window size (wk) c is the bin size is metres. Default values for site slope (s), initial elevation differents (dh0), and maximum elevation difference (dhmax). These will change based on environment.

- genindex
- modindex
- search

4.4.2 Interpolation

Functions which can be used to perform interpolation of point data

exception pylidar.toolbox.interpolation.**InterpolationError**
Interpolation Error

pylidar.toolbox.interpolation.**interpGrid**(*xVals*, *yVals*, *zVals*, *gridCoords*, *method='pynn'*)
A function to interpolate values to a regular gridCoords given an irregular set of input data points

- xVals is an array of X coordinates for known points
- yVals is an array of Y coordinates for known points
- zVals is an array of values associated with the X,Y points to be interpolated

- gridCoords is a 2D array with the X,Y values for each 'pixel' in the grid; use data.info.getBlockCoordArrays()
- method is a string specifying the method of interpolation to use, 'nearest', 'linear', 'cubic', 'cgalnn', 'pynn', 'pylinear'

returns grid of float64 values with the same dimensions are the gridCoords with interpolated Z values.

pylidar.toolbox.interpolation.**interpPoints**(*xVals*, *yVals*, *zVals*, *ptCoords*, *method='pynn'*)
A function to interpolate values to a set of points given an irregular set of input data points

- xVals is an array of X coordinates for known points
- yVals is an array of Y coordinates for known points
- zVals is an array of values associated with the X,Y points to be interpolated
- ptCoords is a 2D array with pairs of xy values (shape: N*2)
- method is a string specifying the method of interpolation to use, 'pynn' is the only currently implemented one.

returns 1d array with Z values.

- genindex
- modindex
- search

4.4.3 Visualisation

Functions which can be used to help with the visualisation of the point clouds

exception pylidar.toolbox.visualisation.**VisualisationError**
A Visualisation error has occured

pylidar.toolbox.visualisation.**colourByClass**(*classArr*, *colourDict=None*)
A function which returns RGB and point size arrays given an input array of numerical classes. Where colourDict has been specified then the default colours (specified in getClassColoursDict()) can be overiden.

pylidar.toolbox.visualisation.**createRGB4Param**(*data*, *stretch='linear'*, *colourMap='Spectral'*)
A function to take a data column (e.g., intensity) and colour into a set of rgb arrays for visualisation. colourMap is a matplotlib colour map (see http://wiki.scipy.org/Cookbook/Matplotlib/Show_colormaps) for colouring the input data. stretch options are 'linear' or 'stddev'

pylidar.toolbox.visualisation.**displayPointCloud**(*x*, *y*, *z*, *r*, *g*, *b*, *s*)
Display the point cloud in 3D where arrays (all of the same length) are provided for the X, Y, Z position of the points and then the RGB (range 0-1) values to colour the points and then the point sizes (s).

X and Y values are centred around 0,0 and then X, Y, Z values are rescale before display

pylidar.toolbox.visualisation.**getClassColoursDict**()

pylidar.toolbox.visualisation.**rescaleRGB**(*r*, *g*, *b*, *bit8=True*)
A function which rescales the RGB arrays to a range of 0-1. Where bit8 is True the arrays will just be divided by 255 otherwise a min-max rescaling will be applied independently to each array.

- genindex
- modindex
- search

4.4.4 Spatial Processing Utilities

Utility functions for assisting with Spatial Processing in Pylidar.

class pylidar.toolbox.spatial.**ImageWriter**(*filename*, *numBands=1*, *gdalDataType=None*, *driverName=<Mock name='mock.applier.DEFAULTDRIVERNAME' id='139696485622288'>*, *driverOptions=<Mock name='mock.applier.DEFAULTCREATIONOPTIONS' id='139696484385552'>*, *tlx=0.0*, *tly=0.0*, *binSize=0.0*, *epsg=None*, *nullVal=None*, *ncols=None*, *nrows=None*, *calcStats=True*)

Class that handles writing out image data with GDAL

close()

Close and flush the dataset, plus calculate stats

createDataset()

Internal method. Assumes self.gdalDataType, self.ncols and self.nrows is set.

setLayer(*array*, *layerNum=1*)

Set a layer in the file as a 2d array

exception pylidar.toolbox.spatial.**SpatialException**

An exception that is raised by this module

pylidar.toolbox.spatial.**getBlockCoordArrays**(*xMin*, *yMax*, *nCols*, *nRows*, *binSize*)

Return a tuple of the world coordinates for every pixel in the current block. Each array has the same shape as the current block. Return value is a tuple:

```
(xBlock, yBlock)
```

where the values in xBlock are the X coordinates of the centre of each pixel, and similarly for yBlock.

The coordinates returned are for the pixel centres. This is slightly inconsistent with usual GDAL usage, but more likely to be what one wants.

pylidar.toolbox.spatial.**getGridInfoFromData**(*xdata*, *ydata*, *binSize*)

Given an array of X coords, an array of Y coords, plus a binSize return a tuple of (xMin, yMax, ncols, nrows) for doing operations on a grid

pylidar.toolbox.spatial.**getGridInfoFromHeader**(*header*, *binSize*, *footprint=<Mock name='mock.imageio.UNION' id='139696478360976'>*)

Given a Lidar file header (or a list of headers - maximum extent will be calculated) plus a binSize return a tuple of (xMin, yMax, ncols, nrows) for doing operations on a grid Specify lidarprocessor.UNION or lidarprocessor.INTERSECTION to determine how multiple headers are combined.

pylidar.toolbox.spatial.**readImageLayer**(*inFile*, *layerNum=1*)

Read a layer from a GDAL supported dataset and return it as a 2d numpy array along with georeferencing information.

Returns a tuple with (data, xMin, yMax, binSize)

pylidar.toolbox.spatial.**readLidarPoints**(*filename*, *classification=None*, *boundingbox=None*, *colNames=['X', 'Y', 'Z']*)

Read the requested columns for the points in the given file (or files if filename is a list), in a memory-efficient manner. Uses pylidar to read only a block of points at a time, and select out just the desired columns. When the input file is a .las file, this saves quite a lot of memory, in comparison to reading in all points at once, since all columns for all points have to be read in at the same time.

Optionally filter by CLASSIFICATION column with a value from the generic.CLASSIFICATION_* constants.

If boundingbox is given, it is a tuple of (xmin, xmax, ymin, ymax) and only points within this box are included.

Return a single recarray with only the selected columns, and only the selected points.

pylidar.toolbox.spatial.**selectColumns**(*data*, *otherargs*)
Called from pylidar's doProcessing.

Read the next block of lidar points, select out the requested columns. If requested, filter to ground only. If requested, restrict to the given bounding box.

pylidar.toolbox.spatial.**xyToRowCol**(*x*, *y*, *xMin*, *yMax*, *pixSize*)
For the given pixel size and xMin, yMax, convert the given arrays of x and y into arrays of row and column in a regular grid across the tile.

xMin and yMax represent the top/left corner of the top/left pixel of the image

Assumes that the bounds of the grid are to be fixed on integer coordinates.

Return a tuple of arrays (row, col)

- genindex
- modindex
- search

4.4.5 Canopy Products

canopy module. Algorithms for creating canopy metrics

Vertical Foliage Profiles

Voxel Traversal and Vertical Cover Profiles

Functions for voxelization of TLS scans (Hancock et al., 2016)

pylidar.toolbox.canopy.voxel_hancock2016.**classify_voxels**(*hits*, *miss*, *occl*, *classification*, *ground=None*)
Classification of voxels

Class Value Hits Misses Occluded Observed 5 >0 >=0 >=0 Empty 4 =0 >0 >=0 Hidden 3 =0 =0 >0 Unobserved 2 =0 =0 =0 Ground 1

pylidar.toolbox.canopy.voxel_hancock2016.**runVoxelization**(*data*, *otherargs*)
Voxelization function for the lidar processor

pylidar.toolbox.canopy.voxel_hancock2016.**run_voxel_hancock2016**(*infiles*, *controls*, *otherargs*, *outfiles*)
Main function for VOXEL_HANCOCK2016

The gap fraction of each voxel is the ratio of the number of beams that reach the voxel to the number that could have passed through.

- genindex
- modindex
- search
- genindex

- modindex
- search

4.5 Indexing

indexing module. A algorithms for creating spatial indexed data

4.5.1 Grid Indexing

Deals with creating a grid spatial index

pylidar.toolbox.indexing.gridindex.**BLOCKSIZE_N_BLOCKS = 2**
Types of spatial indices. Copied from spdv4.

pylidar.toolbox.indexing.gridindex.**INDEX_SCAN = 5**
Types of pulse indexing methods. Copied from spdv4.

pylidar.toolbox.indexing.gridindex.**classifyFunc**(*data*, *otherArgs*)
Called by lidarprocessor. Looks at the input data and splits into the appropriate output files.

pylidar.toolbox.indexing.gridindex.**copyScaling**(*input*, *output*)
Copy the known scaling required fields accross.

Internal method. Called from classifyFunc.

pylidar.toolbox.indexing.gridindex.**createGridSpatialIndex**(*infile*, *outfile*, *binSize=1.0*, *blockSize=None*, *tempDir=None*, *extent=None*, *indexType=1*, *pulseIndexMethod=0*, *wkt=None*)
Creates a grid spatially indexed file from a non spatial input file. Currently only supports creation of a SPD V4 file.

