CONTENTS

1 News ......................................................... 3
   1.1 09-09-2020 ........................................... 3

2 About ....................................................... 5
   2.1 About .................................................. 5

3 Download ................................................... 13
   3.1 Download ................................................ 13

4 Quickstart .................................................. 17
   4.1 Quickstart .............................................. 17

5 Applications ............................................... 25
   5.1 Applications ............................................ 25

6 Community ............................................... 43
   6.1 Community .............................................. 43

7 Drivers ......................................................... 45
   7.1 Pipeline ............................................... 45
   7.2 Readers ............................................... 54
   7.3 Writers ............................................... 113
   7.4 Filters ............................................... 150

8 Dimensions ............................................... 287
   8.1 Dimensions ............................................ 287

9 Types ......................................................... 293
   9.1 Types ................................................... 293

10 Python ...................................................... 295
    10.1 Python ............................................... 295

11 Java ........................................................ 299
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Java</td>
<td>299</td>
</tr>
<tr>
<td>12 Tutorials</td>
<td>305</td>
</tr>
<tr>
<td>12.1 Tutorials</td>
<td>305</td>
</tr>
<tr>
<td>13 Workshop</td>
<td>349</td>
</tr>
<tr>
<td>13.1 Point Cloud Processing and Analysis with PDAL</td>
<td>349</td>
</tr>
<tr>
<td>14 Development</td>
<td>445</td>
</tr>
<tr>
<td>14.1 Development</td>
<td>445</td>
</tr>
<tr>
<td>14.2 Project</td>
<td>501</td>
</tr>
<tr>
<td>14.3 API</td>
<td>519</td>
</tr>
<tr>
<td>14.4 FAQ</td>
<td>581</td>
</tr>
<tr>
<td>14.5 License</td>
<td>584</td>
</tr>
<tr>
<td>14.6 References</td>
<td>585</td>
</tr>
<tr>
<td>15 Indices and tables</td>
<td>587</td>
</tr>
<tr>
<td>Bibliography</td>
<td>589</td>
</tr>
<tr>
<td>Index</td>
<td>591</td>
</tr>
</tbody>
</table>
PDAL is a C++ BSD (http://www.opensource.org/licenses/bsd-license.php) library for translating and manipulating point cloud data (http://en.wikipedia.org/wiki/Point_cloud). It is very much like the GDAL (http://www.gdal.org) library which handles raster and vector data. The About (page 5) page provides high level overview of the library and its philosophy. Visit Readers (page 54) and Writers (page 113) to list data formats it supports, and see Filters (page 150) for filtering operations that you can apply with PDAL.

In addition to the library code, PDAL provides a suite of command-line applications that users can conveniently use to process, filter, translate, and query point cloud data. Applications (page 25) provides more information on that topic.

Finally, PDAL speaks Python by both embedding and extending it. Visit Python (page 295) to find out how you can use PDAL with Python to process point cloud data.

The entire website is available as a single PDF at http://pdal.io/PDAL.pdf
1.1 09-09-2020

PDAL 2.2.0 has been released. You can download (page 13) the source code or follow the quickstart (page 17) to get going in a hurry with Conda.
2.1 About

2.1.1 What is PDAL?

PDAL (https://pdal.io/) is Point Data Abstraction Library. It is a C/C++ open source library and applications for translating and processing point cloud data (https://en.wikipedia.org/wiki/Point_cloud). It is not limited to LiDAR (https://en.wikipedia.org/wiki/Lidar) data, although the focus and impetus for many of the tools in the library have their origins in LiDAR.

2.1.2 What is its big idea?

PDAL allows you to compose operations (page 150) on point clouds into pipelines (page 45) of stages. These pipelines can be written in a declarative JSON syntax or constructed using the available API.

Why would you want to do that?

A task might be to load some ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) (the most common LiDAR binary format) data into a database, but you wanted to transform it into a common coordinate system along the way.

One option would be to write a specialized monolithic program that reads LAS data, reprojects it as necessary, and then handles the necessary operations to insert the data in the appropriate format in the database. This approach has a distinct disadvantage in that without careful planning it could quickly spiral out of control as you add new little tweaks and features to the operation. It ends up being very specific, and it does not allow you to easily reuse the component that reads the LAS data separately from the component that transforms the data.
The PDAL approach is to chain together a set of components, each of which encapsulates specific functionality. The components allow for reuse, composition, and separation of concerns. PDAL views point cloud processing operations as a pipeline composed as a series of stages. You might have a simple pipeline composed of a LAS Reader (page 71) stage, a Reprojection (page 222) stage, and a PostgreSQL Writer (page 138), for example. Rather than writing a single, monolithic specialized program to perform this operation, you can dynamically compose it as a sequence of steps or operations.

A PDAL JSON Pipeline (page 45) that composes this operation to reproject and load the data into PostgreSQL might look something like the following:

```
{
  "pipeline": [
    {
      "type": "readers.las",
      "filename": "input.las"
    },
    {
      "type": "filters.reprojection",
      "out_srs": "EPSG:3857"
    },
    {
      "type": "writers.pgpointcloud",
      "connection": "host='localhost' dbname='lidar' user='hobu'",
      "table": "output",
      "srid": "3857"
    }
  ]
}
```

PDAL can compose intermediate stages for operations such as filtering, clipping, tiling, transforming into a processing pipeline and reuse as necessary. It allows you to define these pipelines as JSON (https://en.wikipedia.org/wiki/JSON), and it provides a command, pipeline (page 32), to allow you to execute them.

**Note:** Raster processing tools often compose operations with this approach. PDAL conceptually steals its pipeline modeling from GDAL (http://gdal.org/)'s Virtual Raster Format
2.1.3 How is it different than other tools?

LASTools

One of the most common open source processing tool suites available for LiDAR processing is LASTools (http://lastools.org) from Martin Isenburg (https://www.cs.unc.edu/~isenburg/). PDAL is different in philosophy in a number of important ways:

1. All components of PDAL are released as open source software under an OSI (https://opensource.org/licenses)-approved license.

2. PDAL allows application developers to provide proprietary extensions that act as stages in processing pipelines. These might be things like custom format readers, specialized exploitation algorithms, or entire processing pipelines.

3. PDAL can operate on point cloud data of any format – not just ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html). LASTools (http://lastools.org) can read and write formats other than LAS, but relates all data to its internal handling of LAS data, limiting it to dimension (page 287) types provided by the LAS format.

4. PDAL is coordinated by users with its declarative JSON (page 45) syntax. LASTools is coordinated by linking lots of small, specialized command line utilities together with intricate arguments.

5. PDAL is an open source project, with all of its development activities available online at https://github.com/PDAL/PDAL

PCL

PCL (http://pointclouds.org) is a complementary, rather than substitute, open source software processing suite for point cloud data. The developer community of the PCL library is focused on algorithm development, robotic and computer vision, and real-time laser scanner processing. PDAL can read and write PCL’s PCD format.
Entwine

Potree

Potree (http://potree.org) is a WebGL (https://en.wikipedia.org/wiki/WebGL) HTML5 point cloud renderer that speaks ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) and LASzip (http://laszip.org) compressed LAS. You can find the software at https://github.com/potree/potree/

Note: See Potree in action using the USGS 3DEP AWS Public Dataset at https://usgs.entwine.io

Others

Other open source point cloud softwares tend to be Desktop GUI, rather than library, focused. They include some processing operations, and sometimes they even embed tools such as PDAL. We’re obviously biased toward PDAL, but you might find useful bits of functionality in them. These other tools include:

- libLAS (http://liblas.org)
- CloudCompare (http://www.danielgm.net/cc/)
- Fusion (http://www.idaholidar.org/tools/fusion-ldv/)
- OrfeoToolbox (https://www.orfeo-toolbox.org/)

Note: The libLAS (http://liblas.org) project is an open source project that predates PDAL, and provides some of the processing capabilities provided by PDAL. It is currently in maintenance mode due to its dependence on LAS, the release of relevant LAStools capabilities as open source, and the completion of Python LAS (https://pypi.python.org/pypi/laspy/1.4.1) software.

2.1.4 Where did PDAL come from?

PDAL takes its cue from another very popular open source project – GDAL (http://gdal.org/). GDAL is Geospatial Data Abstraction Library, and it is used throughout the geospatial software industry to provide translation and processing support for a variety of raster and vector formats. PDAL provides the same capability for point cloud data types.

PDAL evolved out of the development of database storage and access capabilities for the U.S. Army Corps of Engineers CRREL (http://www.erdc.usace.army.mil/Media/Fact-Sheets/Fact-
How is point cloud data different than raster or vector geo data?

Point cloud data are indeed very much like the typical vector point data type of which many geospatial practitioners are familiar, but their volume causes some significant challenges. Besides their X, Y, and Z locations, each point often has full attribute information of other things like Intensity, Time, Red, Green, and Blue.

Typical vector coverages of point data might max out at a million or so features. Point clouds quickly get into the billions and even trillions, and because of this specialized processing and management techniques must be used to handle so much data efficiently.

The algorithms used to extract and exploit point cloud data are also significantly different than typical vector GIS work flows, and data organization is extremely important to be able to efficiently leverage the available computing. These characteristics demand a library oriented toward these approaches and PDAL achieves it.

Note: Possible point cloud dimension types provided and supported by PDAL can be found at Dimensions (page 287).

2.1.5 What tasks are PDAL good at?

PDAL is great at point cloud data translation work flows. It allows users to apply algorithms to data by providing an abstract API to the content – freeing users from worrying about many data format issues. PDAL’s format-free worry does come with a bit of overhead cost. In most cases this is not significant, but for specific processing work flows with specific data, specialized tools will certainly outperform it.

In exchange for possible performance penalty or data model impedance, developers get the freedom to access data over an abstract API, a multitude of algorithms to apply to data within easy reach, and the most complete set of point cloud format drivers in the industry. PDAL also provides a straightforward command line, and it extends simple generic Python processing through Numpy. These features make it attractive to software developers, data managers, and scientists.
2.1.6 What are PDAL’s weak points?

PDAL doesn’t provide a friendly GUI interface, it expects that you have the confidence to dig into the options of Filters (page 150), Readers (page 54), and Writers (page 113). We sometimes forget that you don’t always want to read source code to figure out how things work. PDAL is an open source project in active development, and because of that, we’re always working to improve it. Please visit Community (page 43) to find out how you can participate if you are interested. The project is always looking for contribution, and the mailing list is the place to ask for help if you are stuck.

2.1.7 High Level Overview

PDAL is first and foremost a software library. A successful software library must meet the needs of software developers who use it to provide its software capabilities to their own software. In addition to its use as a software library, PDAL provides some command line applications (page 25) users can leverage to conveniently translate, filter, and process data with PDAL. Finally, PDAL provides Python (http://python.org/) support in the form of embedded operations and Python extensions.

Core C++ Software Library

PDAL provides a C++ API (page 519) software developers can use to provide point cloud processing capabilities in their own software. PDAL is cross-platform C++, and it can compile and run on Linux, OS X, and Windows. The best place to learn how to use PDAL’s C++ API is the test suite (page 510) and its source code (https://github.com/PDAL/PDAL/tree/master/test/unit).

See also:

PDAL software (page 305) development (page 469) tutorials (page 482) have more information on how to use the library from a software developer’s perspective.

Command Line Utilities

PDAL provides a number of applications (page 25) that allow users to coordinate and construct point cloud processing work flows. Some key tasks users can achieve with these applications include:

- Print info (page 29) about a data set
- Data translation (page 39) from one point cloud format to another
- Application of exploitation algorithms
  - Generate a DTM
– Remove noise
– Reproject from one coordinate system to another
– Classify points as *ground/not ground* (page 27)

- *Merge* (page 32) or *split* (page 35) data
- *Catalog* (page 37) collections of data

**Note:** The command line utilities are often simply *pipeline* (page 32) and *Pipeline* (page 45) collected into a convenient application. In many cases you can replicate the functionality of an application entirely within a single pipeline.

---

**Python API**

PDAL supports both embedding Python (http://python.org/) and extending with Python (http://python.org/). These allow you to dynamically interact with point cloud data in a more comfortable and familiar language environment for geospatial practitioners.

**See also:**

The *Python* (page 295) document contains information on how to install and use the PDAL Python extension.

**Julia Plugin**

PDAL supports embedding *Julia* filters. These allow you to dynamically interact with point cloud data in a more comfortable and familiar language environment for geospatial practitioners, while still maintaining high performance.

Additionally the TypedTables.jl, RoamesGeometry.jl and AcceleratedArrays.jl libraries provide some very high-level interfaces for writing efficient filters.

**See also:**

The github repo at https://github.com/cognitive-earth/PDAL-julia contains a docker image, build instructions and some sample filters.

Documentation for the stage *filters.julia* (page 282)
2.1.8 Conclusion

PDAL is an open source project for translating, filtering, and processing point cloud data. It provides a C++ API, command line utilities, and Python extensions. There are many open source software projects for interacting with point cloud data, and PDAL’s niche is in processing, translation, and automation.
CHAPTER
THREE

DOWNLOAD

3.1 Download

Contents

• Download (page 13)
  – Current Release(s) (page 13)
  – Past Releases (page 14)
  – Development Source (page 14)
  – Binaries (page 14)
    – Windows (page 15)
    – RPMs (page 15)
    – Debian (page 15)
    – Alpine (page 15)
    – Conda (page 16)

3.1.1 Current Release(s)

• 2020-09-09 PDAL-2.2.0-src.tar.gz
  (https://github.com/PDAL/PDAL/releases/download/2.2.0/PDAL-2.2.0-src.tar.gz)
  Release Notes (https://github.com/PDAL/PDAL/releases/tag/2.2.0) (md5
  (https://github.com/PDAL/PDAL/releases/download/2.2.0/PDAL-2.2.0-src.tar.gz.md5))
3.1.2 Past Releases

- **2020-03-20** PDAL-2.1.0-src.tar.gz
  (https://github.com/PDAL/PDAL/releases/download/2.1.0/PDAL-2.1.0-src.tar.gz)

- **2019-08-23** PDAL-2.0.1-src.tar.gz
  (https://github.com/PDAL/PDAL/releases/download/2.0.1/PDAL-2.0.1-src.tar.gz)

- **2019-05-09** PDAL-1.9.1-src.tar.gz
  (https://github.com/PDAL/PDAL/releases/download/1.9.1/PDAL-1.9.1-src.tar.gz)

- **2019-04-09** PDAL-1.9.0-src.tar.gz
  (https://github.com/PDAL/PDAL/releases/download/1.9.0/PDAL-1.9.0-src.tar.gz)

- **2018-10-12** PDAL-1.8.0-src.tar.gz
  (http://download.osgeo.org/pdal/PDAL-1.8.0-src.tar.gz)

- **2018-05-13** PDAL-1.7.2-src.tar.gz
  (http://download.osgeo.org/pdal/PDAL-1.7.2-src.tar.gz)

- **2018-04-06** PDAL-1.7.1-src.tar.gz
  (http://download.osgeo.org/pdal/PDAL-1.7.1-src.tar.gz)

3.1.3 Development Source

The main repository for PDAL is located on github at https://github.com/PDAL/PDAL.

You can obtain a copy of the active source code by issuing the following command

```
git clone https://github.com/PDAL/PDAL.git
```

3.1.4 Binaries

In this section we list a number of the binary distributions of PDAL. The table below is intended to provide an overview of some of the differences between the various distributions, as not all features can be enabled in every distribution. This table only summarizes the differences between distributions, and there are several plugins that are not built for any of the distributions. These include Delaunay, GeoWave, MATLAB, MBIO, MRSID, OpenSceneGraph, RDBLIB, and RiVLib. To enable any of these plugins, the reader will need to install any required dependencies and build PDAL from source.
Table 1: PDAL Distribution Feature Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>Docker</th>
<th>RPMs</th>
<th>Debian</th>
<th>Alpine</th>
<th>Conda (page 16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform(s)</td>
<td>linux</td>
<td>linux</td>
<td>linux</td>
<td>linux</td>
<td>win64, mac, linux</td>
</tr>
<tr>
<td>CPD</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greyhound</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Icebridge</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>laszip</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>laz-perf</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NITF</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>OCI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PCL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgpointcloud</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Python</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SQLite</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Windows**

Windows builds are available via Conda Forge (https://anaconda.org/conda-forge/pdal) (64-bit only). See the Conda (page 16) for more detailed information.

**RPMs**

RPMs for PDAL are available at https://copr.fedorainfracloud.org/coprs/neteler/pdal/.

**Debian**

Debian packages are now available on Debian Unstable (https://tracker.debian.org/pkg/pdal).

**Alpine**

Alpine (page 15) is a linux distribution that is compact and frequently used with Docker images. Alpine packages for PDAL are available at https://pkgs.alpinelinux.org/packages?name=*pdal*&branch=edge.

Users have a choice of three separate packages.

1. pdal will install the PDAL binaries only, and is suitable for users who will be using the PDAL command line applications.
2. `pdal-dev` will install development files which are required for users building their own software that will link against PDAL.

3. `py-pdal` will install the PDAL Python extension.

Note that all of these packages reside in Alpine’s `edge/testing` repository, which must be added to your Alpine repositories list. Information on adding and updating repositories can be found in the Alpine documentation.

To install one or more packages on Alpine, use the following command.

```
apk add [package...]  
```

For example, the following command will install both the PDAL application and the Python extension.

```
apk add py-pdal pdal  
```

**Conda**

*Conda* (page 16) can be used on multiple platforms (Windows, macOS, and Linux) to install software packages and manage environments. Conda packages for PDAL are available at https://anaconda.org/conda-forge/pdal.

Conda installation instructions can be found on the Conda website. The instructions below assuming you have a working Conda installation on your system.

Users have a choice of two separate packages.

1. `pdal` will install the PDAL binaries **and** development files.

2. `python-pdal` will install the PDAL Python extension.

To install one or more Conda packages, use the following command.

```
conda install [-c channel] [package...]  
```

Because the PDAL package (and it’s dependencies) live in the Conda Forge (https://anaconda.org/conda-forge/pdal) channel, the command to install both the PDAL application and the Python extension is

```
conda install -c conda-forge pdal python-pdal gdal  
```

It is strongly recommended that you make use of Conda’s environment management system and install PDAL in a separate environment (i.e., not the base environment). Instructions can be found on the Conda website.
4.1 Quickstart

4.1.1 Introduction

The quickest way to start using PDAL is to leverage builds that were constructed by the PDAL development team using Conda (https://conda.io/docs/).

Directly from the Conda front page,

Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer.

This exercise will print the first point of an *ASPRS LAS* (page 71) file. It will utilize the PDAL command line application (page 25) to inspect the file.

**Note:** If you need to compile your own copy of PDAL, see *Compilation* (page 456) for more details.

4.1.2 Install Conda

Conda installation instructions can be found at the following links. Read through them a bit for your platform so you have an idea what to expect.

Note: We will assume you are running on Windows, but the same commands should work in macOS or Linux too – though definition of file paths might provide a significant difference.

Run Conda

On macOS and Linux, all Conda commands are typed into a terminal window. On Windows, commands are typed into the Anaconda Prompt window. Instructions can be found in the Conda Getting Started (https://conda.io/projects/conda/en/latest/user-guide/getting-started.html#starting-conda) guide.

Test Installation

To test your installation, simply run the command `conda list` from your terminal window or the Anaconda Prompt. A list of installed packages should appear.

Install the PDAL Package

A PDAL package based on the latest release, including all recent patches, is pushed to the conda-forge (https://anaconda.org/conda-forge/pdal) channel on anaconda.org with every code change on the PDAL maintenance branch.

Warning: It is a good idea to install PDAL in its own environment (or add it to an existing one). You will NOT want to add it to your default environment named base. Managing environments is beyond the scope of the quickstart, but you can read more about it here (https://conda.io/projects/conda/en/latest/user-guide/getting-started.html#managing-envs).

To install the PDAL package so that we can use it to run PDAL commands, we run the following command to create an environment named myenv, installing PDAL from the conda-forge channel.

```
conda create --yes --name myenv --channel conda-forge pdal
```

Depending on what packages you may or may not have already installed, the output should look something like:

```
Solving environment: done

## Package Plan ##
```
environment location: C:\Miniconda3\envs\myenv

added / updated specs:
- pdal

The following packages will be downloaded:

<table>
<thead>
<tr>
<th>package</th>
<th>build</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdal-1.7.2</td>
<td>py35h33f895e_1 8.6 MB</td>
</tr>
<tr>
<td>conda-forge</td>
<td></td>
</tr>
<tr>
<td>setuptools-39.2.0</td>
<td>py35_0 591 KB</td>
</tr>
<tr>
<td>conda-forge</td>
<td></td>
</tr>
<tr>
<td>numpy-1.14.3</td>
<td>py35h9fa60d3_2 42 KB</td>
</tr>
</tbody>
</table>

Total: 9.2 MB

The following NEW packages will be INSTALLED:

- boost: 1.66.0-py35_vc14_1 conda-forge [vc14]
- boost-cpp: 1.66.0-vc14_1 conda-forge [vc14]
- ca-certificates: 2018.4.16-0 conda-forge
- cairo: 1.14.10-vc14_0 conda-forge [vc14]
- certifi: 2018.4.16-py35_0 conda-forge
- curl: 7.60.0-vc14_0 conda-forge [vc14]
- expat: 2.2.5-vc14_0 conda-forge [vc14]
- flann: 1.9.1-h0953f56_2 conda-forge
- freexl: 1.0.5-vc14_0 conda-forge [vc14]
- geotiff: 1.4.2-vc14_1 conda-forge [vc14]
- hdf4: 4.2.13-vc14_0 conda-forge [vc14]
- hdf5: 1.10.1-vc14_2 conda-forge [vc14]
- hexer: 1.4.0-vc14_1 conda-forge [vc14]
- icc_rt: 2017.0.4-h97af966_0 conda-forge [vc14]
- icu: 58.2-vc14_0 conda-forge [vc14]
- intel-openmp: 2018.0.3-0 conda-forge [vc14]
- jpeg: 9b-vc14_2 conda-forge [vc14]
- kealib: 1.4.7-vc14_4 conda-forge [vc14]
- krb5: 1.14.6-vc14_0 conda-forge [vc14]
- laszip: 3.2.2-vc14_0 conda-forge [vc14]
- laz-perf: 1.2.0-vc14_1 conda-forge [vc14]
- libgdal: 2.2.4-vc14_4 conda-forge [vc14]
- libiconv: 1.15-vc14_0 conda-forge [vc14]
- libnetcdf: 4.6.1-vc14_2 conda-forge [vc14]
<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>libpng</td>
<td>1.6.34-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>libpq</td>
<td>9.6.3-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>libspatialite</td>
<td>4.3.0a-vc14_19</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>libssh2</td>
<td>1.8.0-vc14_2</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>libtiff</td>
<td>4.0.9-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>libxml2</td>
<td>2.9.8-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>libxslt</td>
<td>1.1.32-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>mkl</td>
<td>2018.0.3-1</td>
<td></td>
</tr>
<tr>
<td>mkl_fft</td>
<td>1.0.2-py35_0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>mkl_random</td>
<td>1.0.1-py35_0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>nitro</td>
<td>2.7.dev2-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>numpy</td>
<td>1.14.3-py35h9fa60d3_2</td>
<td></td>
</tr>
<tr>
<td>numpy-base</td>
<td>1.14.3-py35h5c71026_0</td>
<td></td>
</tr>
<tr>
<td>openjpeg</td>
<td>2.3.0-vc14_2</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>openssl</td>
<td>1.0.2o-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>pcl</td>
<td>1.8.1-hd76163c_1</td>
<td>conda-forge</td>
</tr>
<tr>
<td>pdal</td>
<td>1.7.2-py35h33f895e1_1</td>
<td>conda-forge</td>
</tr>
<tr>
<td>pip</td>
<td>9.0.3-py35_0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>pixman</td>
<td>0.34.0-vc14_2</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>postgresql</td>
<td>10.3-py35_vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>proj4</td>
<td>4.9.3-vc14_5</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>python</td>
<td>3.5.5-1</td>
<td>conda-forge</td>
</tr>
<tr>
<td>setuptools</td>
<td>39.2.0-py35_0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>sqlite</td>
<td>3.20.1-vc14_2</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>tiledb</td>
<td>1.4.1</td>
<td>conda-forge</td>
</tr>
<tr>
<td>vc</td>
<td>14-0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>vs2015_runtime</td>
<td>14.0.25420-0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>wheel</td>
<td>0.31.0-py35_0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>wincertstore</td>
<td>0.2-py35_0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>xerces-c</td>
<td>3.2.0-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
<tr>
<td>xz</td>
<td>5.2.3-0</td>
<td>conda-forge</td>
</tr>
<tr>
<td>zlib</td>
<td>1.2.11-vc14_0</td>
<td>conda-forge [vc14]</td>
</tr>
</tbody>
</table>

Downloading **and** Extracting Packages

```
pdal-1.7.2 | 8.6 MB | ####################################
           | 100%   |
setuptools-39.2.0 | 591 KB | ####################################
         | 100%   |
numpy-1.14.3 | 42 KB | ####################################
         | 100%   |
```

Preparing transaction: done
Verifying transaction: done
Executing transaction: done

# To activate this environment, use
# $ conda activate myenv
#
# To deactivate an active environment, use
#
# $ conda deactivate

Note: PDAL’s Python extension is managed separately from the PDAL package. To install it, replace `pdal` with `python-pdal` in any of the commands in this section. Seeing as how PDAL is a dependency of the Python extension, you will actually get two for the price of one!

To install PDAL to an existing environment names `myenv`, we would run the following command.

```
conda install --name myenv --channel conda-forge pdal
```

Finally, to update PDAL to the latest version, run the following.

```
conda update pdal
```

## 4.1.3 Fetch Sample Data

We need some sample data to play with, so we’re going to download the `autzen.laz` file. Inside your terminal (assuming Windows), issue the following command:

```
explorer.exe https://github.com/PDAL/data/raw/master/autzen/autzen.laz
```

In the download dialog, save the file to your Downloads folder, e.g., `C:\Users\hobu\Downloads`.

## 4.1.4 Print the first point

To print the first point only, issue the following command (replacing of course `hobu` with your user name, or another path altogether, depending on where you saved the file).

```
pdal info C:\Users\hobu\Downloads\autzen.laz -p 0
```

Here’s a summary of what’s going on with that command invocation:

1. `pdal`: We’re going to run the `pdal` command.
2. **info**: We want to run *info* (page 29) on the data.

3. **autzen.laz**: The *autzen.laz* file that we want information from.

```
Warning 1: Cannot find datum.csv or gdal_datum.csv
Warning 1: Cannot find ellipsoid.csv
{
    "filename": "C:\Users\hobu\Downloads\autzen.laz",
    "pdal_version": "1.7.2 (git-version: Release)",
    "points":
    {
        "point":
        {
            "Blue": 93,
            "Classification": 1,
            "EdgeOfFlightLine": 0,
            "GpsTime": 245379.3984,
            "Green": 102,
            "Intensity": 4,
            "NumberOfReturns": 1,
            "PointId": 0,
            "PointSourceId": 7326,
            "Red": 84,
            "ReturnNumber": 1,
            "ScanAngleRank": -17,
            "ScanDirectionFlag": 0,
            "UserData": 128,
            "X": 637177.98,
            "Y": 849393.95,
            "Z": 411.19
        }
    }
}
```

### 4.1.5 What’s next?

- Visit *Applications* (page 25) to find out how to utilize PDAL applications to process data on the command line yourself.
- Visit *Development* (page 445) to learn how to embed and use PDAL in your own applications.
- *Readers* (page 54) lists the formats that PDAL can read, *Filters* (page 150) lists the kinds of operations you can do with PDAL, and *Writers* (page 113) lists the formats PDAL can write.
- *Tutorials* (page 305) contains a number of walk-through tutorials for achieving many...
tasks with PDAL.

- *The PDAL workshop* (page 349) contains numerous hands-on examples with screenshots and example data of how to use PDAL *Applications* (page 25) to tackle point cloud data processing tasks.

- *Python* (page 295) describes how PDAL embeds and extends Python and how you can leverage these capabilities in your own programs.

**See also:**

*Community* (page 43) is a good source to reach out to when you’re stuck.
5.1 Applications

PDAL contains consists of a single application, called pdal. Operations are run by invoking the pdal application along with a command name:

```
$ pdal info myfile.las
$ pdal translate input.las output.las
$ pdal pipeline --stdin < pipeline.json
```

Help for each command can be retrieved via the --help switch. The --drivers and --options switches can tell you more about particular drivers and their options:

```
$ pdal info --help
$ pdal translate --drivers
$ pdal pipeline --options writers.las
```

All commands support the following options:

- `--developer-debug` Enable developer debug (don't trap exceptions).
- `--label` A string to use as a process label.
- `--driver` Name of driver to use to override that inferred from file type.

Additional driver-specific options may be specified by using a namespace-prefixed option name. For example, it is possible to set the LAS day of year at translation time with the following option:

```
$ pdal translate \
   --writers.las.creation_doy="42" \
   input.las \
   output.las
```
5.1.1 delta

The delta command is used to select a nearest point from a candidate file for each point in the source file.

$ pdal delta <source> <candidate>

`--source` source file name

`--candidate` candidate file name

`--detail` Output deltas per-point

`--alldims` Compute diffs for all dimensions (not just X,Y,Z)

Example 1:

$ pdal delta ../../test/data/las/1.2-with-color.las \
    ../../test/data/las/1.2-with-color.las

Delta summary for
    source: '../../test/data/las/1.2-with-color.las'
    candidate: '../../test/data/las/1.2-with-color.las'

<table>
<thead>
<tr>
<th>Dimension</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Max</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Mean</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
Example 2:

```
$ pdal delta test/data/1.2-with-color.las \\
test/data/1.2-with-color.las --detail "ID","DeltaX","DeltaY","DeltaZ"
0,0.00,0.00,0.00
1,0.00,0.00,0.00
2,0.00,0.00,0.00
3,0.00,0.00,0.00
4,0.00,0.00,0.00
5,0.00,0.00,0.00
```

5.1.2 density

The density command produces a tessellated hexagonal OGR layer (http://www.gdal.org/ogr_utilities.html) from the output of filters.hexbin (page 263).

```
$ pdal density <input> <output>
```

```
--input, -i Input point cloud file name
--output, -o Output vector data source
--lyr_name OGR layer name to write into datasource
--ogrdriver, -f OGR driver name to use
--sample_size Sample size for automatic edge length calculation. [5000]
--threshold Required cell density [15]
--hole_cull_tolerance_area Tolerance area to apply to holes before cull
--smooth Smooth boundary output
```

5.1.3 ground

The ground command is used to segment the input point cloud into ground versus non-ground returns using filters.smrf (page 155) and filters.outlier (page 159).

```
$ pdal ground [options] <input> <output>
```

```
--input, -i Input filename
--output, -o Output filename
--max_window_size Max window size
--slope Slope
--max_distance Max distance
```

(continues on next page)
The `hausdorff` command is used to compute the Hausdorff distance between two point clouds. In this context, the Hausdorff distance is the greatest of all Euclidean distances from a point in one point cloud to the closest point in the other point cloud.

More formally, for two non-empty subsets \( X \) and \( Y \), the Hausdorff distance \( d_H(X, Y) \) is

\[
d_H(X, Y) = \max \left\{ \sup_{x \in X} \inf_{y \in Y} d(x, y), \sup_{y \in Y} \inf_{x \in X} d(x, y) \right\}
\]

where \( \sup \) and \( \inf \) are the supremum and infimum respectively.

```
$ pdal hausdorff <source> <candidate>
```

The algorithm makes no distinction between source and candidate files (i.e., they can be transposed with no affect on the computed distance).