Creates a tempfile for every block (using blockSize) and them merges them into the output building a spatial index as it goes. If blockSize isn't set then it is picked using BLOCKSIZE_N_BLOCKS. binSize is the size of the bins to create the spatial index. if tempDir is none a temporary directory will be created with tempfile.mkdtemp and removed at the end of processing. extent is an Extent object specifying the extent to work within. indexType is one of the INDEX_* constants. pulseIndexMethod is one of the PULSE_INDEX_* constants. wkt is the projection to use for the output. Copied from the input if not supplied. nPulsesPerChunkMerge is the number of pulses to process at a time when merging.

pylidar.toolbox.indexing.gridindex.**getDefaultWKT**()
When processing data in sensor or project coordinates we may not have a WKT. However, rios.pixelgrid requires something. For now return the WKT for GDA96/MGA zone 55 until we think of something better.

pylidar.toolbox.indexing.gridindex.**indexAndMerge**(*extentList*, *extent*, *wkt*, *outfile*, *header*)
Internal method to merge all the temporary files into the output spatially indexing as we go.

pylidar.toolbox.indexing.gridindex.**indexPulses**(*pulses*, *points*, *pulseIndexMethod*)
Internal method to assign a point coordinates to the X_IDX and Y_IDX columns based on the user specified pulse_index_method.

pylidar.toolbox.indexing.gridindex.**setScalingForCoordField**(*driver*, *srcfield*, *coordfield*)

Internal method to set the output scaling for range of data.

pylidar.toolbox.indexing.gridindex.**splitFileIntoTiles**(*infiles*, *binSize=1.0*, *blockSize=None*, *tempDir='.'*, *extent=None*, *indexType=1*, *pulseIndexMethod=0*, *footprint=<Mock name='mock.imageio.UNION' id='139696478360976'>*, *outputFormat='SPDV4'*, *buildPulses=False*)

Takes a filename (or list of filenames) and creates a tempfile for every block (using blockSize). If blockSize isn't set then it is picked using BLOCKSIZE_N_BLOCKS. binSize is the size of the bins to create the spatial index. indexType is one of the INDEX_* constants. pulseIndexMethod is one of the PULSE_INDEX_* constants. footprint is one of lidarprocessor.UNION or lidarprocessor.INTERSECTION and is how to combine extents if there is more than one file. outputFormat is either 'SPDV4' or 'LAS'. 'LAS' outputs only supported when input is 'LAS'. buildPulses relevant for 'LAS' and determines whether to build the pulse structure or not.

returns the header of the first input file, the extent used and a list of (fname, extent) tuples that contain the information for each tempfile.

- genindex
- modindex
- search

4.6 Translation

Module for doing translation between lidar formats

4.6.1 Command Line

Entry points for running the command line utils

4.6.2 Common functions

Common data and functions for translation

pylidar.toolbox.translate.translatecommon.**DEFAULT_DTYPE_STR = 'DFLT'**
Code that means: use the default type

pylidar.toolbox.translate.translatecommon.**DEFAULT_SCALING = {0: {'AMPLITUDE_RETURN': [1.0,**
all the default scalings as a dictionary

pylidar.toolbox.translate.translatecommon.**POINT_DEFAULT_SCALING = {'AMPLITUDE_RETURN': [1.0**
default scaling for points

pylidar.toolbox.translate.translatecommon.**PULSE_DEFAULT_SCALING = {'AMPLITUDE_PULSE': [100**
Default scaling for pulses

pylidar.toolbox.translate.translatecommon.**STRING_TO_DTYPE = {'FLOAT32': <type 'numpy.float**
String to numpy dtype dictionary

pylidar.toolbox.translate.translatecommon.**WAVEFORM_DEFAULT_SCALING = {'RANGE_TO_WAVEFORM_S**
default scaling for waveforms

pylidar.toolbox.translate.translatecommon.**addConstCols**(*constCols*, *points*, *pulses*, *waveforms=None*)
Add constant columns to points, pulses or waveforms

constCols is a list of tupes with (type, varname, dtype, value)

pylidar.toolbox.translate.translatecommon.**checkRange**(*expectRange*, *points*, *pulses*, *waveforms=None*)
Checks the expected range against the data that has been passed. Raises an exception if data is outside of range

- expectRange is a list of tuples with (type, varname, min, max).
- points, pulses and waveforms are the arrays to check

pylidar.toolbox.translate.translatecommon.**overRideDefaultScalings**(*scaling*)
Any scalings given on the commandline should over-ride the default behaviours. if scalings is not None then it is assumed to be a list of tuples with (type, varname, type, gain, offset).

Returns a dictionary keyed on lidarprocessor.ARRAY_TYPE_PULSES, lidarprocessor.ARRAY_TYPE_POINTS, or lidarprocessor.ARRAY_TYPE_WAVEFORMS. Each value in this dictionary is in turn a dictionary keyed on the column name in which each value is a tuple with gain, offset and dtype.

pylidar.toolbox.translate.translatecommon.**setOutputNull**(*nullVals*, *output*)
Set the null values.

nullVals should be a list of (type, varname, value) tuples

Designed to be called from inside a lidarprocessor function so output should be an instance of *pylidar.userclasses.LidarData*.

pylidar.toolbox.translate.translatecommon.**setOutputScaling**(*scalingDict*, *output*)
Set the scaling on the output SPD V4 file.

Designed to be called from inside a lidarprocessor function so output should be an instance of *pylidar.userclasses.LidarData*.

scalingDict should be what was returned by overRideDefaultScalings().

4.6.3 LAS to SPDV4

Handles conversion between LAS and SPDV4 formats

pylidar.toolbox.translate.las2spdv4.**transFunc**(*data*, *otherArgs*)
Called from lidarprocessor. Does the actual conversion to SPD V4

pylidar.toolbox.translate.las2spdv4.**translate**(*info*, *infile*, *outfile*, *expectRange=None*, *spatial=None*, *extent=None*, *scaling=None*, *epsg=None*, *binSize=None*, *buildPulses=False*, *pulseIndex=None*, *nullVals=None*, *constCols=None*, *useLASScaling=False*)
Main function which does the work.

- Info is a fileinfo object for the input file.
- infile and outfile are paths to the input and output files respectively.
- expectRange is a list of tuples with (type, varname, min, max).

- **spatial is True or False - dictates whether we are processing spatially or not.** If True then spatial index will be created on the output file on the fly.
- **extent is a tuple of values specifying the extent to work with.** xmin ymin xmax ymax
- scaling is a list of tuples with (type, varname, dtype, gain, offset).
- if epsg is not None should be a EPSG number to use as the coord system
- binSize is the used by the LAS spatial index
- buildPulses dictates whether to attempt to build the pulse structure
- **pulseIndex should be 'FIRST_RETURN' or 'LAST_RETURN' and determines how the** pulses are indexed.
- nullVals is a list of tuples with (type, varname, value)
- constCols is a list of tupes with (type, varname, dtype, value)
- **if useLASScaling is True, then the scaling used in the LAS file** is used for columns. Overrides anything given in 'scaling'

pylidar.toolbox.translate.las2spdv4.**updateScalingWithLASValues**(*scalingDict*, *input*, *pointsArray*)

Updates scalingDict with scalings from input (a LAS file). pointsArray is needed so we know what fields exist in the input.

4.6.4 SPDV3 to SPDV4

Handles conversion between SPDV3 and SPDV4 formats

pylidar.toolbox.translate.spdv32spdv4.**transFunc**(*data*, *otherArgs*)

Called from lidarprocessor. Does the actual conversion to SPD V4

pylidar.toolbox.translate.spdv32spdv4.**translate**(*info*, *infile*, *outfile*, *expectRange=None*, *spatial=False*, *extent=None*, *scaling=None*, *nullVals=None*, *constCols=None*)

Main function which does the work.

- Info is a fileinfo object for the input file.
- infile and outfile are paths to the input and output files respectively.
- expectRange is a list of tuples with (type, varname, min, max).
- **spatial is True or False - dictates whether we are processing spatially or not.** If True then spatial index will be created on the output file on the fly.
- **extent is a tuple of values specifying the extent to work with.** xmin ymin xmax ymax
- scaling is a list of tuples with (type, varname, gain, offset).
- nullVals is a list of tuples with (type, varname, value)
- constCols is a list of tupes with (type, varname, dtype, value)

4.6.5 Riegl to SPDV4

4.6.6 ASCII to SPDV4

Handles conversion between ASCII and SPDV4 formats

pylidar.toolbox.translate.ascii2spdv4.**transFunc**(*data*, *otherArgs*)
Called from lidarprocessor. Does the actual conversion to SPD V4

pylidar.toolbox.translate.ascii2spdv4.**translate**(*info*, *infile*, *outfile*, *colTypes*, *pulseCols=None*, *expectRange=None*, *scaling=None*, *classificationTranslation=None*, *nullVals=None*, *constCols=None*)
Main function which does the work.

- Info is a fileinfo object for the input file.
- infile and outfile are paths to the input and output files respectively.
- expectRange is a list of tuples with (type, varname, min, max).
- scaling is a list of tuples with (type, varname, gain, offset).
- colTypes is a list of name and data type tuples for every column
- pulseCols is a list of strings defining the pulse columns
- **classificationTranslation is a list of tuples specifying how to translate** between the codes within the files and the lidarprocessor.CLASSIFICATION_* ones. First element of tuple is file number, second the lidarprocessor code.
- nullVals is a list of tuples with (type, varname, value)
- constCols is a list of tupes with (type, varname, dtype, value)

4.6.7 SPDV4 to LAS

Handles conversion between SPDV4 and LAS formats

pylidar.toolbox.translate.spdv42las.**setOutputScaling**(*points*, *indata*, *outdata*)
Sets the output scaling for las. Tries to copy scaling accross.

pylidar.toolbox.translate.spdv42las.**transFunc**(*data*)
Called from pylidar. Does the actual conversion to las

pylidar.toolbox.translate.spdv42las.**translate**(*info*, *infile*, *outfile*, *spatial=False*, *extent=None*)
Does the translation between SPD V4 and .las format files.