The command returns 0 along with a JSON-formatted message summarizing the PDAL version, source and candidate filenames, and the Hausdorff distance. Identical point clouds will return a Hausdorff distance of 0.

```
$ pdal hausdorff source.las candidate.las
{
  "filenames":
  [
    "\/path\/to\/source.las",
    "\/path\/to\/candidate.las"
  ]
```

(continues on next page)
Note: The Hausdorff is computed for XYZ coordinates only and as such says nothing about differences in other dimensions or metadata.

5.1.5 info

Displays information about a point cloud file, such as:

- basic properties (extents, number of points, point format)
- coordinate reference system
- additional metadata
- summary statistics about the points
- the plain text format should be reStructured text if possible to allow a user to retransform the output into whatever they want with ease

```
$ pdal info <input>
```

```
--input, -i          Input file name
--all                Dump statistics, schema and metadata
--point, -p          Point to dump --point="1-5,10,100-200" (0-indexed)
--query              Return points in order of distance from the specified location (2D or 3D) --query Xcoord,Ycoord[,Zcoord]/count
--stats              Dump stats on all points (reads entire dataset)
--boundary           Compute a hexagonal hull/boundary of dataset
--dimensions         Dimensions on which to compute statistics
--enumerate          Dimensions whose values should be enumerated
--schema             Dump the schema
--pipeline-serialization Output filename for pipeline serialization
--summary            Dump summary of the info
--metadata           Dump file metadata info
--stdin, -s          Read a pipeline file from standard input
```
If no options are provided, --stats is assumed.

Example 1:

```
$ pdal info test/data/las/1.2-with-color.las \
   --query="636601.87, 849018.59, 425.10"
{
    "0":
    {
      "Blue": 134,
      "Classification": 1,
      "EdgeOfFlightLine": 0,
      "GpsTime": 245383.3880801476,
      "Green": 104,
      "Intensity": 124,
      "NumberOfReturns": 1,
      "PointSourceId": 7326,
      "Red": 134,
      "ReturnNumber": 1,
      "ScanAngleRank": -4,
      "ScanDirectionFlag": 1,
      "UserData": 126,
      "X": 636601.87,
      "Y": 849018.59999999998,
      "Z": 425.10000000000002
    },
    "1":
    {
      "Blue": 134,
      "Classification": 2,
      "EdgeOfFlightLine": 0,
      "GpsTime": 246099.17323102333,
      "Green": 106,
      "Intensity": 153,
      "NumberOfReturns": 1,
      "PointSourceId": 7327,
      "Red": 143,
      "ReturnNumber": 1,
      "ScanAngleRank": -10,
      "ScanDirectionFlag": 1,
      "UserData": 126,
      "X": 636606.76000000001,
      "Y": 849053.94000000006,
      "Z": 425.88999999999999
    }
}
```

30 Chapter 5. Applications
Example 2:

```
$ pdal info test/data/1.2-with-color.las -p 0-10
{
   "filename": "../../test/data/las/1.2-with-color.las",
   "pdal_version": "PDAL 1.0.0.b1 (116d7d) with GeoTIFF 1.4.1 GDAL 1.11.2 LASzip 2.2.0",
   "points": [
      {
         "Blue": 88,
         "Classification": 1,
         "EdgeOfFlightLine": 0,
         "GpsTime": 245380.78254962614,
         "Green": 77,
         "Intensity": 143,
         "NumberOfReturns": 1,
         "PointId": 0,
         "PointSourceId": 7326,
         "Red": 68,
         "ReturnNumber": 1,
         "ScanAngleRank": -9,
         "ScanDirectionFlag": 1,
         "UserData": 132,
         "X": 637012.23999999999,
         "Y": 849028.31000000006,
         "Z": 431.66000000000003
      },
      {
         "Blue": 68,
         "Classification": 1,
         "EdgeOfFlightLine": 0,
         "GpsTime": 245381.45279923646,
         "Green": 66,
         "Intensity": 18,
         "NumberOfReturns": 2,
         "PointId": 1,
         "PointSourceId": 7326,
         "Red": 54,
         "ReturnNumber": 1,
         "ScanAngleRank": -11,
         "ScanDirectionFlag": 1,
         "UserData": 128,
         "X": 636896.329999999996,
         "Y": 849028.31000000006,
         "Z": 431.66000000000003
      }
   ]
}
```

(continues on next page)
5.1.6 merge

The `merge` command will combine input files into a single output file.

$ pdal merge <input> ... <output>

--files, -f List of filenames. The last file listed is taken to be the output file.

This command provides simple merging of files. It provides no facility for filtering, reprojection, etc. The file type of the input files may be different from one another and different from that of the output file.

5.1.7 pipeline

The `pipeline` command is used to execute `Pipeline` (page 45) JSON. By default the pipeline is run in stream mode if possible, otherwise in standard mode. See `Reading with PDAL` (page 305) or `Pipeline` (page 45) for more information.

$ pdal pipeline <input>

--input, -i Input filename
--pipeline-serialization Output file for pipeline serialization
--validate Validate but do not process the pipeline. Also reports whether a pipeline can be streamed.
--progress Name of file or FIFO to which stages should write progress information. The file/FIFO must exist. PDAL will not create the progress file.
--stdin, -s Read pipeline from standard input
--metadata Metadata filename
--stream Run in stream mode. If not possible, exit.
--nostream Run in standard mode.
Substitutions

The `pipeline` command can accept command-line option substitutions and they replace existing options that are specified in the input JSON pipeline. For example, to set the output and input LAS files for a pipeline that does a translation, the `filename` for the reader and the writer can be overridden:

```
$ pdal pipeline translate.json --writers.las.filename=output.laz \
   --readers.las.filename=input.las
```

If multiple stages of the same name exist in the pipeline, all stages would be overridden. In the following example, both colorization filters would have their `dimensions` option overridden to the value “Red:1:1.0, Blue, Green::256.0” by the command shown below:

```
{
   "pipeline" : [
      "input.las",
      {
         "type" : "filters.colorization",
         "raster" : "raster1.tiff"
      },
      {
         "type" : "filters.colorization",
         "raster" : "raster2.tiff"
      },
      "placeholder.laz"
   ]
}
```

```
$ pdal pipeline colorize.json --filters.colorization.dimensions=\n   "Red:1:1.0, Blue, Green::256.0"
```

Option substitution can also refer to the tag of an individual stage. This can be done by using the syntax `--stage.<tagname>.<option>`. This allows options to be set on individual stages, even if there are multiple stages of the same type. For example, if a pipeline contained two LAS readers with tags `las1` and `las2` respectively, the following command would allow assignment of different filenames to each stage:

```
{
   "pipeline" : [
      {
         "tag" : "las1",
         "type" : "readers.las"
      },
      {
         "tag" : "las2",
         "type" : "readers.las"
      }
   ]
}
```

(continues on next page)
Options specified by tag names override options specified by stage types.

### 5.1.8 random

The `random` command is used to create a random point cloud. It uses `readers.faux` (page 60) to create a point cloud containing `count` points drawn randomly from either a uniform or normal distribution. For the uniform distribution, the bounds can be specified (they default to a unit cube). For the normal distribution, the mean and standard deviation can both be set for each of the x, y, and z dimensions.

```
$ pdal random <output>
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--output</code>, <code>-o</code></td>
<td>Output file name</td>
</tr>
<tr>
<td><code>--compress</code>, <code>-z</code></td>
<td>Compress output data (if supported by output format)</td>
</tr>
<tr>
<td><code>--count</code></td>
<td>How many points should we write?</td>
</tr>
<tr>
<td><code>--bounds</code></td>
<td>Extent (in XYZ to clip output to)</td>
</tr>
<tr>
<td><code>--mean</code></td>
<td>A comma-separated or quoted, space-separated list of means</td>
</tr>
<tr>
<td><code>--stddev</code></td>
<td>A comma-separated or quoted, space-separated list of standard deviations</td>
</tr>
<tr>
<td><code>--distribution</code></td>
<td>Distribution (uniform / normal)</td>
</tr>
</tbody>
</table>
5.1.9 sort

The `sort` command uses `filters.mortonorder` (page 211) to sort data by XY values.

```
$ pdal sort <input> <output>
```

```
--input, -i Input filename
--output, -o Output filename
--compress, -z Compress output data (if supported by output format)
--metadata, -m Forward metadata (VLRs, header entries, etc) from previous stages
```

5.1.10 split

The `split` command will create multiple output files from a single input file. The command takes an input file name and an output filename (used as a template) or output directory specification.

```
$ pdal split <input> <output>
```

```
--input, -i Input filename
--output, -o Output filename
--length Edge length for splitter cells
--capacity Point capacity of chipper cells
--origin_x Origin in X axis for splitter cells
--origin_y Origin in Y axis for splitter cells
```

If neither the `--length` nor `--capacity` arguments are specified, an implicit argument of capacity with a value of 100000 is added.

The output argument is a template. If the output argument is, for example, `file.ext`, the output files created are `file_.#.ext` where # is a number starting at one and incrementing for each file created.

If the output argument ends in a path separator, it is assumed to be a directory and the input argument is appended to create the output template. The `split` command never creates directories. Directories must pre-exist.
Example 1:

```bash
$ pdal split --capacity 100000 infile.laz outfile.bpf
```

This command takes the points from the input file `infile.laz` and creates output files `outfile_1.bpf`, `outfile_2.bpf`, ... where each output file contains no more than 100000 points.

5.1.11 tile

The `tile` command will create multiple output files from input files by generating square tiles of points. The command takes an input file name and an output filename template.

This command is similar to the `split` (page 35) command, but differs in several ways. The `tile` command:

- Uses streaming mode to limit the amount of memory consumed by point data.
- Uses a placeholder for filename output.
- Provides for reprojection of data to create consistent output.
- Always creates square tiles that contain all points “covered” by each tile.

```bash
$ pdal tile <input> <output>
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--input</code></td>
<td>Input filename</td>
</tr>
<tr>
<td><code>--output</code></td>
<td>Output filename</td>
</tr>
<tr>
<td><code>--length</code></td>
<td>Edge length for cells [Default: 1000]</td>
</tr>
<tr>
<td><code>--origin_x</code></td>
<td>Origin in X axis for cells [Default: None]</td>
</tr>
<tr>
<td><code>--origin_y</code></td>
<td>Origin in Y axis for cells [Default: None]</td>
</tr>
<tr>
<td><code>--buffer</code></td>
<td>Size of buffer (overlap) to include around each tile [Default: 0]</td>
</tr>
<tr>
<td><code>--out_srs</code></td>
<td>Spatial reference system to which all input points will be reprojected. [Default: None]</td>
</tr>
</tbody>
</table>

The input filename can contain a glob pattern (https://en.wikipedia.org/wiki/Glob_%28programming%29) to allow multiple files as input.

The output filename must contain a placeholder character #. The placeholder character is replaced with an X/Y index of the tile as a part of a cartesian system. For example, if the output filename is specified as `out#.las`, the tile containing the origin will be named `out0_0.las`. The tile to its right will be named `out1_0.las`. The tile above it will be named `out0_1.las`. The command does not create directories – create any desired directories before running.

If an origin is not supplied with as argument, the first point read is used as the origin.
Example 1:

$ pdal tile infile.laz "outfile_.bpf"

This command takes the points from the input file `infile.laz` and creates output files `outfile_0_0.bpf`, `outfile_0_1.bpf`, ... where each output file contains points in the 1000x1000 square units represented by the tile. The X/Y location of the first point is used as the origin of the tile grid.

Example 2:

$ pdal tile "/home/me/files/*" "out_.txt" --out_srs="EPSG:4326"

Reads all files in the directory `/home/me/files` as input and reprojects points to geographic coordinates if necessary. The output is written to a set of text files in the current directory.

5.1.12 tindex

The `tindex` command is used to create a GDAL (http://www.gdal.org)-style tile index for PDAL-readable point cloud types (see `gdaltindex` (http://www.gdal.org/gdaltindex.html)).

The `tindex` command has two modes. The first mode creates a spatial index file for a set of point cloud files. The second mode creates a point cloud file that is the result of merging the points from files referred to in a spatial index file that meet some criteria (usually a geographic region filter).

**tindex Creation Mode**

$ pdal tindex create <tindex> <filespec>

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--tindex</td>
<td>OGR-readable/writeable tile index output</td>
</tr>
<tr>
<td>--filespec</td>
<td>Build: Pattern of files to index. Merge:</td>
</tr>
<tr>
<td>Output filename</td>
<td></td>
</tr>
<tr>
<td>--fast_boundary</td>
<td>Use extent instead of exact boundary</td>
</tr>
<tr>
<td>--lyr_name</td>
<td>OGR layer name to write into datasource</td>
</tr>
<tr>
<td>--index_name</td>
<td>Tile index column name</td>
</tr>
<tr>
<td>--ogrdriver, -f</td>
<td>OGR driver name to use</td>
</tr>
<tr>
<td>-t_srs</td>
<td>Target SRS of tile index</td>
</tr>
<tr>
<td>--a_srs</td>
<td>Assign SRS of tile with no SRS to this value</td>
</tr>
<tr>
<td>--write_absolute_path</td>
<td>Write absolute rather than relative file paths</td>
</tr>
<tr>
<td>--stdin, -s</td>
<td>Read filespec pattern from standard input</td>
</tr>
</tbody>
</table>

5.1. Applications
This command will index the files referred to by `filespec` and place the result in `tindex`. The `tindex` is a vector file or database that will be created by `pdal` as necessary to store the file index. The type of the index file can be specified by specifying the OGR code for the format using the `--ogrdriver` option. If no driver is specified, the format defaults to “ESRI Shapefile”. Any filetype that can be handled by OGR (http://www.gdal.org/ogr_formats.html) is acceptable.

In vector file-speak, each file specified by `filespec` is stored as a feature in a layer in the index file. The `filespec` is a glob pattern (http://man7.org/linux/man-pages/man7/glob.7.html), and normally needs to be quoted to prevent shell expansion of wildcard characters.