- Info is a fileinfo object for the input file.
- infile and outfile are paths to the input and output files respectively.
- **spatial is True or False - dictates whether we are processing spatially or not.** If True then spatial index will be created on the output file on the fly.
- **extent is a tuple of values specifying the extent to work with.** xmin ymin xmax ymax

Currently does not take any command line scaling options so LAS scaling will be the same as the SPDV4 input file scaling. Not sure if this is a problem or not...

- genindex

- modindex
- search

CHAPTER 5

Drivers

5.1 basedriver

Generic 'driver' class. To be subclassed by both LiDAR and raster drivers.

Also contains the Extent class which defines the extent to use for reading or writing the current block.

pylidar.basedriver.**CREATE = 2**
access modes passed to driver constructor

class pylidar.basedriver.**Driver**(*fname*, *mode*, *controls*, *userClass*)
Base Driver object to be subclassed be both the LiDAR and raster drivers

close()
Close all open file handles

getPixelGrid()
Return the PixelGridDefn for this file

setExtent(*extent*)
Set the extent for reading or writing

setPixelGrid(*pixGrid*)
Set the PixelGridDefn for the reading or writing we will do

class pylidar.basedriver.**Extent**(*xMin*, *xMax*, *yMin*, *yMax*, *binSize*)
Class that defines the extent in world coords of an area to read or write

class pylidar.basedriver.**FileInfo**(*fname*)
Class that contains information about a file At this stage only subclassed by the lidar drivers

pylidar.basedriver.**READ = 0**
access modes passed to driver constructor

pylidar.basedriver.**UPDATE = 1**
access modes passed to driver constructor

- genindex

- modindex
- search

5.2 gdaldriver

Driver for GDAL supported files

class pylidar.gdaldriver.**GDALDriver**(*fname*, *mode*, *controls*, *userClass*)
This driver supports reading and writing of raster data using GDAL.

close()
Calculate stats etc

getData()
Read a 3d numpy array with data for the current extent

getPixelGrid()
Get the pixel grid for this file

setData(*data*)
Write a 3d numpy array to the image

setExtent(*extent*)
Set the extent for the next read or write. Convert from world coords to file coords.

setPixelGrid(*pixGrid*)
Set the pixel grid to use for this new file

exception pylidar.gdaldriver.**GDALException**
An exception that is raised by this driver

- genindex
- modindex
- search

5.3 generic

Base class for LiDAR format reader/writers

pylidar.lidarformats.generic.**ARRAY_TYPE_POINTS = 0**
For use in userclass.LidarData.translateFieldNames() and LiDARFile.getTranslationDict()

pylidar.lidarformats.generic.**ARRAY_TYPE_PULSES = 1**
For use in userclass.LidarData.translateFieldNames() and LiDARFile.getTranslationDict()

pylidar.lidarformats.generic.**ARRAY_TYPE_WAVEFORMS = 2**
For use in userclass.LidarData.translateFieldNames() and LiDARFile.getTranslationDict()

pylidar.lidarformats.generic.**CLASSIFICATION_BRANCH = 102**
Extended classifications

pylidar.lidarformats.generic.**CLASSIFICATION_BRIDGE = 12**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_BUILDING = 6**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_COLNAME = 'CLASSIFICATION'**
Name of column to treat as classification

pylidar.lidarformats.generic.**CLASSIFICATION_CREATED = 0**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_FOLIAGE = 101**
Extended classifications

pylidar.lidarformats.generic.**CLASSIFICATION_GROUND = 2**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_HIGHPOINT = 8**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_HIGHVEGE = 5**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_INSULATOR = 16**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_LOWPOINT = 7**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_LOWVEGE = 3**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_MEDVEGE = 4**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_RAIL = 10**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_ROAD = 11**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_TRANSTOWER = 15**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_TRUNK = 100**
Extended classifications

pylidar.lidarformats.generic.**CLASSIFICATION_UNCLASSIFIED = 1**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_WATER = 9**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_WIRECOND = 14**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CLASSIFICATION_WIREGUARD = 13**
Classifications from the LAS spec. See LiDARFile.recodeClassification

pylidar.lidarformats.generic.**CREATE = 2**
access modes passed to driver constructor

pylidar.lidarformats.generic.**FIELD_POINTS_RETURN_NUMBER = 1**
'standard' fields that have different names for different formats

pylidar.lidarformats.generic.**FIELD_PULSES_TIMESTAMP = 2**
'standard' fields that have different names for different formats

pylidar.lidarformats.generic.**HEADER_NUMBER_OF_POINTS = 1**
'standard' header fields that have different names for different formats

exception pylidar.lidarformats.generic.**LiDARArrayColumnError**
Unsupported operation on a structured array

class pylidar.lidarformats.generic.**LiDARFile**(*fname*, *mode*, *controls*, *userClass*)
Base class for all LiDAR Format reader/writers.

It is intended that very little work happens until the user actually asks for the data - then read it in. Subsequent calls for the same extent should return cached data.

close()
Write any updated spatial index and close any file handles.

static getDriverName()
Return name of driver - just a short unique name is fine.

getHeader()
Return a dictionary of key/value pairs containing header info

static getHeaderTranslationDict()
Return a dictionary keyed on HEADER_* values (above) that can be used to translate dictionary field names between the formats

getHeaderValue(*name*)
Just extract the one value and return it

getNativeDataType(*colName*, *arrayType*)
Return the native dtype (numpy.int16 etc) that a column is stored as internally after scaling (if any) is applied. Provided so scaling can be adjusted when translating between formats.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

Raises generic.LiDARArrayColumnError if information cannot be found for the column.

getNullValue(*colName*, *arrayType*, *scaled=True*)
Get the 'null' value for the given column.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

By default the returned value is scaled, change this with the 'scaled' parameter.

Raises generic.LiDARArrayColumnError if information cannot be found for the column.

getScaling(*colName*, *arrayType*)
Returns the scaling (gain, offset) for the given column name

arrayType is one of the ARRAY_TYPE_* constants.

Raises generic.LiDARArrayColumnError if no scaling (yet) set for this column.

getScalingColumns(*arrayType*)
Return a list of columns names that will need scaling to be set when creating a new file.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

getTotalNumberPulses()
Returns the total number of pulses in this file. Used for progress.

Raise a LiDARFunctionUnsupported error if driver does not support easily finding the total number of pulses.

static getTranslationDict(*arrayType*)
Return a dictionary keyed on FIELD_* values (above) that can be used to translate field names between the formats arrayType is the type of array that is to be translated (ARRAY_TYPE_*)

For use by the `pylidar.userclases.LidarData.translateFieldNames()` function.

hasSpatialIndex()
Returns True if file has a spatial index defined

readPointsByPulse()
Read a 2d structured masked array containing the points for each pulse.

readPointsForExtent(*colNames=None*)
Read all the points within the given extent as 1d structured array. The names of the fields in this array will be defined by the driver.

colNames can be a name or list of column names to return. By default all columns are returned.

readPointsForExtentByBins(*colNames=None*, *indexByPulse=False*, *returnPulseIndex=False*)
Read all the points within the given extent as a 3d structured masked array to match the block/bins being used.

The extent/binning for the read data can be overriden by passing in a Extent instance.

colNames can be a name or list of column names to return. By default all columns are returned.

Pass indexByPulse=True to bin the points by the locations of the pulses instead of the points.

Pass returnPulseIndex=True to also return a masked 3d array of the indices into the 1d pulse array (as returned by readPulsesForExtent())

readPointsForRange(*colNames=None*)
Reads the points for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readPulsesForExtent(*colNames=None*)
Read all the pulses within the given extent as 1d structured array. The names of the fields in this array will be defined by the driver.

colNames can be a name or list of column names to return. By default all columns are returned.

readPulsesForExtentByBins(*colNames=None*)
Read all the pulses within the given extent as a 3d structured masked array to match the block/bins being used.

The extent/binning for the read data can be overriden by passing in a Extent instance.

colNames can be a name or list of column names to return. By default all columns are returned.

readPulsesForRange(*colNames=None*)
Reads the pulses for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readReceived()
Read the received waveform for all pulses returns a 2d masked array

readTransmitted()
Read the transmitted waveform for all pulses returns a 3d masked array.

readWaveformInfo()
2d structured masked array containing information about the waveforms.

recodeClassification(*array*, *direction*, *colNames=None*)
Recode classification column (if it exists in array) in the specified direction.

If array is not structured and colNames is a string equal to CLASSIFICATION_COLNAME, then the array is treated as the classification column.

setHeader(*newHeaderDict*)
Update all of the header values as a dictionary

setHeaderValue(*name*, *value*)
Just update one value in the header

setNativeDataType(*colName*, *arrayType*, *dtype*)
Set the native dtype (numpy.int16 etc) that a column is stored as internally after scaling (if any) is applied.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

generic.LiDARArrayColumnError is raised if this cannot be set for the column.

The default behaviour is to create new columns in the correct type for the format, or if they are optional, in the same type as the input array.

setNullValue(*colName*, *arrayType*, *value*, *scaled=True*)
Set the 'null' value for the given column.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

By default the value is treated as the scaled value, but this can be changed with the 'scaled' parameter.

generic.LiDARArrayColumnError is raised if this cannot be set for the column.

setPulseRange(*pulseRange*)
Sets the PulseRange object to use for non spatial reads/writes.

Return False if outside the range of data.

setScaling(*colName*, *arrayType*, *gain*, *offset*)
Set the scaling for the given column name

arrayType is one of the ARRAY_TYPE_* constants

static subsetColumns(*array*, *colNames*)
Internal method. Subsets the given column names from the array and returns it. colNames can be either a string or a sequence of column names. If None the input array is returned.

writeData(*pulses=None*, *points=None*, *transmitted=None*, *received=None*, *waveformInfo=None*)
Write data to file. pulses to be 1d structured array. points to be 2d points-by-pulses format. waveformInfo, transmitted and received to be 2d by-pulses format.

Pass None if no data to be written or data unchanged (for update).

exception pylidar.lidarformats.generic.**LiDARFileException**
Base class for LiDAR format reader/writers

class pylidar.lidarformats.generic.**LiDARFileInfo**(*fname*)
Info for a Lidar file

static getDriverName()
Return name of driver - just a short unique name is fine. should match the *pylidar.lidarformats.generic.LiDARFile.getDriverName()* call for the same format.

static getHeaderTranslationDict()
Return a dictionary keyed on HEADER_* values (above) that can be used to translate dictionary field names between the formats

exception pylidar.lidarformats.generic.**LiDARFormatDriverNotFound**
None of the drivers can open the file

exception pylidar.lidarformats.generic.**LiDARFormatNotUnderstood**
Raised when driver cannot open file

exception pylidar.lidarformats.generic.**LiDARFunctionUnsupported**
Function unsupported by LiDAR driver

exception pylidar.lidarformats.generic.**LiDARInvalidData**
Something is wrong with the data read or given

exception pylidar.lidarformats.generic.**LiDARInvalidSetting**
Setting does not make sense

exception pylidar.lidarformats.generic.**LiDARNonSpatialProcessing**
Functionality not available when not processing spatially

exception pylidar.lidarformats.generic.**LiDARPulseIndexUnsupported**
The specified pulse index method is currently unsupported

exception pylidar.lidarformats.generic.**LiDARScalingError**
scaled data is outside the bounds of the data type

exception pylidar.lidarformats.generic.**LiDARSpatialIndexNotAvailable**
The specified spatial index not available for this file

exception pylidar.lidarformats.generic.**LiDARWritingNotSupported**
driver does not support writing

pylidar.lidarformats.generic.**MESSAGE_DEBUG = 2**
to be passed to message handler function controls.messageHandler

pylidar.lidarformats.generic.**MESSAGE_INFORMATION = 1**
to be passed to message handler function controls.messageHandler

pylidar.lidarformats.generic.**MESSAGE_WARNING = 0**
to be passed to message handler function controls.messageHandler

class pylidar.lidarformats.generic.**PulseRange**(*startPulse*, *endPulse*)
Class for setting the range of pulses to read/write for non spatial mode. Note: range does not include endPulse

pylidar.lidarformats.generic.**READ = 0**
access modes passed to driver constructor

pylidar.lidarformats.generic.**RECODE_TO_DRIVER = 0**
Codes to pass to LiDARFile.recodeClassification

pylidar.lidarformats.generic.**RECODE_TO_LAS = 1**
Codes to pass to LiDARFile.recodeClassification

pylidar.lidarformats.generic.**UPDATE = 1**
access modes passed to driver constructor

pylidar.lidarformats.generic.**getLidarFileInfo**(*fname*, *verbose=False*)
Returns an instance of a LiDAR format info class. Or raises an exception if none found for the file.

pylidar.lidarformats.generic.**getReaderForLiDARFile**(*fname*, *mode*, *controls*, *userClass*, *verbose=False*)
Returns an instance of a LiDAR format reader/writer or raises an exception if none found for the file.

pylidar.lidarformats.generic.**getWriterForLiDARFormat**(*driverName*, *fname*, *mode*, *controls*, *userClass*)
Given a driverName returns an instance of the given driver class Raises LiDARFormatDriverNotFound if not found

- genindex
- modindex
- search

5.4 spdv3

SPD V3 format driver and support functions

pylidar.lidarformats.spdv3.**HEADER_ARRAY_FIELDS = ('BANDWIDTHS', 'WAVELENGTHS')**
header fields that are actually arrays

pylidar.lidarformats.spdv3.**HEADER_FIELDS = {'AZIMUTH_MAX': <type 'numpy.float64'>, 'AZIMUTH**
Header fields and their types

pylidar.lidarformats.spdv3.**HEADER_TRANSLATION_DICT = {1: 'NUMBER_OF_POINTS'}**
Translation of header field names

pylidar.lidarformats.spdv3.**POINTS_HEADER_UPDATE_DICT = {'RANGE': ('RANGE_MIN', 'RANGE_MAX'**
for updating the header

pylidar.lidarformats.spdv3.**POINT_DTYPE = dtype([('RETURN_ID', 'u1'), ('GPS_TIME', '<f8'),**
so we can check the user has passed in expected array type

pylidar.lidarformats.spdv3.**PULSES_HEADER_UPDATE_DICT = {'AZIMUTH': ('AZIMUTH_MIN', 'AZIMUTH**
for updating the header

pylidar.lidarformats.spdv3.**PULSE_DTYPE = dtype([('GPS_TIME', '<u8'), ('PULSE_ID', '<u8'),**
so we can check the user has passed in expected array type

class pylidar.lidarformats.spdv3.**SPDV3File**(*fname*, *mode*, *controls*, *userClass*)
Class to support reading and writing of SPD Version 3.x files.

Uses h5py to handle access to the underlying HDF5 file.

close()
Write out the spatial index, header and close file handle.

static convertHeaderToDictionary(*header*)
Static method to convert the header returned by h5py into a normal dictionary

static getDriverName()
Name of this driver

getHeader()
Return our cached dictionary

static getHeaderTranslationDict()
Return dictionary with non-standard header names

getHeaderValue(*name*)
Just extract the one value and return it

getNativeDataType(*colName*, *arrayType*)
Return the native dtype (numpy.int16 etc)that a column is stored as internally. Provided so scaling can be adjusted when translating between formats.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

getPixelGrid()
Return the pixel grid of this spatial index.

getTotalNumberPulses()
Return the total number of pulses

static getTranslationDict(*arrayType*)
Translation dictionary between formats

hasSpatialIndex()
Return True if we have a spatial index.

preparePointsForWriting(*points*, *pulses*)
Called from writeData(). Massages what the user has passed into something we can write back to the file.

preparePulsesForWriting(*pulses*)
Called from writeData(). Massages what the user has passed into something we can write back to the file.

prepareReceivedForWriting(*received*, *waveformInfo*)
Called from writeData(). Massages what the user has passed into something we can write back to the file.

prepareTransmittedForWriting(*transmitted*, *waveformInfo*)
Called from writeData(). Massages what the user has passed into something we can write back to the file.

readPointsByPulse(*colNames=None*)
Return a 2d masked structured array of point that matches the pulses.

readPointsForExtent(*colNames=None*)
Read out the points for the given extent as a 1d structured array.

readPointsForExtentByBins(*extent=None*, *colNames=None*, *indexByPulse=False*, *returnPulseIndex=False*)
Return the points as a 3d structured masked array.

Note that because the spatial index on a SPDV3 file is on pulses this may miss points that are attached to pulses outside the current extent. If this is a problem then select an overlap large enough.

Pass indexByPulse=True to bin the points by the locations of the pulses (using X_IDX and Y_IDX rather than the locations of the points)

Pass returnPulseIndex=True to also return a masked 3d array of the indices into the 1d pulse array (as returned by readPulsesForExtent())

readPointsForRange(*colNames=None*)
Read all the points for the specified range of pulses

readPulsesForExtent(*colNames=None*)
Return the pulses for the given extent as a 1d structured array

readPulsesForExtentByBins(*extent=None*, *colNames=None*)
Return the pulses as a 3d structured masked array.

readPulsesForRange(*colNames=None*)
Read the specified range of pulses

readReceived()
Return the 3d masked integer array of received for each of the current pulses. SPDV3 only has 1 transmitted per pulse so the second axis is empty. First axis is waveform bin and last is pulse.

readTransmitted()
Return the 3d masked integer array of transmitted for each of the current pulses. SPDV3 only has 1 transmitted per pulse so the second axis is empty. First axis is waveform bin and last is pulse.

readWaveformInfo()
Return 2d masked array of information about the waveforms.

setExtent(*extent*)
Set the extent to use for the ForExtent() functions.

setHeader(*newHeaderDict*)
Update our cached dictionary

setHeaderValue(*name*, *value*)
Just update one value in the header

setPixelGrid(*pixGrid*)
Set the pixel grid on creation or update

setPulseRange(*pulseRange*)
Set the range of pulses to read

updateHeaderFromData(*points*, *pulses*)
Given some data, updates the _MIN, _MAX etc

writeData(*pulses=None*, *points=None*, *transmitted=None*, *received=None*, *waveformInfo=None*)
Write all the updated data. Pass None for data that do not need to be updated. It is assumed that each parameter has been read by the reading functions

class pylidar.lidarformats.spdv3.**SPDV3FileInfo**(*fname*)
Class that gets information about a SPDV3 file and makes it available as fields.

static getDriverName()
Name of this driver

static getHeaderTranslationDict()
Return dictionary with non-standard header names

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_ALLCLASSES = 100**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_ALLCLASSES_TOP = 101**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_BRANCH = 11**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_BUILDING = 7**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_CREATED = 2**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_FOLIAGE = 10**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_GROUND = 3**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_HIGHVEGE = 6**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_KEYGRDPTS = 105**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_LOWVEGE = 4**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_MEDVEGE = 5**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_NOTGROUND = 104**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_TRUNK = 9**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_UNCLASSIFIED = 1**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_UNDEFINED = 0**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_VEGE = 103**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_VEGETOP = 102**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_WALL = 12**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_CLASSIFICATION_WATER = 8**
classification codes

pylidar.lidarformats.spdv3.**SPDV3_INDEX_CARTESIAN = 1**
types of indexing in the file

pylidar.lidarformats.spdv3.**SPDV3_INDEX_CYLINDRICAL = 3**
types of indexing in the file

pylidar.lidarformats.spdv3.**SPDV3_INDEX_POLAR = 4**
types of indexing in the file

pylidar.lidarformats.spdv3.**SPDV3_INDEX_SCAN = 5**
types of indexing in the file

pylidar.lidarformats.spdv3.**SPDV3_INDEX_SPHERICAL = 2**
types of indexing in the file

pylidar.lidarformats.spdv3.**SPDV3_SI_COUNT_DTYPE**
alias of `numpy.uint32`

pylidar.lidarformats.spdv3.**SPDV3_SI_INDEX_DTYPE**
alias of `numpy.uint64`

- genindex
- modindex
- search

5.5 spdv4

SPD V4 format driver and support functions

5.5.1 Write Driver Options

These are contained in the WRITESUPPORTEDOPTIONS module level variable.