**tindex Merge Mode**

```bash
$ pdal tindex merge <tindex> <filespec>
```

This command will read the existing index file `tindex` and merge the points in the indexed files that pass any filter that might be specified, writing the output to the point cloud file specified in `filespec`. The type of the output file is determined automatically from the filename extension.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--tindex</code></td>
<td>OGR-readable/writeable tile index output</td>
</tr>
<tr>
<td><code>--filespec</code></td>
<td>Build: Pattern of files to index. Merge: Output</td>
</tr>
<tr>
<td><code>--lyr_name</code></td>
<td>OGR layer name to write into datasource</td>
</tr>
<tr>
<td><code>--index_name</code></td>
<td>Tile index column name</td>
</tr>
<tr>
<td><code>--ogrdriver, -f</code></td>
<td>OGR driver name to use</td>
</tr>
<tr>
<td><code>--bounds</code></td>
<td>Extent (in XYZ) to clip output to</td>
</tr>
<tr>
<td><code>--polygon</code></td>
<td>Well-known text of polygon to clip output</td>
</tr>
<tr>
<td><code>--t_srs</code></td>
<td>Spatial reference of the clipping geometry.</td>
</tr>
</tbody>
</table>

**Example 1:**

Find all LAS files via `find`, send that file list via STDIN to `pdal tindex`, and write a SQLite tile index file with a layer named `pdal`:

```bash
$ find las/ -iname "*.las" | pdal tindex create index.sqlite -f "SQLite" \
   --stdin --lyr_name pdal
```
Example 2:

Glob a list of LAS files, output the SRS for the index entries to EPSG:4326, and write out an SQLite (http://www.sqlite.org) file.

```
$ pdal tindex create index.sqlite ".las" -f "SQLite" --lyr_name "pdal" \
   --t_srs "EPSG:4326"
```

5.1.13 translate

The `translate` command can be used for simple conversion of files based on their file extensions. It can also be used for constructing pipelines directly from the command-line. By default, processing is done in stream mode if possible, standard mode if not.

```
$ pdal translate [options] input output [filter]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--input, -i</td>
<td>Input filename</td>
</tr>
<tr>
<td>--output, -o</td>
<td>Output filename</td>
</tr>
<tr>
<td>--filter, -f</td>
<td>Filter type</td>
</tr>
<tr>
<td>--json</td>
<td>PDAL pipeline from which to extract filters.</td>
</tr>
<tr>
<td>--pipeline, -p</td>
<td>Pipeline output</td>
</tr>
<tr>
<td>--metadata, -m</td>
<td>Dump metadata output to the specified file</td>
</tr>
<tr>
<td>--reader, -r</td>
<td>Reader type</td>
</tr>
<tr>
<td>--writer, -w</td>
<td>Writer type</td>
</tr>
<tr>
<td>--stream</td>
<td>Run in stream mode. If not possible, exit.</td>
</tr>
<tr>
<td>--nostream</td>
<td>Run in standard mode.</td>
</tr>
</tbody>
</table>

The `--input` and `--output` file names are required options.

If provided, the `--pipeline` option will write the pipeline constructed from the command-line arguments to the specified file. The translate command will not actually run when this argument is given.

The `--json` flag can use used to specify a PDAL pipeline from which filters will be extracted. If a reader or writer exist in the pipeline, they will be removed and replaced with the input and output provided on the command line. If a reader/writer stage references tags in the provided pipeline, the overriding files will assume those tags. If the argument to the `--json` option references an existing file, it is assumed that the file contains the pipeline to be processed. If the argument value is not a filename, it is taken to be a literal JSON string that is the pipeline. The flag can’t be used if filters are listed on the command line or explicitly with the `--filter` option.

The `--filter` flag is optional. It is used to specify drivers used to filter the data. `--filter` accepts multiple arguments if provided, thus constructing a multi-stage filtering operation. Filters can’t be specified using this method and with the `--json` flag.
The --metadata flag accepts a filename for the output of metadata associated with the execution of the translate operation.

If no --reader or --writer type are given, PDAL will attempt to infer the correct drivers from the input and output file name extensions respectively.

Example 1:

The translate command can be augmented by specifying fully specified options at the command-line invocation. For example, the following invocation will translate 1.2-with-color.las to output.laz while doing the following:

- Setting the creation day of year to 42
- Setting the creation year to 2014
- Setting the LAS point format to 1
- Cropping the file with the given polygon

```
$ pdal translate \
   --writers.las.creation_doy="42" \ 
   --writers.las.creation_year="2014" \ 
   --writers.las.format="1" \ 
   ./test/data/1.2-with-color.las \
   output.laz \
   filters.crop
```
Example 2:

Given these tools, we can now construct a custom pipeline on-the-fly. The example below uses a simple LAS reader and writer, but stages a voxel grid filter, followed by the SMRF filter and a range filter. We can even set stage-specific parameters as shown.

```bash
$ pdal translate input.las output.las voxelcenternearestneighbor
   --smrf range
   --filters.range.limits="Classification[2:2]"
```

Example 3:

This command reads the input text file “myfile” and writes an output LAS file “output.las”, processing the data through the stats filter. The metadata output (including the output from the stats filter) is written to the file “meta.json”.

```bash
$ pdal translate myfile output.las --metadata=meta.json -r readers.
   --text
   --json="{"pipeline": [ { "type": "filters.stats" } ] }"
```

Example 4:

This command reprojects the points in the file “input.las” to another spatial reference system and writes the result to the file “output.las”.

```bash
$ pdal translate input.las output.las -f filters.reprojection
   --filters.reprojection.out_srs="EPSG:4326"
```
6.1 Community

PDAL’s community interacts through Mailing List (page 43), GitHub (page 44), Gitter (https://gitter.im/PDAL/PDAL) and IRC (page 44). Please feel welcome to ask questions and participate in all of the venues. The Mailing List (page 43) communication channel is for general questions, development discussion, and feedback. The GitHub (page 44) communication channel is for development activities, bug reports, and testing. The IRC (page 44) and Gitter (https://gitter.im/PDAL/PDAL) channels are for real-time chat activities such as meetings and interactive debugging sessions.

6.1.1 Mailing List

Developers and users of PDAL participate on the PDAL mailing list. It is OK to ask questions about how to use PDAL, how to integrate PDAL into your own software, and report issues that you might have.

http://lists.osgeo.org/mailman/listinfo/pdal

Note: Please remember that an email to the PDAL list is going to 100s of individuals. Do your diligence the best you can on your question before asking, but don’t be afraid to ask. We won’t bite. Promise.
6.1.2 GitHub

Visit http://github.com/PDAL/PDAL to file issues you might be having with the software. GitHub is also where you can obtain a current development version of the software in the git (https://en.wikipedia.org/wiki/Git_(software)) revision control system. The PDAL project is eager to take contributions in all forms, and we welcome those who are willing to roll up their sleeves and start filing tickets, pushing code, generating builds, and answering questions.

See also:

Development (page 445) provides more information on how the PDAL software development activities operate.

6.1.3 Gitter

Some PDAL developers are active on Gitter (https://gitter.im/PDAL/PDAL) and you can use that mechanism for asking questions and interacting with the developers in a mode that is similar to IRC (page 44). Gitter uses your GitHub (page 44) credentials for access, so you will need an account to get started.

6.1.4 Keybase

Some PDAL developers are available via Keybase’s pdal chat. See https://keybase.io/blog/keybase-chat for more details.

6.1.5 IRC

You can find some PDAL developers on IRC on #pdal at Freenode (http://freenode.net). This mechanism is usually reserved for active meetings and other outreach with the community. The Mailing List (page 43) and GitHub (page 44) avenues are going to be more productive communication channels in most situations.
7.1 Pipeline

Pipelines define the processing of data within PDAL. They describe how point cloud data are read, processed and written. PDAL internally constructs a pipeline to perform data translation operations using `translate` (page 39), for example. While specific applications (page 25) are useful in many contexts, a pipeline provides useful advantages for many workflows:

1. You have a record of the operation(s) applied to the data
2. You can construct a skeleton of an operation and substitute specific options (filenames, for example)
3. You can construct complex operations using the JSON (http://www.json.org/) manipulation facilities of whatever language you want.

**Note:** `pipeline` (page 32) is used to invoke pipeline operations via the command line.

### 7.1.1 Introduction

A PDAL processing pipeline is represented in JSON. The structure may either:

- a JSON object, with a key called `pipeline` whose value is an array of inferred or explicit PDAL `Stage Objects` (page 49) representations.
- a JSON array, being the array described above without being encapsulated by a JSON object.
Simple Example

A simple PDAL pipeline, inferring the appropriate drivers for the reader and writer from filenames, and able to be specified as a set of sequential steps:

```
[
    "input.las",
    {
        "type" : "filters.crop",
        "bounds" : "([0,100],[0,100])"
    },
    "output.bpf"
]
```

Fig. 1: A simple pipeline to convert LAS (page 71) to BPF (page 54) while only keeping points inside the box \([0 \leq x \leq 100, 0 \leq y \leq 100]\).

Reprojection Example

A more complex PDAL pipeline reprojects the stage tagged \(A_1\), merges the result with \(B\), and writes the merged output to a GeoTIFF file with the \texttt{writers.gdal} (page 119) writer:

```
[
    {
        "filename" : "A.las",
        "spatialreference" : "EPSG:26916"
    },
    {
        "type" : "filters.reprojection",
        "in_srs" : "EPSG:26916",
        "out_srs" : "EPSG:4326",
        "tag" : "A2"
    },
    {
        "filename" : "B.las",
        "tag" : "B"
    },
    {
        "type" : "filters.merge",
        "tag" : "merged",
    }
]
```

(continues on next page)
Point Views and Multiple Outputs

Some filters produce sets of points as output. filters.splitter (page 256), for example, creates a point set for each tile (rectangular area) in which input points exist. Each of these output sets is called a point view. Point views are carried through a PDAL pipeline individually. Some writers can produce separate output for each input point view. These writers use a placeholder character (#) in the output filename which is replaced by an incrementing integer for each input point view.

The following pipeline provides an example of writing multiple output files from a single pipeline. The crop filter creates two output point views (one for each specified geometry) and the writer creates output files ‘output1.las’ and ‘output2.las’ containing the two sets of points:

```json
{
    "inputs": [
        "A2",
        "B"
    ],
    "type": "writers.gdal",
    "filename": "output.tif"
}
```
7.1.2 Processing Modes

PDAL process data in one of two ways: standard mode or stream mode. With standard mode, all input is read into memory before it is processed. Many algorithms require standard mode processing because they need access to all points. Operations that do sorting or require neighbors of points, for example, require access to all points.

For operations that don’t require access to all points, PDAL provides stream mode. Stream mode processes points through a pipeline in chunks, which reduces memory requirements.

When using `pdal translate` (page 39) or `pdal pipeline` (page 32) PDAL uses stream mode if possible. If stream mode can’t be used the applications fall back to standard mode processing. Streamable stages are tagged in the stage documentation with a blue bar. Users can explicitly choose to use standard mode by using the `--nostream` option. Users of the PDAL API can explicitly control the selection of the PDAL processing mode.

7.1.3 Pipelines

Pipeline Array

PDAL JSON pipelines are an array of stages.

**Note:** In versions of PDAL prior to 1.9, the array of stages needed to be the value of a key named “pipeline” which was encapsulated in an object. The earlier format is still accepted for backward compatibility.

Old format:

```json
{
    "pipeline" : [
        "inputfile",
        "outputfile"
    ]
}
```
Equivalent new format:

```
[  "inputfile",
   "outputfile"
]
```

- The pipeline array may have any number of string or *Stage Objects* (page 49) elements.
- String elements shall be interpreted as_filenames_. PDAL will attempt to infer the proper driver from the file extension and position in the array. A writer stage will only be created if the string is the final element in the array.

**Stage Objects**

For more on PDAL stages and their options, check the PDAL documentation on *Readers* (page 54), *Writers* (page 113), and *Filters* (page 150).

- A stage object may have a member with the name `tag` whose value is a string. The purpose of the tag is to cross-reference this stage within other stages. Each `tag` must be unique.
- A stage object may have a member with the name `inputs` whose value is an array of strings. Each element in the array is the tag of another stage to be set as input to the current stage.
- Reader stages will disregard the `inputs` member.
- If `inputs` is not specified for the first non-reader stage, all reader stages leading up to the current stage will be used as inputs.
- If `inputs` is not specified for any subsequent non-reader stages, the previous stage in the array will be used as input.
- A `tag` mentioned in another stage’s `inputs` must have been previously defined in the pipeline array.
- A reader or writer stage object may have a member with the name `type` whose value is a string. The `type` must specify a valid PDAL reader or writer name.
- A filter stage object must have a member with the name `type` whose value is a string. The `type` must specify a valid PDAL filter name.
- A stage object may have additional members with names corresponding to stage-specific option names and their respective values. Values provided as JSON objects or arrays will be stringified and parsed within the stage. Some options allow multiple inputs. In those cases, provide the option values as a JSON array.
• A user_data option can be added to any stage object and it will be carried through to any serialized pipeline output.

• All stages support the option_file option that allows options to be placed in a separate file. See Option Files (page 50) for details.

Filename Globbing

• A filename may contain the wildcard character * to match any string of characters. This can be useful if working with multiple input files in a directory (e.g., merging all files).

Filename globbing ONLY works in pipeline file specifications. It doesn’t work when a filename is provided as an option through a command-line application like pdal pipeline or pdal translate.

Option Files

All stages accept the option_file option that allows extra options for a stage to be placed in a separate file. The value of the option is the filename in which the additional options are located.

Option files can be written using either JSON syntax or command line syntax. When using the JSON syntax, the format is a block of options just as if the options were placed in a pipeline:

```
{
    "minor_version": 4,
    "out_srs": "EPSG_4326"
}
```

When using the command line syntax, the options are specified as they would be on the command line without the need to qualify the option names with the stage name:

```
--minor_version=4 --out_srs="EPSG_4326"
```

7.1.4 Extended Examples

BPF to LAS

The following pipeline converts the input file from BPF (page 54) to LAS (page 125), inferring both the reader and writer type, and setting a number of options on the writer stage.
In our next example, the reader and writer types are once again inferred. After reading the input file, the ferry filter is used to copy the Z dimension into a new height above ground (HAG) dimension. Next, the `filters.python` (page 276) is used with a Python script to compute height above ground values by comparing the Z values to a surface model. These height above ground values are then written back into the Z dimension for further analysis. See the Python code at `hag.py` (https://raw.githubusercontent.com/PDAL/PDAL/master/test/data/autzen/hag.py.in).

See also:

`filters.hag_nn` (page 171) describes using a specific filter to do this job in more detail.
DTM

A common task is to create a digital terrain model (DTM) from the input point cloud. This pipeline infers the reader type, applies an approximate ground segmentation filter using `filters.smrf` (page 155), filters out all points but the ground returns (classification value of 2) using the `filters.range` (page 248), and then creates the DTM using the `writers.gdal` (page 119).

```
[   
    "autzen-full.las",
    {   
      "type":"filters.smrf",
      "window":33,
      "slope":1.0,
      "threshold":0.15,
      "cell":1.0
    },
    {   
      "type":"filters.range",
      "limits":"Classification[2:2]"
    },
    {   
      "type":"writers.gdal",
      "filename":"autzen-surface.tif",
      "output_type":"min",
      "gdaldriver":"GTiff",
      "window_size":3,
      "resolution":1.0
    }
]
```

Decimate & Colorize

This example still infers the reader and writer types while applying options on both. The pipeline decimates the input LAS file by keeping every other point, and then colorizes the points using the provided raster image. The output is written as ASCII text.

```
[   
    {   
      "filename":"1.2-with-color.las",
      "spatialreference":"EPSG:2993"
    },
    {   
      "type":"filters.decimation",
      "step":2,
      "offset":1
    }
]
```

(continues on next page)
Reproject

Our first example with multiple readers, this pipeline infers the reader types, and assigns spatial reference information to each. `filters.reprojection` (page 222) filter reprojects data to the specified output spatial reference system.

```json
[
  {
    "filename":"1.2-with-color.las",
    "spatialreference":"EPSG:2027"
  },
  {
    "filename":"1.2-with-color.las",
    "spatialreference":"EPSG:2027"
  },
  {
    "type":"filters.reprojection",
    "out_srs":"EPSG:2028"
  }
]
```

Globbed Inputs

Finally, we capture another merge pipeline demonstrating the ability to glob multiple input LAS files from a given directory.