Name	Use
SCALING_BUT_NO_DATA_WARNING	Warn when scaling set for a column that doesn't get created. Defaults to True
HDF5_CHUNK_SIZE	Set the HDF5 chunk size when creating columns. Defaults to 250.

pylidar.lidarformats.spdv4.**GAIN_NAME = 'GAIN'**
For storing in hdf5 attributes

pylidar.lidarformats.spdv4.**HEADER_ARRAY_FIELDS = ('BANDWIDTHS', 'WAVELENGTHS', 'VERSION_SPI**
fields in the header that are actually arrays

pylidar.lidarformats.spdv4.**HEADER_ESSENTIAL_FIELDS = ('SPATIAL_REFERENCE', 'VERSION_DATA')**
VERSION_SPD always created by pylidar

pylidar.lidarformats.spdv4.**HEADER_FIELDS = {'AZIMUTH_MAX': <type 'numpy.float64'>, 'AZIMUTH**
Header fields have defined type in SPDV4

pylidar.lidarformats.spdv4.**HEADER_TRANSLATION_DICT = {1: 'NUMBER_OF_POINTS'}**
Translation of header field names

pylidar.lidarformats.spdv4.**NULL_NAME = 'NULL'**
For storing in hdf5 attributes

pylidar.lidarformats.spdv4.**OFFSET_NAME = 'OFFSET'**
For storing in hdf5 attributes

pylidar.lidarformats.spdv4.**POINTS_ESSENTIAL_FIELDS = ('X', 'Y', 'Z', 'CLASSIFICATION')**
RETURN_NUMBER always created by pylidar

pylidar.lidarformats.spdv4.**POINT_FIELDS = {'AMPLITUDE_RETURN': <type 'numpy.uint16'>, 'BLUE**
Thes fields have defined type

pylidar.lidarformats.spdv4.**POINT_SCALED_FIELDS = ('X', 'Y', 'Z', 'HEIGHT', 'RANGE', 'INTENS**
need scaling applied

pylidar.lidarformats.spdv4.**PULSES_ESSENTIAL_FIELDS = ()**
Note: PULSE_ID, NUMBER_OF_RETURNS and PTS_START_IDX always created by pylidar

pylidar.lidarformats.spdv4.**PULSE_FIELDS = {'AMPLITUDE_PULSE': <type 'numpy.uint16'>, 'AZIMU**
Thes fields have defined type

pylidar.lidarformats.spdv4.**PULSE_SCALED_FIELDS = ('AZIMUTH', 'ZENITH', 'X_IDX', 'Y_IDX', 'X**
need scaling applied

pylidar.lidarformats.spdv4.**READSUPPORTEDOPTIONS = ()**
driver options

pylidar.lidarformats.spdv4.**RECEIVED_DTYPE**
alias of `numpy.uint32`

class pylidar.lidarformats.spdv4.**SPDV4File**(*fname*, *mode*, *controls*, *userClass*)
Class to support reading and writing of SPD Version 4.x files.

Uses h5py to handle access to the underlying HDF5 file.

close()
Close all open file handles

createDataColumn(*groupHandle*, *name*, *data*)
Creates a new data column under groupHandle with the given name with standard HDF5 params.

The type is the same as the numpy array data and data is written to the column

sets the chunk size to self.hdf5ChunkSize which can be overridden in the driver options.

static getDriverName()
Name of this driver

getHeader()
Return our attributes on the file

static getHeaderTranslationDict()
Return dictionary with non-standard header names

getHeaderValue(*name*)
Just extract the one value and return it

getNativeDataType(*colName*, *arrayType*)
Return the native dtype (numpy.int16 etc)that a column is stored as internally after scaling is applied. Provided so scaling can be adjusted when translating between formats.

Note that for 'non essential' columns this will depend on the data type that the column was to begin with.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

getNullValue(*colName*, *arrayType*, *scaled=True*)
Get the 'null' value for the given column.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

getPixelGrid()
Return the PixelGridDefn for this file

getScaling(*colName*, *arrayType*)
Returns the scaling (gain, offset) for the given column name reads from our cache since only written to file on close

Raises generic.LiDARArrayColumnError if no scaling (yet) set for this column.

getScalingColumns(*arrayType*)
arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

getTotalNumberPulses()
Return the total number of pulses

static getTranslationDict(*arrayType*)
Translation dictionary between formats

hasSpatialIndex()
Return True if we have a spatial index.

prepareDataForWriting(*data*, *name*, *arrayType*)
Prepares data for writing to a field.

arrayType is one of the ARRAY_TYPE values from .generic.

Does unscaling if possible unless name ends with '_U'. Raises exception if column needs to have scale and offset defined, but aren't.

Returns the data to write, plus the 'hdfname' which is the field name the data should be written to. This has the '_U' removed.

preparePointsForWriting(*points*, *mask*)
Called from writeData(). Massages what the user has passed into something we can write back to the file.

preparePulsesForWriting(*pulses*)
Called from writeData(). Massages what the user has passed into something we can write back to the file.

prepareReceivedForWriting(*received*, *waveformInfo*)
Called from writeData(). Massages what the user has passed into something we can write back to the file.

prepareTransmittedForWriting(*transmitted*, *waveformInfo*)
Called from writeData(). Massages what the user has passed into something we can write back to the file.

prepareWaveformInfoForWriting(*waveformInfo*)
Flattens the waveformInfo back out so it can be written

static readFieldAndUnScale(*handle*, *name*, *selection*, *unScaled=False*)
Given a h5py handle, field name and selection does any unscaling if asked (unScaled=False).

static readFieldsAndUnScale(*handle*, *colNames*, *selection*)
Given a list of column names returns a structured array of the data. If colNames is a string, a single unstructred array will be returned. It will work out of any of the column names end with '_U' and deal with them appropriately. selection should be a h5space.H5Space.

readPointsByPulse(*colNames=None*)
Return a 2d masked structured array of point that matches the pulses.

readPointsForExtent(*colNames=None*)
Read all the points within the given extent as 1d structured array.

colNames can be a name or list of column names to return. By default all columns are returned.

readPointsForExtentByBins(*extent=None*, *colNames=None*, *indexByPulse=False*, *returnPulseIndex=False*)
Return the points as a 3d structured masked array.

Note that because the spatial index on a SPDV4 file currently is on pulses this may miss points that are attached to pulses outside the current extent. If this is a problem then select an overlap large enough.

Pass indexByPulse=True to bin the points by the locations of the pulses (using X_IDX and Y_IDX rather than the locations of the points) This is the default for non-cartesian indices.

Pass returnPulseIndex=True to also return a masked 3d array of the indices into the 1d pulse array (as returned by readPulsesForExtent())

readPointsForRange(*colNames=None*)
Read all the points for the specified range of pulses

readPulsesForExtent(*colNames=None*)
Read all the pulses within the given extent as 1d structured array.

colNames can be a name or list of column names to return. By default all columns are returned.

readPulsesForExtentByBins(*extent=None*, *colNames=None*)
Return the pulses as a 3d structured masked array.

readPulsesForRange(*colNames=None*)
Read the specified range of pulses

readReceived()
Return the 3d masked integer array of received for each of the current pulses. First axis is the waveform bin. Second axis is waveform number and last is pulse.

readTransmitted()
Return the 3d masked integer array of transmitted for each of the current pulses. First axis is the waveform bin. Second axis is waveform number and last is pulse.

readWaveformInfo()
Return 2d masked array of information about the waveforms.

setExtent(*extent*)
Set the extent for reading or writing

setHeader(*newHeaderDict*)
Update our cached dictionary

setHeaderValue(*name*, *value*)
Just update one value in the header

setNativeDataType(*colName*, *arrayType*, *dtype*)
Set the native dtype (numpy.int16 etc)that a column is stored as internally after scaling (if any) is applied.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

generic.LiDARArrayColumnError is raised if this cannot be set for the format.

The default behaviour is to create new columns in the correct type for the format, or if they are optional, in the same type as the input array.

setNullValue(*colName*, *arrayType*, *value*, *scaled=True*)
Sets the 'null' value for the given column.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

We don't write anything out at this stage as the scaling, if needed, mightn't be set at this stage. The work is done when the file is closed.

setPixelGrid(*pixGrid*)
Set the PixelGridDefn for the reading or writing

setPulseRange(*pulseRange*)
Set the range of pulses to read

setScaling(*colName*, *arrayType*, *gain*, *offset*)
Set the scaling for the given column name

updateHeaderFromData(*points*, *pulses*, *waveformInfo*)
Given some data, updates the _MIN, _MAX etc

writeData(*pulses=None*, *points=None*, *transmitted=None*, *received=None*, *waveformInfo=None*)
Write all the updated data. Pass None for data that do not need to be updated. It is assumed that each parameter has been read by the reading functions

writeStructuredArray(*hdfHandle*, *structArray*, *generatedColumns*, *arrayType*)
Writes a structured array as named datasets under hdfHandle. Also writes columns in dictionary generatedColumns to the same place.

Only use for file creation.

class pylidar.lidarformats.spdv4.**SPDV4FileInfo**(*fname*)
Class that gets information about a SPDV4 file and makes it available as fields.

static getDriverName()
Name of this driver

static getHeaderTranslationDict()
Return dictionary with non-standard header names

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_BRANCH = 11**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_BUILDING = 7**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_CREATED = 2**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_FOLIAGE = 10**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_GROUND = 3**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_HIGHVEGE = 6**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_LOWVEGE = 4**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_MEDVEGE = 5**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_RAIL = 13**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_TRUNK = 9**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_UNCLASSIFIED = 1**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_UNDEFINED = 0**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_WALL = 12**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_CLASSIFICATION_WATER = 8**
classification codes

pylidar.lidarformats.spdv4.**SPDV4_INDEXTYPE_SIMPLEGRID = 0**
types of spatial indices

pylidar.lidarformats.spdv4.**SPDV4_INDEX_CARTESIAN = 1**
types of indexing in the file

pylidar.lidarformats.spdv4.**SPDV4_INDEX_CYLINDRICAL = 3**
types of indexing in the file

pylidar.lidarformats.spdv4.**SPDV4_INDEX_POLAR = 4**
types of indexing in the file

pylidar.lidarformats.spdv4.**SPDV4_INDEX_SCAN = 5**
types of indexing in the file

pylidar.lidarformats.spdv4.**SPDV4_INDEX_SPHERICAL = 2**
types of indexing in the file

pylidar.lidarformats.spdv4.**SPDV4_POINT_FLAGS_IGNORE = 1**
flags for POINT_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_POINT_FLAGS_KEY_POINT = 8**
flags for POINT_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_POINT_FLAGS_OVERLAP = 2**
flags for POINT_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_POINT_FLAGS_SYNTHETIC = 4**
flags for POINT_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_POINT_FLAGS_WAVEFORM = 16**
flags for POINT_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_PULSE_FLAGS_IGNORE = 1**
flags for PULSE_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_PULSE_FLAGS_OVERLAP = 2**
flags for PULSE_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_PULSE_FLAGS_SCANLINE_DIRECTION = 4**
flags for PULSE_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_PULSE_FLAGS_SCANLINE_EDGE = 8**
flags for PULSE_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_PULSE_INDEX_END_WAVEFORM = 3**
pulse indexing methods

pylidar.lidarformats.spdv4.**SPDV4_PULSE_INDEX_FIRST_RETURN = 0**
pulse indexing methods

pylidar.lidarformats.spdv4.**SPDV4_PULSE_INDEX_GROUND = 6**
pulse indexing methods

pylidar.lidarformats.spdv4.**SPDV4_PULSE_INDEX_LAST_RETURN = 1**
pulse indexing methods

pylidar.lidarformats.spdv4.**SPDV4_PULSE_INDEX_MAX_INTENSITY = 5**
pulse indexing methods

pylidar.lidarformats.spdv4.**SPDV4_PULSE_INDEX_ORIGIN = 4**
pulse indexing methods

pylidar.lidarformats.spdv4.**SPDV4_PULSE_INDEX_START_WAVEFORM = 2**
pulse indexing methods

pylidar.lidarformats.spdv4.**SPDV4_PULSE_INDEX_ZPLANE = 7**
pulse indexing methods

pylidar.lidarformats.spdv4.**SPDV4_SIMPLEGRID_COUNT_DTYPE**
alias of numpy.uint32

pylidar.lidarformats.spdv4.**SPDV4_SIMPLEGRID_INDEX_DTYPE**
alias of numpy.uint32

pylidar.lidarformats.spdv4.**SPDV4_VERSION_MAJOR = 4**
version - major

pylidar.lidarformats.spdv4.**SPDV4_VERSION_MINOR = 0**
version - minor

pylidar.lidarformats.spdv4.**SPDV4_WAVEFORM_FLAGS_BASELINE_FIXED = 4**
flags for WAVEFORM_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_WAVEFORM_FLAGS_IGNORE = 1**
flags for WAVEFORM_FLAGS

pylidar.lidarformats.spdv4.**SPDV4_WAVEFORM_FLAGS_SATURATION_FIXED = 2**
flags for WAVEFORM_FLAGS

pylidar.lidarformats.spdv4.**TRANSMITTED_DTYPE**
alias of numpy.uint32

pylidar.lidarformats.spdv4.**WAVEFORMS_HEADER_UPDATE_DICT = {'RANGE': ('RANGE_MIN', 'RANGE_MA**
for updating the header

pylidar.lidarformats.spdv4.**WAVEFORM_FIELDS = {'CHANNEL': <type 'numpy.uint8'>, 'NUMBER_OF_W**
Thes fields have defined type

pylidar.lidarformats.spdv4.**WAVEFORM_SCALED_FIELDS = ('RANGE_TO_WAVEFORM_START',)**
need scaling applied

pylidar.lidarformats.spdv4.**WRITESUPPORTEDOPTIONS = ('SCALING_BUT_NO_DATA_WARNING', 'HDF5_CH**
driver options

- genindex
- modindex
- search

5.6 las

Driver for .las files. Uses lastools (https://github.com/LAStools/LAStools).

5.6.1 Read Driver Options

These are contained in the READSUPPORTEDOPTIONS module level variable.

Name	Use
BUILD_PULSES	A boolean. If set to true (the default) pylidar attempts to build pulses assuming that data is in time sequential order. If false, a 'fake' pulse is created for each point.
BIN_SIZE	A number. For files with a spatial index present this is the bin size that the File be presented at. Las indexes can use any arbitary bin size it seems, but works to specific ones which can be set with this option for this file. An error will be raised if a spatial read is attempted and this hasn't been set.
PULSE_INDEX	FIRST_RETURN or LAST_RETURN Dictates which point will be used to set the X_IDX and Y_IDX pulse fields

5.6.2 Write Driver Options

These are contained in the WRITESUPPORTEDOPTIONS module level variable.

Name	Use
FORMAT_VERSION	LAS point format. Defaults to 1. Not sure what it means.
RECORD_LENGTH	LAS record length. Defaults to 28. Not sure what it means.
WAVEFORM_DESCR	Data returned from getWavePacketDescriptions() which does an initial run through the data to get the unique waveform info for writing to the LAS header. No output waveforms are written if this is not provided.

Note that for writing, the extension currently controls the format witten:

Extension	Format
.las	LAS
.laz	LAZ (compressed las)
.bin	terrasolid
.qi	QFIT
.wrl	VRML
other	ASCII

pylidar.lidarformats.las.**DEFAULT_HEADER = {'FILE_CREATION_DAY': 231, 'FILE_CREATION_YEAR':**
for new files

pylidar.lidarformats.las.**FIRST_RETURN = None**
for indexing pulses

pylidar.lidarformats.las.**HEADER_TRANSLATION_DICT = {1: 'NUMBER_OF_POINT_RECORDS'}**
Non standard header names

pylidar.lidarformats.las.**LAST_RETURN = None**
for indexing pulses

pylidar.lidarformats.las.**LAS_SIMPLEGRID_COUNT_DTYPE**
alias of numpy.uint32

pylidar.lidarformats.las.**LAS_SIMPLEGRID_INDEX_DTYPE**
alias of numpy.uint64

pylidar.lidarformats.las.**LAS_WAVEFORM_TABLE_FIELDS = []**
for building waveforms - need to build unique table of these

class pylidar.lidarformats.las.**LasFile**(*fname*, *mode*, *controls*, *userClass*)
Reader/Writer for .las files.

close()
Write any updated spatial index and close any file handles.

static getDriverName()

getHeader()
Return the Las header as a dictionary.

static getHeaderTranslationDict()
Return dictionary with non-standard header names

getHeaderValue(*name*)
Just extract the one value and return it

getNativeDataType(*colName*, *arrayType*)
Return the native dtype (numpy.int16 etc)that a column is stored as internally after scaling is applied. Provided so scaling can be adjusted when translating between formats.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

getPixelGrid()
Return the PixelGridDefn for this file

getScaling(*colName*, *arrayType*)
Returns the scaling (gain, offset) for the given column name reads from our cache since only written to file on close

Raises generic.LiDARArrayColumnError if no scaling (yet) set for this column.

getScalingColumns(*arrayType*)
Return the list of columns that need scaling. Only valid on write

getTotalNumberPulses()
If BUILD_PULSES == False then the number of pulses will equal the number of points and we can return that. Otherwise we have no idea how many so we raise an exception to flag that.

static getWktFromEPSG(*epsg*)
Gets the WKT from a given EPSG via GDAL.

hasSpatialIndex()
Returns True if the las file has an associated spatial index.

readData(*extent=None*)
Internal method. Just reads into the self.last* fields

readPointsByPulse(*colNames=None*)
Read a 2d structured masked array containing the points for each pulse.

readPointsForExtent(*colNames=None*)
Read all the points within the given extent as 1d structured array. The names of the fields in this array will be defined by the driver.

colNames can be a name or list of column names to return. By default all columns are returned.

readPointsForExtentByBins(*extent=None*, *colNames=None*, *indexByPulse=False*, *returnPulseIndex=False*)
Read all the points within the given extent as a 3d structured masked array to match the block/bins being used.

The extent/binning for the read data can be overriden by passing in a Extent instance.

colNames can be a name or list of column names to return. By default all columns are returned.