```json
[
  "/path/to/data/*.las",
  "output.las"
]
```
See also:
The PDAL source tree contains a number of example pipelines that are used for testing. You might find these inspiring. Go to https://github.com/PDAL/PDAL/tree/master/test/data/pipeline to find more.

Note: Issuing the command `pdal info --options` will list all available stages and their options. See `info` (page 29) for more.

## 7.2 Readers

Readers provide *Dimensions* (page 287) to *Pipeline* (page 45). PDAL attempts to normalize common dimension types, like X, Y, Z, or Intensity, which are often found in LiDAR point clouds. Not all dimension types need to be fixed, however. Database drivers typically return unstructured lists of dimensions. A reader might provide a simple file type, like `readers.text` (page 106), a complex database like `readers.pgpointcloud` (page 89), or a network service like `readers.ept` (page 56).

### 7.2.1 readers.bpf

BPF is an NGA [specification](https://nsgreg.nga.mil/doc/view?i=4220&month=8&day=30&year=2016) for point cloud data. The BPF reader supports reading from BPF files that are encoded as version 1, 2 or 3.

This BPF reader only supports Zlib compression. It does NOT support the deprecated compression types QuickLZ and FastLZ. The reader will consume files containing ULEM frame data and polarimetric data, although these data are not made accessible to PDAL; they are essentially ignored.

Data that follows the standard header but precedes point data is taken to be metadata and is UTF-encoded and added to the reader’s metadata.

**Default Embedded Stage**

This stage is enabled by default

**Streamable Stage**

This stage supports streaming operations
Example

```json
[
    "inputfile.bpf",
    {
        "type": "writers.text",
        "filename": "outputfile.txt"
    }
]
```

Options

- **filename**  BPF file to read [Required]
- **fix_dims**  BPF files may contain dimension names that aren’t allowed by PDAL. When this option is ‘true’, invalid characters in dimension names are replaced by ‘_’ in order to make the names valid. [Default: true]
- **count**  Maximum number of points to read. [Default: unlimited]
- **override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
- **default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

7.2.2 readers.buffer

The *readers.buffer* (page 55) stage is a special stage that allows you to read data from your own PointView rather than fetching the data from a specific reader. In the *Writing with PDAL* (page 469) example, it is used to take a simple listing of points and turn them into an LAS file.

Default Embedded Stage

This stage is enabled by default
Example

See *Writing with PDAL* (page 469) for an example usage scenario for *readers.buffer* (page 55).

Options

**count**  Maximum number of points to read. [Default: unlimited]

**override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with `default_srs` [Default: none]

**default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with `override_srs` [Default: none]

### 7.2.3 readers.ept

Entwine Point Tile (https://entwine.io/entwine-point-tile.html) (EPT) is a hierarchical octree-based point cloud format suitable for real-time rendering and lossless archival. Entwine (https://entwine.io/) is a producer of this format. The EPT Reader supports reading data from the EPT format, including spatially accelerated queries and file reconstruction queries.

Sample EPT datasets of hundreds of billions of points in size may be viewed with Potree (http://potree.entwine.io/data/nyc.html) or `Plasio`_.

**Default Embedded Stage**

This stage is enabled by default

**Example**

This example downloads a small area around the the Statue of Liberty from the New York City data set (4.7 billion points) which can be viewed in its entirety in Potree (http://potree.entwine.io/data/nyc.html).

```py
[
  {
    "type": "readers.ept",
    "filename": "http://na.entwine.io/nyc/ept.json",
    "bounds": "([-8242669, -8242529], [4966549, 4966674])"
  },
  "statue-of-liberty.las"
]`
Additional attributes created by the EPT addon writer (page 116) can be referenced with the addon option. Here is an example that overrides the Classification dimension with an addon dimension derived from the original dataset:

```json
[
  {
    "type": "readers.ept",
    "filename": "http://na.entwine.io/autzen/ept.json",
    "addons": {
      "Classification": "~/entwine/addons/autzen/smrf"
    }
  },
  {
    "type": "writers.las",
    "filename": "autzen-ept-smrf.las"
  }
]
```

For more details about addon dimensions and how to produce them, see writers.ept_addon (page 116).

### Options

- **filename** Path to the EPT resource from which to read, ending with ept.json. For example, /Users/connor/entwine/autzen/ept.json or http://na.entwine.io/autzen/ept.json. [Required]

- **spatialreference** Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. [Default: none]

- **bounds** The extents of the resource to select in 2 or 3 dimensions, expressed as a string, e.g.: ([xmin, xmax], [ymin, ymax], [zmin, zmax]). If omitted, the entire dataset will be selected.

- **resolution** A point resolution limit to select, expressed as a grid cell edge length. Units correspond to resource coordinate system units. For example, for a coordinate system expressed in meters, a resolution value of 0.1 will select points up to a ground resolution of 100 points per square meter.

  The resulting resolution may not be exactly this value: the minimum possible resolution that is at least as precise as the requested resolution will be selected. Therefore the result may be a bit more precise than requested.

- **addons** A mapping of assignments of the form DimensionName: AddonPath, which assigns dimensions from the specified paths to the named dimensions. These addon dimensions are created by the EPT addon writer (page 116). If the dimension names already exist in the EPT Schema (https://entwine.io/entwine-point-tile.html#schema) for

7.2. Readers 57
the given resource, then their values will be overwritten with those from the appropriate add-on.

Add-ons may be used to override well-known dimension (page 287). For example, an add-on assignment of "Classification": "~/addons/autzen/MyGroundDimension/" will override an existing EPT
Classification dimension with the custom dimension.

**origin**  EPT datasets are lossless aggregations of potentially multiple source files. The **origin** options can be used to select all points from a single source file. This option may be specified as a string or an integral ID.

The string form of this option selects a source file by its original file path. This may be a substring instead of the entire path, but the string must uniquely select only one source file (via substring search). For example, for an EPT dataset created from source files *one.las*, *two.las*, and *two.bpf*, “one” is a sufficient selector, but “two” is not.

The integral form of this option selects a source file by its **OriginId** dimension, which can be determined from the file’s position in EPT metadata file *entwine-files.json*.

**Note:** When using **pdal info --summary**, using the **origin** option will cause the resulting bounds to be clipped to those of the selected origin, and the resulting number of points to be an upper bound for this selection.

**polygon**  The clipping polygon, expressed in a well-known text string, eg: “POLYGON((0 0, 5000 10000, 10000 0, 0 0))”. This option can be specified more than once by placing values in an array.

**Note:** When using **pdal info --summary**, using the **polygon** option will cause the resulting bounds to be clipped to the maximal extents of all provided polygons, and the resulting number of points to be an upper bound for this polygon selection.

**threads**  Number of worker threads used to download and process EPT data. A minimum of 4 will be used no matter what value is specified.

**header**  HTTP headers to forward for remote EPT endpoints, structured as a JSON object of key/value string pairs.

**query**  HTTP query parameters to forward for remote EPT endpoints, structured as a JSON object of key/value string pairs.
7.2.4 readers.e57

The **E57 Reader** supports reading from E57 files.

The reader supports E57 files with Cartesian point clouds.

---

Note: E57 files can contain multiple point clouds stored in a single file. If that is the case, the reader will read all the points from all of the internal point clouds as one. Only dimensions present in all of the point clouds will be read.

---

Note: Point clouds stored in spherical format are not supported.

---

Note: The E57 `cartesianInvalidState` dimension is mapped to the Omit PDAL dimension. A range filter can be used to filter out the invalid points.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Streamable Stage

This stage supports streaming operations

Example 1

```json
[
  {
    "type": "readers.e57",
    "filename": "inputfile.e57"
  },
  {
    "type": "writers.text",
    "filename": "outputfile.txt"
  }
]
```
Example 2

```
[  
    {  
        "type":"readers.e57",
        "filename":"inputfile.e57"
    },
    {  
        "type":"filters.range",
        "limits":"Omit[0:0]"
    },
    {  
        "type":"writers.text",
        "filename":"outputfile.txt"
    }
]
```

Options

filename  E57 file to read [Required]

count  Maximum number of points to read. [Default: unlimited]

override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

7.2.5 readers.faux

The faux reader is used for testing pipelines. It does not read from a file or database, but generates synthetic data to feed into the pipeline.

The faux reader requires a mode argument to define the method in which points should be generated. Valid modes are as follows:

constant  The values provided as the minimums to the bounds argument are used for the X, Y and Z value, respectively, for every point.

random  Random values are chosen within the provided bounds.

ramp  Value increase uniformly from the minimum values to the maximum values.

uniform  Random values of each dimension are uniformly distributed in the provided ranges.

normal  Random values of each dimension are normally distributed in the provided ranges.
grid  Creates points with integer-valued coordinates in the range provided (excluding the upper bound).

**Default Embedded Stage**
This stage is enabled by default

**Streamable Stage**
This stage supports streaming operations

**Example**

```json
[
  {
    "type": "readers.faux",
    "bounds": "([0,1000000],[0,1000000],[0,100])",
    "count": "10000",
    "mode": "random"
  },
  {
    "type": "writers.text",
    "filename": "outputfile.txt"
  }
]
```

**Options**

- **bounds**  The spatial extent within which points should be generated. Specified as a string in the form “((xmin,xmax],[ymin,ymax],zmin,zmax))”. [Default: unit cube]
- **count**  The number of points to generate. [Required, except when mode is ‘grid’]
- **override_srs**  Spatial reference to apply to data. [Optional]
- **mean_x|y|z**  Mean value in the x, y, or z dimension respectively. (Normal mode only) [Default: 0]
- **stdev_x|y|z**  Standard deviation in the x, y, or z dimension respectively. (Normal mode only) [Default: 1]
- **mode**  “constant,” “random”, “ramp”, “uniform”, “normal” or “grid” [Required]
7.2.6 readers.gdal

The GDAL (http://gdal.org) reader reads GDAL readable raster (http://www.gdal.org/formats_list.html) data sources as point clouds.

Each pixel is given an X and Y coordinate (and corresponding PDAL dimensions) that are center pixel, and each band is represented by “band-1”, “band-2”, or “band-n”. Using the ‘header’ option allows naming the band data to standard PDAL dimensions.

Default Embedded Stage

This stage is enabled by default

Basic Example

Simply writing every pixel of a JPEG to a text file is not very useful.

```
[{
  "type": "readers.gdal",
  "filename": "/pdal/test/data/autzen/autzen.jpg"
},
{
  "type": "writers.text",
  "filename": "outputfile.txt"
}]
```

LAS Example

The following example assigns the bands from a JPG to the RGB values of an ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) file using writers.las (page 125).

```
[{
  "type": "readers.gdal",
  "filename": "/pdal/test/data/autzen/autzen.jpg",
  "header": "Red, Green, Blue"
},
{
  "type": "writers.text",
  "filename": "outputfile.txt"
}]
```

(continues on next page)
Options

filename GDALOpen
(https://gdal.org/api/raster_c_api.html#gdal_8h_1aca0545547235996415f9c891d678d5e)
‘able raster file to read [Required]

count Maximum number of points to read. [Default: unlimited]

override_srs Spatial reference to apply to the data. Overrides any SRS in the input itself. Can
be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs Spatial reference to apply to the data if the input does not specify one. Can be
specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

header A comma-separated list of dimension (page 287) IDs to map bands to. The length of
the list must match the number of bands in the raster.

7.2.7 readers.geowave

The GeoWave reader uses GeoWave (https://github.com/locationtech/geowave) to read from
Accumulo. GeoWave entries are stored using EPSG:4326 (http://epsg.io/4326/).

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```json
[
  {
    "type": "readers.geowave",
    "zookeeper_url": "zookeeper1:2181,zookeeper2:2181,
zookeeper3:2181",
    "instance_name": "GeoWave",
    "username": "user",
    "password": "pass",
    "table_namespace": "PDAL_Table",
}
```

(continues on next page)
"feature_type_name": "PDAL_Point",
"data_adapter": "FeatureCollectionDataAdapter",
"points_per_entry": "5000u",
"bounds": "((0,1000000),(0,1000000),(0,100))",
"filename": "/pdal/test/data/autzen/autzen.jpg",
"
",
{
"type": "writers.text",
"filename": "outputfile.txt"
}
"
}
]

Options

count Maximum number of points to read. [Default: unlimited]

override_srs Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be
specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs Spatial reference to apply to the data if the input does not specify one. Can be
specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

zookeeper_url The comma-delimited URLs for all zookeeper servers, this will be directly
used to instantiate a ZookeeperInstance. [Required]

instance_name The zookeeper instance name, this will be directly used to instantiate a
ZookeeperInstance. [Required]

username The username for the account to establish an Accumulo connector. [Required]

password The password for the account to establish an Accumulo connector. [Required]

table_namespace The table name to be used when interacting with GeoWave. [Required]

feature_type_name The feature type name to be used when interacting GeoWave. [Default: PDAL_Point]

data_adapter FeatureCollectionDataAdapter stores multiple points per Accumulo entry.
FeatureDataAdapter stores a single point per Accumulo entry. [Default: FeatureCollectionDataAdapter]

points_per_entry Sets the maximum number of points per Accumulo entry when using
FeatureCollectionDataAdapter. [Default: 5000]

bounds The extent of the bounding rectangle to use to query points, expressed as a string, eg:
“((xmin,xmax],[ymin,ymax],[zmin,zmax])”. [Default: unit cube]
7.2.8 readers.hdf

The HDF reader reads data from files in the HDF5 format. You must explicitly specify a mapping of HDF datasets to PDAL dimensions using the dimensions parameter. ALL dimensions must be scalars and be of the same length. Compound types are not supported at this time.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Streamable Stage

This stage supports streaming operations

Example

This example reads from the Autzen HDF example with all dimension properly mapped and then outputs a LAS file.

```json
{
    "type": "readers.hdf",
    "filename": "test/data/hdf/autzen.h5",
    "dimensions": {
        "X": "autzen/X",
        "Y": "autzen/Y",
        "Z": "autzen/Z",
        "Red": "autzen/Red",
        "Blue": "autzen/Blue",
        "Green": "autzen/Green",
        "Classification": "autzen/Classification",
        "EdgeOfFlightLine": "autzen/EdgeOfFlightLine",
        "GpsTime": "autzen/GpsTime",
        "Intensity": "autzen/Intensity",
        "NumberOfReturns": "autzen/NumberOfReturns",
        "PointSourceId": "autzen/PointSourceId",
        "ReturnNumber": "autzen/ReturnNumber",
        "ScanAngleRank": "autzen/ScanAngleRank",
        "ScanDirectionFlag": "autzen/ScanDirectionFlag",
        "UserData": "autzen/UserData"
    }
}
```

(continues on next page)
Note: All dimensions must be simple numeric HDF datasets with equal lengths. Compound types, enum types, string types, etc. are not supported.

Warning: The HDF reader does not set an SRS.

Common Use Cases

A possible use case for this driver is reading NASA’s ICESat-2 (https://icesat-2.gsfc.nasa.gov/) data. This example reads the X, Y, and Z coordinates from the ICESat-2 ATL03 (https://icesat-2.gsfc.nasa.gov/sites/default/files/page_files/ICESat2_ATL03_ATBD_r002.pdf) format and converts them into a LAS file.

Options

**count** Maximum number of points to read. [Default: unlimited]

**override_srs** Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

**default_srs** Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

**dimensions** A JSON map with PDAL dimension names as the keys and HDF dataset paths as the values.

7.2.9 readers.i3s

Indexed 3d Scene Layer (I3S) (https://github.com/Esri/i3s-spec/blob/master/format/Indexed%203d%20Scene%20Layer%20Format%20Specification.md) is a specification created by Esri as a format for their 3D Scene Layer and scene services. The I3S reader handles RESTful webservice in an I3S file structure/format.

Example

This example will download the Autzen dataset from the ArcGIS scene server and output it to a las file. This is done through PDAL’s command line interface or through the pipeline.