Pass indexByPulse=True to bin the points by the locations of the pulses instead of the points.

Pass returnPulseIndex=True to also return a masked 3d array of the indices into the 1d pulse array (as returned by readPulsesForExtent())

readPointsForRange(*colNames=None*)
Reads the points for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readPulsesForExtent(*colNames=None*)
Read all the pulses within the given extent as 1d structured array. The names of the fields in this array will be defined by the driver.

colNames can be a name or list of column names to return. By default all columns are returned.

readPulsesForExtentByBins(*extent=None*, *colNames=None*)
Read all the pulses within the given extent as a 3d structured masked array to match the block/bins being used.

The extent/binning for the read data can be overriden by passing in a Extent instance.

colNames can be a name or list of column names to return. By default all columns are returned.

readPulsesForRange(*colNames=None*)
Reads the pulses for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readReceived()
Read the received waveform for all pulses returns a 2d masked array

readTransmitted()
las (AFAIK) doesn't support transmitted

readWaveformInfo()
2d structured masked array containing information about the waveforms.

setExtent(*extent*)
Set the extent for reading for the ForExtent() functions.

setHeader(*newHeaderDict*)
Update our cached dictionary

setHeaderValue(*name*, *value*)
Just update one value in the header

setNativeDataType(*colName*, *arrayType*, *dtype*)
Set the native dtype (numpy.int16 etc)that a column is stored as internally after scaling (if any) is applied.

arrayType is one of the lidarprocessor.ARRAY_TYPE_* constants

generic.LiDARArrayColumnError is raised if this cannot be set for the format.

The default behaviour is to create new columns in the correct type for the format, or if they are optional, in the same type as the input array.

setPixelGrid(*pixGrid*)
Set the PixelGridDefn for the reading or writing. We don't need to do much here apart from record the EPSG since LAS doesn't use a grid.

setPulseRange(*pulseRange*)
Sets the PulseRange object to use for non spatial reads/writes.

setScaling(*colName*, *arrayType*, *gain*, *offset*)
Set the scaling for the given column name. Currently scaling is only supported for X, Y and Z columns for points, or optional fields.

writeData(*pulses=None*, *points=None*, *transmitted=None*, *received=None*, *waveformInfo=None*)
This driver does not support writing so ignore if reading, throw and error otherwise.

class pylidar.lidarformats.las.**LasFileInfo**(*fname*)
Class that gets information about a .las file and makes it available as fields.

static getDriverName()

static getHeaderTranslationDict()
Return dictionary with non-standard header names

pylidar.lidarformats.las.**READSUPPORTEDOPTIONS = None**
driver options

pylidar.lidarformats.las.**WRITESUPPORTEDOPTIONS = None**
driver options

pylidar.lidarformats.las.**gatherWavePackets**(*data*, *otherArgs*)
Called from lidarprocessor, is the function that does the identification of unique waveform length and gain and offset.

pylidar.lidarformats.las.**getWavePacketDescriptions**(*fname*)
When writing a LAS file, it is necessary to write information to a table in the header that really belongs to the waveforms. This function reads the waveform info from the input file (in any format) and gathers the unique information from it so it can be passed to as the WAVEFORM_DESCR LAS driver option.

Note: LAS only supports received waveforms.

pylidar.lidarformats.las.**isLasFile**(*fname*)
Helper function that looks at the start of the file to determine if it is a las file or not

- genindex
- modindex
- search

5.7 riegl

- genindex
- modindex
- search

5.8 ASCII

Driver for ASCII files. Currently uncompressed files with a .dat or .csv extension are supported. If the ZLIB_ROOT environment variable was set at build time, then gzip compressed files with a .gz extension are also supported.

The user must specify the column names and types via the COL_TYPES driver option. For time-sequential files, the PULSE_COLS driver option needs to be provided so the points can be grouped into pulses. When PULSE_COLS isn't provided then the file is assumed to be non-time-sequential and a pulse is created for each point.

5.8.1 Driver Options

These are contained in the SUPPORTEDOPTIONS module level variable.

Name	Use
COL_TYPES	A numpy style list of tuples defining the data types of each column. Each tuple should have a name and a numpy dtype.
PULSE_COLS	A list of fields which define the pulses. The values in the these columns will be matched and where equal will be put into one pulse and the other values into the points for that pulse.
CLASSIFICATION_CODES	A list of tuples to translate the codes used within the file to the lidarprocessor.CLASSIFICATION_* ones. Each tuple should have the internalCode first, then the lidarprocessor code Codes without a translation will be copied through without change.
COMMENT_CHAR	A single character that defines what is used in the file to denote comments. Lines that start with this character are ignored. Defaults to '#'

class pylidar.lidarformats.ascii.**ASCIIFile**(*fname*, *mode*, *controls*, *userClass*)

Driver for reading ASCII files. Uses the underlying _ascii C++ module.

close()

Write any updated spatial index and close any file handles.

static getDriverName()

getHeader()

ASCII files have no header

static getHeaderTranslationDict()

No header so not really supported. Return empty dict.

getHeaderValue(*name*)

Just extract the one value and return it

getTotalNumberPulses()

No idea how to find this out...

hasSpatialIndex()

ASCII files aren't spatially indexed

readData()

Internal method. Reads all the points and pulses for the current pulse range.

readPointsByPulse(*colNames=None*)

Read a 3d structured masked array containing the points for each pulse.

readPointsForRange(*colNames=None*)

Reads the points for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readPulsesForRange(*colNames=None*)

Reads the pulses for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readReceived()

ASCII (AFAIK) doesn't support received

readTransmitted()

ASCII (AFAIK) doesn't support transmitted

readWaveformInfo()

ASCII (AFAIK) doesn't support waveforms

setPulseRange(*pulseRange*)
Sets the PulseRange object to use for non spatial reads/writes.

writeData(*pulses=None*, *points=None*, *transmitted=None*, *received=None*, *waveformInfo=None*)
This driver does not support writing so ignore if reading, throw and error otherwise.

class pylidar.lidarformats.ascii.**ASCIIFileInfo**(*fname*)
Class that gets information about a .las file and makes it available as fields.

static getDriverName()

static getHeaderTranslationDict()
No header so not really supported. Return empty dict.

pylidar.lidarformats.ascii.**COMPULSARYOPTIONS = ('COL_TYPES',)**
necessary driver options

pylidar.lidarformats.ascii.**SUPPORTEDOPTIONS = ('COL_TYPES', 'PULSE_COLS', 'CLASSIFICATION_C**
driver options

- genindex
- modindex
- search

5.9 LVIS Binary

Driver for LVIS Binary files. Read only.

5.9.1 Read Driver Options

These are contained in the READSUPPORTEDOPTIONS module level variable.

Name	Use
POINT_FROM	integer. Set to one of the POINT_FROM_* module level constants. Determines which file the coordinates for the point is created from. Defaults to POINT_FROM_LCE

class pylidar.lidarformats.lvisbin.**LVISBinFile**(*fname*, *mode*, *controls*, *userClass*)
Reader for LVIS Binary files

close()
Write any updated spatial index and close any file handles.

static getDriverName()

getHeader()
No header for LVIS files

getHeaderValue(*name*)
Just extract the one value and return it

getTotalNumberPulses()
Return the total number of pulses

hasSpatialIndex()
LVIS does not have a spatial index

readData(*extent=None*)
Internal method. Just reads into the self.last* fields

readPointsByPulse(*colNames=None*)
Return a 2d masked structured array of point that matches the pulses.

readPointsForRange(*colNames=None*)
Reads the points for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readPulsesForRange(*colNames=None*)
Reads the pulses for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readReceived()
Read the received waveform for all pulses returns a 2d masked array

readTransmitted()
Read the transmitted waveform for all pulses returns a 3d masked array.

readWaveformInfo()
2d structured masked array containing information about the waveforms.

setPulseRange(*pulseRange*)
Sets the PulseRange object to use for non spatial reads/writes.

writeData(*pulses=None*, *points=None*, *transmitted=None*, *received=None*, *waveformInfo=None*)
Write all the updated data. Pass None for data that do not need to be up It is assumed that each parameter has been read by the reading functions

class pylidar.lidarformats.lvisbin.**LVISBinFileInfo**(*fname*)
Class that gets information about a LVIS file and makes it available as fields.

static getDriverName()

pylidar.lidarformats.lvisbin.**POINT_FROM_LGWEND = None**
How the points are set

pylidar.lidarformats.lvisbin.**READSUPPORTEDOPTIONS = ('POINT_FROM',)**
Supported read options

pylidar.lidarformats.lvisbin.**getFilenames**(*fname*)
Given a filename, determines if it is one of the .lce, .lge, .lgw files and determines the other ones. Returns name of lce, lge and lgw

pylidar.lidarformats.lvisbin.**translateChars**(*input*, *old*, *new*)
Translate any instances of old into new in string input. Assumes old and new are lowercase. Checks also for uppercase old and replaces with uppercase new.

Use this to replace chars in the file extension while preserving the case.

- genindex
- modindex
- search

5.10 LVIS HDF5

Driver for LVIS HDF5 files. Read only.

5.10.1 Read Driver Options

These are contained in the READSUPPORTEDOPTIONS module level variable.