```json
{
   "type": "readers.i3s",
   "filename": "https://tiles.arcgis.com/tiles/8cv2FuXuWSf0nbL/arcgis/rest/services/AUTZEN_LiDAR/SceneServer",
   "obb": {
      "center": [
         636590,
         849216,
         460
   }
}
```
pdal translate i3s://https://tiles.arcgis.com/tiles/8cv2FuXuWSfF0nbL/\arcgis/rest/services/AUTZEN_LiDAR/SceneServer \autzen.las \--readers.i3s.threads=64

Options

count  Maximum number of points to read. [Default: unlimited]

override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

filename  I3S file stored remotely. These must be prefaced with an “i3s://”.

Example remote file: pdal translate i3s://https://tiles.arcgis.com/tiles/arcgis/rest/services/AUTZEN_LiDAR/SceneServer\autzen.las

threads  This specifies the number of threads that you would like to use while reading. The default number of threads to be used is 8. This affects the speed at which files are fetched and added to the PDAL view.

Example: --readers.i3s.threads=64

obb  An oriented bounding box used to filter the data being retrieved. The obb is specified as JSON exactly as described by the I3S specification
dimensions Comma-separated list of dimensions that should be read. Specify the Esri name, rather than the PDAL dimension name.

<table>
<thead>
<tr>
<th>Esri</th>
<th>PDAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTENSITY</td>
<td>Intensity</td>
</tr>
<tr>
<td>CLASS_CODE</td>
<td>ClassFlags</td>
</tr>
<tr>
<td>FLAGS</td>
<td>Flag</td>
</tr>
<tr>
<td>RETURNS</td>
<td>NumberOfReturns</td>
</tr>
<tr>
<td>USER_DATA</td>
<td>UserData</td>
</tr>
<tr>
<td>POINT_SRC_ID</td>
<td>PointSourceId</td>
</tr>
<tr>
<td>GPS_TIME</td>
<td>GpsTime</td>
</tr>
<tr>
<td>SCAN_ANGLE</td>
<td>ScanAngleRank</td>
</tr>
<tr>
<td>RGB</td>
<td>Red</td>
</tr>
</tbody>
</table>

Example: `--readers.i3s.dimensions="returns, rgb"

min_density and max_density This is the range of density of the points in the nodes that will be selected during the read. The density of a node is calculated by the vertex count divided by the effective area of the node. Nodes do not have a uniform density across depths in the tree, so some sections may be more or less dense than others. The default values for these parameters will pull all the leaf nodes (the highest resolution).

Example: `--readers.i3s.min_density=2`
`--readers.i3s.max_density=2.5`

7.2.10 readers.ilvis2

The ILVIS2 reader read from files in the ILVIS2 format. See the product spec (https://nsidc.org/data/ilvis2) for more information.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations
The IceBridge L1S Level-2 Geolocated Surface Elevation Product ASCII text format data files contain fields as described in Table 2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVIS_LFID</td>
<td>LVIS file identification, including date and time of collection and file number. The second through sixth values in the first field represent the Modified Julian Date of data collection.</td>
<td>n/a</td>
</tr>
<tr>
<td>SHOTNUMBER</td>
<td>Laser shot assigned during collection</td>
<td>n/a</td>
</tr>
<tr>
<td>TIME</td>
<td>UTC decimal seconds of the day</td>
<td>Seconds</td>
</tr>
<tr>
<td>LONGITUDE_CENTROID</td>
<td>Refers to the centroid longitude of the corresponding LVIS Level-1B waveform.</td>
<td>Degrees east</td>
</tr>
<tr>
<td>LATITUDE_CENTROID</td>
<td>Refers to the centroid latitude of the corresponding LVIS Level-1B waveform.</td>
<td>Degrees north</td>
</tr>
<tr>
<td>ELEVATION_CENTROID</td>
<td>Refers to the centroid elevation of the corresponding LVIS Level-1B waveform.</td>
<td>Meters</td>
</tr>
<tr>
<td>LONGITUDE_LOW</td>
<td>Longitude of the lowest detected mode within the waveform</td>
<td>Degrees east</td>
</tr>
<tr>
<td>LATITUDE_LOW</td>
<td>Latitude of the lowest detected mode within the waveform</td>
<td>Degrees north</td>
</tr>
<tr>
<td>ELEVATION_LOW</td>
<td>Mean elevation of the lowest detected mode within the waveform</td>
<td>Meters</td>
</tr>
<tr>
<td>LONGITUDE_HIGH</td>
<td>Longitude of the center of the highest mode in the waveform</td>
<td>Degrees east</td>
</tr>
<tr>
<td>LATITUDE_HIGH</td>
<td>Latitude of the center of the highest mode in the waveform</td>
<td>Degrees north</td>
</tr>
<tr>
<td>ELEVATION_HIGH</td>
<td>Elevation of the center of the highest mode in the waveform</td>
<td>Meters</td>
</tr>
</tbody>
</table>

Fig. 3: Dimensions provided by the ILVIS2 reader

Example

```json
[
  {
    "type":"readers.ilvis2",
    "filename":"ILVIS2_GL2009_0414_R1401_042504.TXT",
    "metadata":"ILVIS2_GL2009_0414_R1401_042504.xml"
  },
  {
    "type":"writers.las",
    "filename":"outputfile.las"
  }
]
```
Options

filename File to read from [Required]

count Maximum number of points to read. [Default: unlimited]

override_srs Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

mapping Which ILVIS2 field type to map to X, Y, Z dimensions ‘LOW’, ‘CENTROID’, or ‘HIGH’ [Default: ‘CENTROID’]

metadata XML metadata file to coincidentally read [Optional]

7.2.11 readers.las

The LAS Reader supports reading from LAS format (http://asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) files, the standard interchange format for LIDAR data. The reader does NOT support point formats containing waveform data (4, 5, 9 and 10).

The reader also supports compressed LAS files, known as LAZ files or LASzip (http://laszip.org) files. In order to use compressed LAS (LAZ), your version of PDAL must be built with one of the two supported decompressors, LASzip (http://laszip.org) or LAZperf (https://github.com/verma/laz-perf). See the compression (page 73) option below for more information.

Note: LAS stores X, Y and Z dimensions as scaled integers. Users converting an input LAS file to an output LAS file will frequently want to use the same scale factors and offsets in the output file as existed in the input file in order to maintain the precision of the data. Use the forward option on the writers.las (page 125) to facilitate transfer of header information from source to destination LAS/LAZ files.

Note: LAS 1.4 files can contain datatypes that are actually arrays rather than individual dimensions. Since PDAL doesn’t support these datatypes, it must map them into datatypes it supports. This is done by appending the array index to the name of the datatype. For example, datatypes 11 - 20 are two dimensional array types and if a field had the name Foo for datatype 11, PDAL would create the dimensions Foo0 and Foo1 to hold the values associated with LAS field Foo. Similarly, datatypes 21 - 30 are three dimensional arrays and a field of type 21 with the name Bar would cause PDAL to create dimensions Bar0, Bar1 and Bar2. See the information on the extra bytes VLR in the LAS Specification.
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

(http://www.asprs.org/a/society/committees/standards/LAS_1_4_r13.pdf) for more information on the extra bytes VLR and array datatypes.

**Warning:** LAS 1.4 files that use the extra bytes VLR and datatype 0 will be accepted, but the data associated with a dimension of datatype 0 will be ignored (no PDAL dimension will be created).

**Default Embedded Stage**
This stage is enabled by default

**Streamable Stage**
This stage supports streaming operations

**Example**

```json
[
  {
    "type": "readers.las",
    "filename": "inputfile.las"
  },
  {
    "type": "writers.text",
    "filename": "outputfile.txt"
  }
]
```

**Options**

**filename** LAS file to read [Required]

**count** Maximum number of points to read. [Default: unlimited]

**override_srs** Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

**default_srs** Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
extra_dims Extra dimensions to be read as part of each point beyond those specified by the LAS point format. The format of the option is `<dimension_name>=<type>[, . . .]`. Any valid PDAL type (page 293) can be specified.

**Note:** The presence of an extra bytes VLR when reading a version 1.4 file or a version 1.0 - 1.3 file with use_eb_vlr set causes this option to be ignored.

use_eb_vlr If an extra bytes VLR is found in a version 1.0 - 1.3 file, use it as if it were in a 1.4 file. This option has no effect when reading a version 1.4 file. [Default: false]

compression May be set to “lazperf” or “laszip” to choose either the LazPerf decompressor or the LASzip decompressor for LAZ files. PDAL must have been built with support for the decompressor being requested. The LazPerf decompressor doesn’t support version 1 LAZ files or version 1.4 of LAS. [Default: ‘none’]

### 7.2.12 readers.matlab

The **Matlab Reader** supports readers Matlab `.mat` files. Data must be in a Matlab struct (https://www.mathworks.com/help/matlab/ref/struct.html), with field names that correspond to dimension (page 287) names. No ability to provide a name map is yet provided.

Additionally, each array in the struct should ideally have the same number of points. The reader takes its number of points from the first array in the struct. If the array has fewer elements than the first array in the struct, the point’s field beyond that number is set to zero.

**Note:** The Matlab reader requires the Mat-File API from MathWorks, and it must be explicitly enabled at compile time with the **BUILD_PLUGIN_MATLAB=ON** variable.

### Dynamic Plugin

This stage requires a dynamic plugin to operate

### Streamable Stage

This stage supports streaming operations
Example

```json
[
  {
    "type": "readers.matlab",
    "struct": "PDAL",
    "filename": "autzen.mat"
  },
  {
    "type": "writers.las",
    "filename": "output.las"
  }
]
```

Options

- **filename**  Input file name. [Required]
- **count**  Maximum number of points to read. [Default: unlimited]
- **override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
- **default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
- **struct**  Array structure name to read. [Default: ‘PDAL’]

### 7.2.13 readers.memoryview

The memoryview reader is a special stage that allows the reading of point data arranged in rows directly from memory – each point needs to have dimension data arranged at a fixed offset from a base address of the point. Before each point is read, the memoryview reader calls a function that should return the point’s base address, or a null pointer if there are no points to be read.

Note that the memoryview reader does not currently work with columnar data (data where individual dimensions are packed into arrays).
7.2.14 Usage

The memoryview reader cannot be used from the command-line. It is for use by software using the PDAL API.

After creating an instance of the memoryview reader, the user should call pushField() for every dimension that should be read from memory. pushField() takes a single argument, a MemoryViewReader::Field, that consists of a dimension name, a type and an offset from the point base address:

```c
struct Field {
    std::string m_name;
    Dimension::Type m_type;
    size_t m_offset;
};
void pushField(const Field&);
```

The user should also call setIncrementer(), a function that takes a single argument, a std::function that receives the ID of the point to be added and should return the base address of the point data, or a null pointer if there are no more points to be read.

```c
using PointIncrementer = std::function<char*(PointId)>;
void setIncrementer(PointIncrementer inc);
```

Options

None.

7.2.15 readers.mbio

The mbio reader allows sonar bathymetry data to be read into PDAL and treated as data collected using LIDAR sources. PDAL uses the MB-System (https://www.mbari.org/products/research-software/mb-system/) library to read the data and therefore supports all formats (http://www3.mbari.org/products/mbsystem/html/mbsystem_formats.html) supported by that library. Some common sonar systems are NOT supported by MB-System, notably Kongsberg, Reson and Norbit. The mbio reader reads each “beam” of data after averaging and processing by the MB-System software and stores the values for the dimensions ‘X’, ‘Y’, ‘Z’ and ‘Amplitude’. X and Y use longitude and latitude for units and the Z values are in meters (negative, being below the surface). Units for ‘Amplitude’ is not specified and may vary.
**Dynamic Plugin**

This stage requires a dynamic plugin to operate

---

**Streamable Stage**

This stage supports streaming operations

---

**Example**

This reads beams from a sonar data file and writes points to a LAS file.

```json
[
  {
    "type" : "readers.mbio",
    "filename" : "shipdata.m57",
    "format" : "MBF_EM3000RAW"
  },
  {
    "type":"writers.las",
    "filename":"outputfile.las"
  }
]
```

**Options**

- **filename**  Filename to read from [Required]
- **count**  Maximum number of points to read. [Default: unlimited]
- **override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
- **default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
- **format**  Name of number of format of file being read. See MB-System documentation for a list of all formats (http://www3.mbari.org/products/mbsystem/html/mbsystem_formats.html). [Required]
- **datatype**  Type of data to read. Either ‘multibeam’ or ‘sidescan’. [Default: ‘multibeam’]
**timegap** The maximum number of seconds that can elapse between pings before the end of the data stream is assumed. [Default: 1.0]

**speedmin** The minimum speed that the ship can be moving to before the end of the data stream is assumed. [Default: 0]

### 7.2.16 readers.mrsid

Implements MrSID 4.0 LiDAR Compressor. It requires the Lidar_DSDK (https://www.extensis.com/support/developers) to be able to decompress and read data.

#### Dynamic Plugin

This stage requires a dynamic plugin to operate

#### Example

```json
[
  {
    "type": "readers.mrsid",
    "filename": "myfile.sid"
  },
  {
    "type": "writers.las",
    "filename": "outputfile.las"
  }
]
```

#### Options

**filename** Filename to read from. [Required]

**count** Maximum number of points to read. [Default: unlimited]

**override_srs** Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

**default_srs** Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
7.2.17 readers.nitf

The NITF (http://en.wikipedia.org/wiki/National_Imagery_Transmission_Format) format is used primarily by the US Department of Defense and supports many kinds of data inside a generic wrapper. The NITF 2.1 (http://www.gwg.nga.mil/ntb/baseline/docs/2500c/index.html) version added support for LIDAR point cloud data, and the NITF file reader supports reading that data, if the NITF file supports it.

- The file must be NITF 2.1
- There must be at least one Image segment (“IM”).
- There must be at least one DES segment (http://jitc.fhu.disa.mil/cgi/nitf/registers/desreg.aspx) (“DE”) named “LIDARA”.
- Only LAS or LAZ data may be stored in the LIDARA segment

The dimensions produced by the reader match exactly to the LAS dimension names and types for convenience in file format transformation.

Note: Only LAS or LAZ data may be stored in the LIDARA segment. PDAL uses the readers.las (page 71) and writers.las (page 125) to actually read and write the data.


Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations
Example

```
[
  {
    "type": "readers.nitf",
    "filename": "mynitf.nitf"
  },
  {
    "type": "writers.las",
    "filename": "outputfile.las"
  }
]
```

Options

- **filename**  Filename to read from  [Required]
- **count**  Maximum number of points to read.  [Default: unlimited]
- **override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’  [Default: none]
- **default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’  [Default: none]
- **extra_dims**  Extra dimensions to be read as part of each point beyond those specified by the LAS point format. The format of the option is `<dimension_name>=<type>[, ...]`. Any PDAL type (page 293) can be specified.

**Note:** The presence of an extra bytes VLR when reading a version 1.4 file or a version 1.0 - 1.3 file with `use_eb_vlr` set causes this option to be ignored.

- **use_eb_vlr**  If an extra bytes VLR is found in a version 1.0 - 1.3 file, use it as if it were in a 1.4 file. This option has no effect when reading a version 1.4 file.  [Default: false]
- **compression**  May be set to “lazperf” or “laszip” to choose either the LazPerf decompressor or the LASzip decompressor for LAZ files. PDAL must have been built with support for the decompressor being requested. The LazPerf decompressor doesn’t support version 1 LAZ files or version 1.4 of LAS.  [Default: “none”]
7.2.18 readers.numpy

PDAL has support for processing data using filters.python (page 276), but it is also convenient to read data from Numpy (http://www.numpy.org/) for processing in PDAL.

Numpy (http://www.numpy.org/) supports saving files with the save method, usually with the extension .npy. As of PDAL 1.7.0, .npz files were not yet supported.

**Warning:** It is untested whether problems may occur if the versions of Python used in writing the file and for reading the file don’t match.

**Array Types**

readers.numpy supports reading data in two forms:

- As a structured array (https://docs.scipy.org/doc/numpy/user/basics.rec.html) with specified field names (from laspy (https://github.com/laspy/laspy) for example)
- As a standard array that contains data of a single type.

**Structured Arrays**

Numpy arrays can be created as structured data, where each entry is a set of fields. Each field has a name. As an example, laspy (https://github.com/laspy/laspy) provides its .points as an array of named fields:

```python
import laspy
f = laspy.file.File('test/data/autzen/autzen.las')
print (f.points[0:1])
```

```
array([[ (63608330, 84939865, 40735, 65, 73, 1, -11, 126, 7326, ...
       →245385.60820904),)]],
      dtype=[('point', [('X', '<i4'), ('Y', '<i4'), ('Z', '<i4'), ('intensity', '<u2'), ('flag_byte', 'u1'), ('raw_classification', 'u1'), ('scan_angle_rank', 'i1'), ('user_data', 'u1'), ('pt_src_id', '<u2'), ('gps_time', '<f8')])])
```

The numpy reader supports reading these Numpy arrays and mapping field names to standard PDAL dimension (page 287) names. If that fails, the reader retries by removing _, -., or space in turn. If that also fails, the array field names are used to create custom PDAL dimensions.
Standard (non-structured) Arrays

Arrays without field information contain a single datatype. This datatype is mapped to a dimension specified by the `dimension` option.