Name	Use
POINT_FROM	A 3 element tuple defining which fields to create a fake point from (x,y,z). Default is ('LON0', 'LAT0', 'Z0')

pylidar.lidarformats.lvishdf5.**CLASSIFICATION_NAME = 'CLASSIFICATION'**
LVIS Files don't have a CLASSIFICATION column so we have to create a blank one for SPDV4

class pylidar.lidarformats.lvishdf5.**LVISHDF5File**(*fname*, *mode*, *controls*, *userClass*)
Reader for LVIS HDF5 files

close()
Write any updated spatial index and close any file handles.

static getDriverName()

getHeader()
Get the header as a dictionary

getHeaderValue(*name*)
Just extract the one value and return it

getTotalNumberPulses()
Return the total number of pulses

hasSpatialIndex()
LVIS does not have a spatial index

static readHeaderAsDict(*fileHandle*)
Internal method to gather info from file and build into a dictionary.

readPointsByPulse(*colNames=None*)
Return a 2d masked structured array of point that matches the pulses.

readPointsForRange(*colNames=None*)
Reads the points for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readPulsesForRange(*colNames=None*)
Reads the pulses for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readRange(*colNames=None*)
Internal method. Returns the requested column(s) as a structured array. Since both points and pulses come from the same place this function is called to read both.

Assumes colName is not None

readReceived()
Return the 3d masked integer array of received for each of the current pulses. First axis is the waveform bin. Second axis is waveform number and last is pulse.

readTransmitted()
Return the 3d masked integer array of transmitted for each of the current pulses. First axis is the waveform bin. Second axis is waveform number and last is pulse.

readWaveformInfo()
2d structured masked array containing information about the waveforms.

setPulseRange(*pulseRange*)
Sets the PulseRange object to use for non spatial reads/writes.

writeData(*pulses=None*, *points=None*, *transmitted=None*, *received=None*, *waveformInfo=None*)
Write all the updated data. Pass None for data that do not need to be up It is assumed that each parameter has been read by the reading functions

class pylidar.lidarformats.lvishdf5.**LVISHDF5FileInfo**(*fname*)
Class that gets information about a LVIS file and makes it available as fields.

static getDriverName()

pylidar.lidarformats.lvishdf5.**READSUPPORTEDOPTIONS = ('POINT_FROM',)**
Supported read options

- genindex
- modindex
- search

5.11 PulseWaves

Driver for PulseWaves.

5.11.1 Read Driver Options

These are contained in the READSUPPORTEDOPTIONS module level variable.

Name	Use
POINT_FROM	an integer. Set to one of the POINT_FROM_* module level constants. Determines which file the coordinates for the point is created from. Defaults to POINT_FROM_ANCHOR

pylidar.lidarformats.pulsewaves.**DEFAULT_HEADER = {'FILE_CREATION_DAY': 231, 'FILE_CREATION_**
for new files

pylidar.lidarformats.pulsewaves.**POINT_FROM_TARGET = None**
How the points are set

class pylidar.lidarformats.pulsewaves.**PulseWavesFile**(*fname*, *mode*, *controls*, *userClass*)
Reader/Writer for PulseWaves files

close()
Write any updated spatial index and close any file handles.

static getDriverName()

getHeader()
Get header as a dictionary

getHeaderValue(*name*)
Just extract the one value and return it

getTotalNumberPulses()
Return the total number of pulses

hasSpatialIndex()
PulseWaves does not have a spatial index

readData(*extent=None*)
Internal method. Just reads into the self.last* fields

readPointsByPulse(*colNames=None*)
Return a 2d masked structured array of point that matches the pulses.

readPointsForRange(*colNames=None*)
Reads the points for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readPulsesForRange(*colNames=None*)
Reads the pulses for the current range. Returns a 1d array.

Returns an empty array if range is outside of the current file.

colNames can be a list of column names to return. By default all columns are returned.

readReceived()
Read the received waveform for all pulses returns a 2d masked array

readTransmitted()
Read the transmitted waveform for all pulses returns a 3d masked array.

readWaveformInfo()
2d structured masked array containing information about the waveforms.

setHeader(*newHeaderDict*)
Update our cached dictionary

setHeaderValue(*name*, *value*)
Just update one value in the header

setPulseRange(*pulseRange*)
Sets the PulseRange object to use for non spatial reads/writes.

writeData(*pulses=None*, *points=None*, *transmitted=None*, *received=None*, *waveformInfo=None*)
Write all the updated data. Pass None for data that do not need to be up It is assumed that each parameter has been read by the reading functions

class pylidar.lidarformats.pulsewaves.**PulseWavesFileInfo**(*fname*)
Class that gets information about a PulseWaves file and makes it available as fields.

static getDriverName()

pylidar.lidarformats.pulsewaves.**READSUPPORTEDOPTIONS = ('POINT_FROM',)**
Supported read options

pylidar.lidarformats.pulsewaves.**isPulseWavesFile**(*fname*)
Helper function that looks at the start of the file to determine if it is a pulsewaves file or not

- genindex
- modindex
- search

5.12 h5space

A utilities to extend h5py's reading and writing of ranges of data, specifically the ability to quickly deal with multiple ranges.

class pylidar.lidarformats.h5space.**H5Space**(*size*, *boolArray=None*, *boolStart=None*, *indices=None*)

Object that wraps a h5py.h5s.SpaceID object and allows conversion quickly from boolean arrays used elsewhere. Also contains methods for reading and writing to/from h5py datasets.

getSelectedIndices()

Return the selected indices, mainly for used my advanced spatial indices. Returns self.indices if set, otherwise works it out form boolArray etc.

getSelectionSize()

Return the number of elements that are cuurently selected

read(*dataSet*)

Given a h5py dataset read the data ranges selected and return a numpy array.

updateBoolArray(*mask*)

Update the h5py.h5s.SpaceID object (and cached boolArray) with the mask which applies to the current selection.

write(*dataSet*, *data*)

Given a h5py dataset and a numpy array write the array into the ranges selected.

pylidar.lidarformats.h5space.**createSpaceFromRange**(*start*, *end*, *size*)

Creates a H5Space object given the start and end of a range

- genindex
- modindex
- search

5.13 gridindexutils

Common utility functions for dealing with grid spatial indices

pylidar.lidarformats.gridindexutils.**CreateSpatialIndex**(*coordOne*, *coordTwo*, *binSize*, *coordOneMax*, *coordTwoMin*, *nRows*, *nCols*, *indexDtype*, *countDtype*)

Create a SPD grid spatial index given arrays of the coordinates of the elements.

This can then be used for writing a SPD V3/4 spatial index to file, or to re-bin data to a new grid.

Any elements outside of the new spatial index are ignored and the arrays returned will not refer to them.

Parameters:

- coordOne is the coordinate corresponding to bin row.

- **coordTwo corresponds to bin col.** Note that coordOne will always be reversed, in keeping with widespread conventions that a Y coordinate increases going north, but a grid row number increases going south. This same assumption will be applied even when the coordinates are not cartesian (e.g. angles).
- **binSize is the size (in world coords) of each bin. The V3/4 index definition** requires that bins are square.
- **coordOneMax and coordTwoMin define the top left of the** spatial index to be built. This is the world coordinate of the top-left corner of the top-left bin
- nRows, nCols - size of the spatial index
- indexDtype is the numpy dtype for the index (si_start, below)
- countDtype is the numpy dtype for the count (si_count, below)

Returns:

- **mask - a 1d array of bools of the valid elements. This must be applied** before sortedBins.
- **sortedBins - a 1d array of indices that is used to** re-sort the data into the correct order for using the created spatial index. Since the spatial index puts all elements in the same bin together this is a very important step!
- **si_start - a 2d array of start indexes into the sorted data (see** above)
- si_count - the count of elements in each bin.

pylidar.lidarformats.gridindexutils.**SNAPMETHOD_GREATER = 2**
Constant for use with snapToGrid. Snaps to greater grid value

pylidar.lidarformats.gridindexutils.**SNAPMETHOD_LESS = 1**
Constant for use with snapToGrid. Snaps to lesser grid value

pylidar.lidarformats.gridindexutils.**SNAPMETHOD_NEAREST = 0**
Constant for use with snapToGrid. Snaps to nearest grid value

pylidar.lidarformats.gridindexutils.**convertSPDIdxToReadIdxAndMaskInfo**(*start_idx_array*, *count_array*, *outSize=None*)
Convert either a 2d SPD spatial index or 1d index (pulse to points, pulse to waveform etc) information for reading with h5py and creating a masked array with the indices into the read subset.

Parameters:

- start_idx_array is the 2 or 1d input array of file start indices from SPD
- count_array is the 2 or 1d input array of element counts from SPD
- outSize is the size of the h5py dataset to be read. Set to None to not return the h5space.H5Space object

Returns:

- If outSize is not None, A h5space.H5Space object for reading and writing data.
- A 3 or 2d (depending on if a 2 or 1 array was input) array containing indices into the new subset of the data. This array is arranged so that the first axis contains the indices for each bin (or pulse) and the other axis is the row (and col axis for 3d output) This array can be used to rearrange the data ready from h5py into a ragged array of the correct shape constaining the data from each bin.
- A 3 or 2d (depending on if a 2 or 1 array was input) bool array that can be used as a mask in a masked array of the ragged array (above) of the actual data.

pylidar.lidarformats.gridindexutils.**getSlicesForExtent**(*siPixGrid*, *siShape*, *overlap*, *xMin*, *xMax*, *yMin*, *yMax*)

xMin, xMax, yMin, yMax is the extent snapped to the pixGrid.

pylidar.lidarformats.gridindexutils.**snapToGrid**(*val*, *valOnGrid*, *res*, *method*)

Snaps a coordinate (val) to a grid specified by one coord on that grid (valOnGrid). res is the pixel size of that grid and method is on of SNAPMETHOD_NEAREST, SNAPMETHOD_LESS or SNAPMETHOD_GREATER.

- genindex
- modindex
- search

CHAPTER 6

Testing

6.1 Testing

Functions for performing automated testing on pylidar

6.1.1 Utility Functions

- genindex
- modindex
- search

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

H

I

L

M

N

O

P

R

S

T

U

V

W

X