```python
f = open('./perlin.npy', 'rb')
data = np.load(f)
data.shape
(100, 100)
data.dtype
dtype('float64')
```

```
pdal info perlin.npy --readers.numpy.dimension=Intensity --readers.numpy.assign_z=4

```

```
{
    "filename": "..\test\data\plang\perlin.npy",
    "pdal_version": "1.7.1 (git-version: 399e19)",
    "stats":
    {
        "statistic": [
            {
                "average": 49.5,
                "count": 10000,
                "maximum": 99,
                "minimum": 0,
                "name": "X",
                "position": 0,
                "stddev": 28.86967866,
                "variance": 833.4583458
            },
            {
                "average": 49.5,
                "count": 10000,
                "maximum": 99,
                "minimum": 0,
                "name": "Y",
                "position": 1,
                "stddev": 28.87633116,
                "variance": 833.8425015
            },
            {
                "average": 0.01112664759,
                "count": 10000,
                "maximum": 1.0112664759,
                "minimum": 0.00998708877,
                "name": "Z",
                "position": 2,
                "stddev": 0.09998708877,
                "variance": 0.9998708877
            }
        ]
    }
}
```
"count": 10000,
"maximum": 0.5189296418,
"minimum": -0.5189296418,
"name": "Intensity",
"position": 2,
"stddev": 0.2024120437,
"variance": 0.04097063545
}
]
]

**X, Y and Z Mapping**

Unless the X, Y or Z dimension is specified as a field in a structured array, the reader will create dimensions X, Y and Z as necessary and populate them based on the position of each item of the array. Although Numpy arrays always contain contiguous, linear data, that data can be seen to be arranged in more than one dimension. A two-dimensional array will cause dimensions X and Y to be populated. A three dimensional array will cause X, Y and Z to be populated. An array of more than three dimensions will reuse the X, Y and Z indices for each dimension over three.

When reading data, X Y and Z can be assigned using row-major (C) order or column-major (Fortran) order by using the **order** option.

---

**Dynamic Plugin**

This stage requires a dynamic plugin to operate

---

**Streamable Stage**

This stage supports streaming operations
Loading Options

`readers.numpy` (page 80) supports two modes of operation - the first is to pass a reference to a `.npy` file to the `filename` argument. It will simply load it and read.

The second is to provide a reference to a `.py` script to the `filename` argument. It will then invoke the Python function specified in `module` and `function` with the `fargs` that you provide.

Loading from a Python script

A reference to a Python function that returns a Numpy array can also be used to tell `readers.numpy` (page 80) what to load. The following example itself loads a Numpy array from a Python script

Python Script

```python
import numpy as np
def load(filename):
    array = np.load(filename)
    return array
```

Command Line Invocation

Using the above Python file with its `load` function, the following `pdal info` invocation passes in the reference to the filename to load.

```
pdal info threedim.py \
    --readers.numpy.function=load \
    --readers.numpy.fargs=threedim.npy \
    --driver readers.numpy
```

Pipeline

An example `Pipeline` (page 45) definition would follow:

```
[
    {
        "function": "load",
```
Options

**filename**  npy file to read or optionally, a .py file that defines a function that returns a Numpy array using the module, function, and fargs options.  [Required]

**count**  Maximum number of points to read.  [Default: unlimited]

**override_srs**  Spatial reference to apply to the data.  Overrides any SRS in the input itself.  Can be specified as a WKT, proj.4 or EPSG string.  Can’t use with ‘default_srs’  [Default: none]

**default_srs**  Spatial reference to apply to the data if the input does not specify one.  Can be specified as a WKT, proj.4 or EPSG string.  Can’t use with ‘override_srs’  [Default: none]

**dimension**  *Dimension*  (page 287) name to map raster values

**order**  Either ‘row’ or ‘column’ to specify assigning the X,Y and Z values in a row-major or column-major order.  [Default: matches the natural order of the array.]

**module**  The Python module name that is holding the function to run.

**function**  The function name in the module to call.

**fargs**  The function args to pass to the function

---

**Note:**  The functionality of the ‘assign_z’ option in previous versions is provided with *filters.assign*  (page 204)

The functionality of the ‘x’, ‘y’, and ‘z’ options in previous versions are generally handled with the current ‘order’ option.
7.2.19 readers.obj

The **OBJ reader** reads data from files in the OBJ format. This reader constructs a mesh from the faces specified in the OBJ file, ignoring vertices that are not associated with any face. Faces, vertices, vertex normals and vertex textures are read, while all other obj elements (such as lines and curves) are ignored.

**Example**

This pipeline reads from an example OBJ file outputs the vertices as a point to a LAS file.

```
{
  "type": "readers.obj",
  "filename": "test/data/obj/1.2-with-color.obj"
},
{
  "type": "writers.las",
  "filename": "output.las",
  "scale_x": 1.0e-5,
  "scale_y": 1.0e-5,
  "scale_z": 1.0e-5,
  "offset_x": "auto",
  "offset_y": "auto",
  "offset_z": "auto"
}
```

**Options**

- **count** Maximum number of points to read. [Default: unlimited]
- **override_srs** Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
- **default_srs** Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
- **filename** File to read. [Required]
7.2.20 readers.oci

Note: The OCI reader is deprecated and will be removed in a future release.

The OCI reader is used to read data from Oracle point cloud (http://docs.oracle.com/cd/B28359_01/appdev.111/b28400/sdo_pc_pkg_ref.htm) databases.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```json
[{
   "type":"readers.oci",
   "query":"SELECT \
 l."OBJ_ID", l."BLK_ID", \
 FROM AUTZEN_BLOCKS l, \
 AUTZEN_CLOUD b \
 WHERE l.obj_id = b.id and l.obj_id in (1,2) ORDER BY l.obj_id",
   "connection":"grid/grid@localhost/orcl",
   "populate_pointsourceid":"true"
},
{
   "type":"writers.las",
   "filename":"outputfile.las"
}]
```

Options

**count**  Maximum number of points to read. [Default: unlimited]

**override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

**default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
connection  Oracle connection string to connect to database, in the form
   “user/pass@host/instance” [Required]

query  SELECT statement that returns an SDO_PC object as its first and only queried item
   [Required]

xml_schema_dump  Filename to dump the XML schema to.

populate_pointsourceid  Boolean value. If true, then add in a point cloud to every point read
   on the PointSourceId dimension. [Default: false]

### 7.2.21 readers.optech

The Optech reader reads Corrected Sensor Data (.csd) files. These files contain scan angles,
ranges, IMU and GNSS information, and boresight calibration values, all of which are
combined in the reader into XYZ points using the WGS84 reference frame.

**Default Embedded Stage**

This stage is enabled by default

**Example**

```json
[
  {
    "type": "readers.optech",
    "filename": "input.csd"
  },
  {
    "type": "writers.text",
    "filename": "outputfile.txt"
  }
]
```

**Options**

filename  csd file to read [Required]

count  Maximum number of points to read. [Default: unlimited]

override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can
   be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
**default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

### 7.2.22 readers.pcd

The **PCD Reader** supports reading from Point Cloud Data (PCD) (https://pcl-tutorials.readthedocs.io/en/latest/pcd_file_format.html) formatted files, which are used by the Point Cloud Library (PCL) (http://pointclouds.org).

#### Default Embedded Stage

This stage is enabled by default.

#### Streamable Stage

This stage supports streaming operations.

#### Example

```json
[
  {
    "type": "readers.pcd",
    "filename": "inputfile.pcd"
  },
  {
    "type": "writers.text",
    "filename": "outputfile.txt"
  }
]
```

#### Options

- **filename**  PCD file to read [Required]
- **count**  Maximum number of points to read. [Default: unlimited]
- **override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
- **default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
7.2.23 readers.pgpointcloud

The PostgreSQL Pointcloud Reader allows you to read points from a PostgreSQL database with PostgreSQL Pointcloud (https://github.com/pramsey/pointcloud) extension enabled. The Pointcloud extension stores point cloud data in tables that contain rows of patches. Each patch in turn contains a large number of spatially nearby points.

The reader pulls patches from a table, potentially sub-setting the query with a “where” clause.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```json
[
  {
    "type": "readers.pgpointcloud",
    "connection": "dbname='lidar' user='user'",
    "table": "lidar",
    "column": "pa",
    "spatialreference": "EPSG:26910",
    "where": "PC_Intersects(pa, ST_MakeEnvelope(560037.36, 5114846.45, 562667.31, 5118943.24, 26910))"
  },
  {
    "type": "writers.text",
    "filename": "output.txt"
  }
]
```

Options

**count** Maximum number of points to read. [Default: unlimited]

**override_srs** Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

**default_srs** Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

**connection** PostgreSQL connection string. In the form “host=hostname dbname=database user=username password=pw port=5432” [Required]
table  Database table to read from. [Required]
schema  Database schema to read from. [Default: public]
column  Table column to read patches from. [Default: pa]

7.2.24 readers.ply

The ply reader reads points and vertices from the polygon file format (http://paulbourke.net/dataformats/ply/), a common file format for storing three dimensional models. The ply reader can read ASCII and binary ply files.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

```json
[
  {
    "type": "readers.ply",
    "filename": "inputfile.ply"
  },
  {
    "type": "writers.text",
    "filename": "outputfile.txt"
  }
]
```

Options

filename  ply file to read [Required]
count  Maximum number of points to read. [Default: unlimited]
override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

7.2.25 readers.pts

The PTS reader reads data from Leica Cyclone PTS files. It infers dimensions from points stored in a text file.

Default Embedded Stage

This stage is enabled by default

Example Pipeline

```
[
  {
    "type":"readers.pts",
    "filename":"test.pts"
  },
  {
    "type":"writers.text",
    "filename":"outputfile.txt"
  }
]
```

Options

filename  File to read. [Required]

count  Maximum number of points to read. [Default: unlimited]

override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
7.2.26 readers.qfit

The QFIT reader read from files in the QFIT format (http://nsidc.org/data/docs/daac/icebridge/ilatm1b/docs/ReadMe.qfit.txt) originated for the Airborne Topographic Mapper (ATM) project at NASA Goddard Space Flight Center.

Default Embedded Stage

This stage is enabled by default

Example

```
[
  {
    "type": "readers.qfit",
    "filename": "inputfile.qi",
    "flip_coordinates": "false",
    "scale_z": "1.0"
  },
  {
    "type": "writers.las",
    "filename": "outputfile.las"
  }
]
```

Options

- **filename**  File to read from [Required]
- **count**  Maximum number of points to read. [Default: unlimited]
- **override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
- **default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
- **flip_coordinates**  Flip coordinates from 0-360 to -180-180 [Default: true]
- **scale_z**  Z scale. Use 0.001 to go from mm to m. [Default: 1]
- **little_endian**  Are data in little endian format? This should be automatically detected by the driver. [Optional]
7.2.27 readers.rdb

The RDB reader reads from files in the RDB format, the in-house format used by RIEGL Laser Measurement Systems GmbH (http://www.riegl.com).

**Dynamic Plugin**

This stage requires a dynamic plugin to operate

**Streamable Stage**

This stage supports streaming operations

**Installation**

To build PDAL with rdb support, set `rdb_DIR` to the path of your local rdblib installation. rdblib can be obtained from the RIEGL download pages (http://www.riegl.com/members-area/software-downloads/libraries/) with a properly enabled user account. The rdblib files do not need to be in a system-level directory, though they could be (e.g. they could be in `/usr/local`, or just in your home directory somewhere). For help building PDAL with optional libraries, see the optional library documentation (http://pdal.io/compilation/unix.html#configure-your-optional-libraries).

**Note:**

- Minimum rdblib version required to build the driver and run the tests: 2.1.6
- This driver was developed and tested on Ubuntu 17.10 using GCC 7.2.0.

**Example**

This example pipeline reads points from a RDB file and stores them in LAS format. Only points classified as “ground points” are read since option `filter` is set to “riegl.class == 2” (see line 5).

```
[  
  {  
    "type": "readers.rdb",  
    "filename": "autzen-thin-srs.rdbx",  
    "filter": "riegl.class == 2"
  }
]
```

(continues on next page)
Options

filename  Name of file to read [Required]

count  Maximum number of points to read. [Default: unlimited]

override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

filter  Point filter expression string (see RDB SDK documentation for details) [Optional] [Default: empty string (= no filter)]

extras  Read all available dimensions (true) or known PDAL dimensions only (false) [Optional] [Default: false]

Dimensions

The reader maps following default RDB point attributes to PDAL dimensions (if they exist in the RDB file):

```json
{
    "type": "writers.las",
    "filename": "autzen-thin-srs.rdbx"
}
```
<table>
<thead>
<tr>
<th>RDB attribute</th>
<th>PDAL dimension(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>riegl.id</td>
<td>Id::PointId</td>
</tr>
<tr>
<td>riegl.source_cloud_id</td>
<td>Id::OriginId</td>
</tr>
<tr>
<td>riegl.timestamp</td>
<td>Id::InternalTime</td>
</tr>
<tr>
<td>riegl.xyz</td>
<td>Id::X, Id::Y, Id::Z</td>
</tr>
<tr>
<td>riegl.intensity</td>
<td>Id::Intensity</td>
</tr>
<tr>
<td>riegl.amplitude</td>
<td>Id::Amplitude</td>
</tr>
<tr>
<td>riegl.reflectance</td>
<td>Id::Reflectance</td>
</tr>
<tr>
<td>riegl.deviation</td>
<td>Id::Deviation</td>
</tr>
<tr>
<td>riegl.pulse_width</td>
<td>Id::PulseWidth</td>
</tr>
<tr>
<td>riegl.background_radiation</td>
<td>Id::BackgroundRadiation</td>
</tr>
<tr>
<td>riegl.target_index</td>
<td>Id::ReturnNumber</td>
</tr>
<tr>
<td>riegl.target_count</td>
<td>Id::NumberOfReturns</td>
</tr>
<tr>
<td>riegl.scan_direction</td>
<td>Id::ScanDirectionFlag</td>
</tr>
<tr>
<td>riegl.scan_angle</td>
<td>Id::ScanAngleRank</td>
</tr>
<tr>
<td>riegl.class</td>
<td>Id::Classification</td>
</tr>
<tr>
<td>riegl.rgba</td>
<td>Id::Red, Id::Green, Id::Blue</td>
</tr>
<tr>
<td>riegl.surface_normal</td>
<td>Id::NormalX, Id::NormalY, Id::NormalZ</td>
</tr>
</tbody>
</table>

All other point attributes that may exist in the RDB file are ignored unless the option `extras` is set to `true`. If so, a custom dimension is defined for each additional point attribute, whereas the dimension name is equal to the point attribute name.

**Note:** Point attributes are read “as-is”, no scaling or unit conversion is done by the reader. The only exceptions are point coordinates (`riegl.xyz`) and surface normals (`rieglm.surface_normal`) which are transformed to the RDB file’s SRS by applying the matrix defined in the (optional) RDB file metadata object `rieglm.geo_tag`.

**Metadata**

The reader adds following objects to the stage’s metadata node:

**Object “database”**

Contains basic information about the RDB file such as the bounding box, number of points and the file ID.
Listing 1: Example:

```json
{
  "bounds": {
    "maximum": {
      "X": -2504493.762,
      "Y": -3846841.252,
      "Z": 4413210.394
    },
    "minimum": {
      "X": -2505882.459,
      "Y": -3848231.393,
      "Z": 4412172.548
    }
  },
  "points": 10653,
  "uuid": "637de54d-7e6b-4004-b6ab-b6bc588ec9ea"
}
```

**List “dimensions”**

List of point attribute description objects.

Listing 2: Example:

```json
[{
  "compression_options": "shuffle",
  "default_value": 0,
  "description": "Cartesian point coordinates wrt. application coordinate system (0: X, 1: Y, 2: Z)",
  "invalid_value": ",
  "length": 3,
  "maximum_value": 535000,
  "minimum_value": -535000,
  "name": "riegl.xyz",
  "resolution": 0.00025,
  "scale_factor": 1,
  "storage_class": "variable",
  "title": "XYZ",
  "unit_symbol": "m"
},
{
  "compression_options": "shuffle",
  "default_value": 0,
```

(continues on next page)
"description": "Target surface reflectance",
"invalid_value": "",
"length": 1,
"maximum_value": 327.67,
"minimum_value": -327.68,
"name": "riegl.reflectance",
"resolution": 0.01,
"scale_factor": 1,
"storage_class": "variable",
"title": "Reflectance",
"unit_symbol": "dB"
}

Details about the point attribute properties see RDB SDK documentation.

Object “metadata”

Contains one sub-object for each metadata object stored in the RDB file.

Listing 3: Example:

```json
{
    "riegl.scan_pattern": {
        "rectangular": {
            "phi_start": 45.0,
            "phi_stop": 270.0,
            "phi_increment": 0.040,
            "theta_start": 30.0,
            "theta_stop": 130.0,
            "theta_increment": 0.040,
            "program": {
                "name": "High Speed"
            }
        }
    },
    "riegl.geo_tag": {
        "crs": {
            "epsg": 4956,
            "wkt": "GEOCCS[NAD83(HARN) / Geocentric],DATUM[\n                "NAD83(HARN)",SPHEROID["GRS 1980",6378137.00,298.257222101,\n                AUTHORITY["EPSG","7019"],AUTHORITY["EPSG","6152"]],PRIMEM["Greenwich",0.0000000000000000,\n                AUTHORITY["EPSG","8901"]],UNIT["Meter",1.0000000000000000,AUTHORITY["EPSG","\n                9001"]],AXIS["X",OTHER],AXIS["Y",EAST],AXIS["Z",NORTH],\n                AUTHORITY["EPSG","4956"]]"
        }
    }
}
```

(continues on next page)
The `riegl.geo_tag` object defines the Spatial Reference System (SRS) of the file. The point coordinates are actually stored in a local coordinate system (usually horizontally leveled) which is based on the SRS. The transformation from the local system to the SRS is defined by the 4x4 matrix `pose` which is stored in row-wise order. Point coordinates (`riegl.xyz`) and surface normals (`riegl.surface_normal`) are automatically transformed to the SRS by the reader.

Details about the metadata objects see RDB SDK documentation.

**List “transactions”**

List of transaction objects describing the history of the file.

**Listing 4: Example:**

```json
[
  {
    "agent": "RDB Library 2.1.6-1677 (x86_64-windows, Apr 5 2018, 10:58:39)",
    "comments": "",
    "id": 1,
    "rdb": "RDB Library 2.1.6-1677 (x86_64-windows, Apr 5 2018, 10:58:39)",
    "settings": {
      "cache_size": 52428800,
      "chunk_size": 65536,
      "chunk_size_lod": 20,
      "compression_level": 10,
      "primary_attribute": {
        "compression_options": "shuffle",
        "default_value": 0,
        "description": "Cartesian point coordinates wrt. application coordinate system (0: X, 1: Y, 2: Z)"
      },
      "invalid_value": "",
      "length": 3,
    }
  }
]```

(continues on next page)
"maximum_value": 535000,
"minimum_value": -535000,
"name": "riegl.xyz",
"resolution": 0.00025,
"scale_factor": 1,
"storage_class": "variable",
"title": "XYZ",
"unit_symbol": "m"
}
}
"start": "2018-04-06 10:10:39.336",
"stop": "2018-04-06 10:10:39.336",
"title": "Database creation"
},
{
    "agent": "rdbconvert",
    "comments": "",
    "id": 2,
    "rdb": "RDB Library 2.1.6-1677 (x86_64-windows, Apr 5 2018,\n\t10:58:39)",
    "settings": "",
    "stop": "2018-04-06 10:10:39.380",
    "title": "Import"
},
{
    "agent": "RiSCAN PRO 64 bit v2.6.3",
    "comments": "",
    "id": 3,
    "rdb": "RDB Library 2.1.6-1677 (x86_64-windows, Apr 5 2018,\n\t10:58:39)",
    "settings": "",
    "start": "2018-04-06 10:10:41.666",
    "stop": "2018-04-06 10:10:41.666",
    "title": "Meta data saved"
}]
7.2.28 readers.rxp

The **RXP reader** read from files in the RXP format, the in-house streaming format used by RIEGL Laser Measurement Systems GmbH (http://www.riegl.com).

**Warning:** This software has not been developed by RIEGL, and RIEGL will not provide any support for this driver. Please do not contact RIEGL with any questions or issues regarding this driver. RIEGL is not responsible for damages or other issues that arise from use of this driver. This driver has been tested against RiVLib version 1.39 on a Ubuntu 14.04 using gcc43.

---

**Dynamic Plugin**

This stage requires a dynamic plugin to operate.

---

**Streamable Stage**

This stage supports streaming operations.

---

**Installation**

To build PDAL with rxp support, set RiVLib_DIR to the path of your local RiVLib installation. RiVLib can be obtained from the [RIEGL download pages](http://www.riegl.com/members-area/software-downloads/libraries/) with a properly enabled user account. The RiVLib files do not need to be in a system-level directory, though they could be (e.g. they could be in `/usr/local`, or just in your home directory somewhere). For help building PDAL with optional libraries, see the optional library documentation.

---

**Example**

This example rescales the points, given in the scanner’s own coordinate system, to values that can be written to a las file. Only points with a valid gps time, as determined by a pps pulse, are read from the rxp, since the `sync_to_pps` option is “true”. Reflectance values are mapped to intensity values using sensible defaults.

```json
{
    "type": "readers.rxp",
    "filename": "120304_204030.rxp",
    "sync_to_pps": true,
    "reflectance": 1.0,
    "intensity": 0.0
}
```

(continues on next page)
"sync_to_pps": "true",
"reflectance_as_intensity": "true"
],
{
    "type": "writers.las",
    "filename": "outputfile.las",
    "discard_high_return_numbers": "true"
}
]

We set the `discard_high_return_numbers` option to `true` on the `writers.las` (page 125). RXP files can contain more returns per shot than is supported by las, and so we need to explicitly tell the las writer to ignore those high return number points. You could also use `filters.python` (page 276) to filter those points earlier in the pipeline.

**Options**

`filename` File to read from, or rdtp URI for network-accessible scanner. [Required]

`count` Maximum number of points to read. [Default: unlimited]

`override_srs` Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

`default_srs` Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

`rdtp` Boolean to switch from file-based reading to RDTP-based. [Default: false]

`sync_to_pps` If “true”, ensure all incoming points have a valid pps timestamp, usually provided by some sort of GPS clock. If “false”, use the scanner’s internal time. [Default: true]

`reflectance_as_intensity` If “true”, in addition to storing reflectance values directly, also stores the values as Intensity by mapping the reflectance values in the range from `min_reflectance` to `max_reflectance` to the range 0-65535. Values less than `min_reflectance` are assigned the value 0. Values greater `max_reflectance` are assigned the value 65535. [Default: true]

`min_reflectance` The low end of the reflectance-to-intensity map. [Default: -25.0]

`max_reflectance` The high end of the reflectance-to-intensity map. [Default: 5.0]
7.2.29 readers.sbet

The SBET reader read from files in the SBET format, used for exchange data from inertial measurement units (IMUs). SBET files store angles as radians, but by default this reader converts all angle-based measurements to degrees. Set angles_as_degrees to false to disable this conversion.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

```json
[  "sbetfile.sbet",  "output.las"
]
```

Options

filename  File to read from [Required]

count  Maximum number of points to read. [Default: unlimited]

override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

angles_as_degrees  Convert all angles to degrees. If false, angles are read as radians. [Default: true]
7.2.30 readers.sqlite

The SQLite Reader allows you to read data stored in a SQLite database (https://sqlite.org/) using a scheme that PDAL wrote using the writers.sqlite (page 144) writer. The SQLite driver stores data in tables that contain rows of patches. Each patch contains a number of spatially contiguous points.

Dynamic Plugin

This stage requires a dynamic plugin to operate.

Example

```
{
   "type": "readers.sqlite",
   "connection": "inputfile.sqlite",
   "query": "SELECT b.schema, l.cloud, l.block_id, l.num_points,  
             l.bbox, l.extent, l.points, b.cloud  
             FROM simple_blocks l, simple_cloud b  
             WHERE l.cloud = b.cloud and l.cloud in (1)  
             order by l.cloud",
   "query": "writers.sqlite",
   "filename": "outputfile.las"
}
```

Options

**query**  SQL statement that selects a schema XML, cloud id, bbox, and extent [Required]

**count**  Maximum number of points to read. [Default: unlimited]

**override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

**default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]
7.2.31 readers.slpk

Scene Layer Packages (SLPK) ([https://github.com/Esri/i3s-spec/blob/master/format/Indexed%203d%20Scene%20Layer%20Format%20Specification.md#_8_1](https://github.com/Esri/i3s-spec/blob/master/format/Indexed%203d%20Scene%20Layer%20Format%20Specification.md#_8_1)) is a specification created by Esri as a format for their 3D Scene Layer and scene services. SLPK is a format that allows you to package all the necessary I3S (page 67) files together and store them locally rather than find information through REST.

Example

This example will unarchive the slpk file, store it in a temp directory, and traverse it. The data will be output to a las file. This is done through PDAL’s command line interface or through the pipeline.

```json
{
    "type": "readers.slpk",
    "filename": "PDAL/test/data/i3s.SMALL_AUTZEN_LAS_All.slpk",
    "obb": {
        "center": [636590, 849216, 460],
        "halfSize": [590, 281, 60],
        "quaternion": [0, 0, 0, 1]
    }
}
```

```
pdal translate PDAL/test/data/i3s.SMALL_AUTZEN_LAS_All.slpk autzen.
```

→ las
Options

filename  SLPK file must have a file extension of .slpk. Example: `pdal translate /PDAL/test/data/i3s/SMALL_AUTZEN_LAS_ALL.slpk output.las`

count  Maximum number of points to read. [Default: unlimited]

override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

obb  An oriented bounding box used to filter the data being retrieved. The obb is specified as JSON exactly as described by the I3S specification (https://github.com/Esri/i3s-spec/blob/master/docs/2.0/obb.cmn.md).

dimensions  Comma-separated list of dimensions that should be read. Specify the Esri name, rather than the PDAL dimension name.

<table>
<thead>
<tr>
<th>Esri</th>
<th>PDAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTENSITY</td>
<td>Intensity</td>
</tr>
<tr>
<td>CLASS_CODE</td>
<td>ClassFlags</td>
</tr>
<tr>
<td>FLAGS</td>
<td>Flag</td>
</tr>
<tr>
<td>RETURNS</td>
<td>NumberOfReturns</td>
</tr>
<tr>
<td>USER_DATA</td>
<td>UserData</td>
</tr>
<tr>
<td>POINT_SRC_ID</td>
<td>PointSourceId</td>
</tr>
<tr>
<td>GPS_TIME</td>
<td>GpsTime</td>
</tr>
<tr>
<td>SCAN_ANGLE</td>
<td>ScanAngleRank</td>
</tr>
<tr>
<td>RGB</td>
<td>Red</td>
</tr>
</tbody>
</table>

Example: `--readers.slpk.dimensions="rgb, intensity"

min_density and max_density  This is the range of density of the points in the nodes that will be selected during the read. The density of a node is calculated by the vertex count divided by the effective area of the node. Nodes do not have a uniform density across depths in the tree, so some sections may be more or less dense than others. Default values for these parameters will select all leaf nodes (the highest resolution).

Example: `--readers.slpk.min_density=2`  
`--readers.slpk.max_density=2.5`
7.2.32 readers.terrasolid

The Terrasolid Reader loads points from Terrasolid (https://www.terrasolid.com/home.php) files (.bin). It supports both Terrasolid format 1 and format 2.

Example

```
[  
  {  
    "type": "readers.terrasolid",  
    "filename": "autzen.bin"  
  },  
  {  
    "type": "writers.las",  
    "filename": "output.las"  
  }  
]
```

Options

- **filename** Input file name [Required]
- **count** Maximum number of points to read. [Default: unlimited]
- **override_srs** Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
- **default_srs** Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

7.2.33 readers.text

The text reader reads data from ASCII text files. Each point is represented in the file as a single line. Each line is expected to be divided into a number of fields by a separator. Each field represents a value for a point’s dimension. Each value needs to be formatted (http://en.cppreference.com/w/cpp/string/basic_string/stof) properly for C++ language double-precision values.

The text reader expects a header line to indicate the dimensions are in each subsequent line. There are two types of header lines.
**Quoted dimension names**

When the first character of the header is a double quote, each dimension name is assumed to be surrounded by double quotes. A single separator character is expected between the dimension names (spaces are stripped). If no separator character is found, a space is assumed. You can set the separator (page 109) character if it differs from that in the header. Note that PDAL requires dimension names that consist only of alphabetic characters and underscores. Edit the header line or use the header (page 109) option to set the dimension names to ones that PDAL understands.

**Unquoted dimension names**

The first non alpha-numeric character encountered is treated as a separator between dimension names. The separator in the header line can be overridden by the separator (page 109) option.

Each line in the file must contain the same number of fields as indicated by dimension names in the header. Spaces are generally ignored in the input unless used as a separator. When a space character is used as a separator, any number of consecutive spaces are treated as single space and leading/trailing spaces are ignored.

Blank lines are ignored after the header line is read.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Streamable Stage**

This stage supports streaming operations

---

**Example Input File**

This input file contains X, Y and Z value for 10 points.

```
X, Y, Z
289814.15, 4320978.61, 170.76
289814.64, 4320978.84, 170.76
289815.12, 4320979.06, 170.75
289815.60, 4320979.28, 170.74
289816.08, 4320979.50, 170.68
289816.56, 4320979.71, 170.66
289817.03, 4320979.92, 170.63
```

(continues on next page)
Example #1

```
[
  {
    "type": "readers.text",
    "filename": "inputfile.txt"
  },
  {
    "type": "writers.text",
    "filename": "outputfile.txt"
  }
]
```

Example #2

This reads the data in the input file as Red, Green and Blue instead of as X, Y and Z.

```
[
  {
    "type": "readers.text",
    "filename": "inputfile.txt",
    "header": "Red, Green, Blue",
    "skip": 1
  },
  {
    "type": "writers.text",
    "filename": "outputfile.txt"
  }
]
```
## Options

**filename**  text file to read [Required]

**count**  Maximum number of points to read. [Default: unlimited]

**override_srs**  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

**default_srs**  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

**header**  String to use as the file header. All lines in the file are assumed to be records containing point data unless skipped with the `skip` (page 109) option. [Default: None]

**separator**  Separator character to override that found in header line. [Default: None]

**skip**  Number of lines to ignore at the beginning of the file. [Default: 0]

### 7.2.34 readers.tiledb

Implements TileDB (https://tiledb.io) 1.4.1+ storage.

#### Dynamic Plugin

This stage requires a dynamic plugin to operate

#### Streamable Stage

This stage supports streaming operations

### Example

```json
[
  {
    "type": "readers.tiledb",
    "array_name": "my_array"
  },
  {
    "type": "writers.las",
    "filename": "outputfile.las"
  }
]
```

7.2.  Readers 109
Options

array_name  TileDB (https://tiledb.io) array to read from. [Required]
config_file  TileDB (https://tiledb.io) configuration file [Optional]
chunk_size  Size of chunks to read from TileDB array [Optional]
stats  Dump query stats to stdout [Optional]
bbox3d  TileDB subarray to read in format ([minx, maxx], [miny, maxy], [minz, maxz]) [Optional]
count  Maximum number of points to read. [Default: unlimited]
override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]
default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

7.2.35 readers.tindex

A GDAL tile index (http://www.gdal.org/gdaltindex.html) is an OGR (http://gdal.org/ogr/)-readable data source of boundary information. PDAL provides a similar concept for PDAL-readable point cloud data. You can use the tindex (page 37) application to generate tile index files in any format that OGR (http://gdal.org/ogr/) supports writing. Once you have the tile index, you can then use the tindex reader to automatically merge and query the data described by the tiles.

Default Embedded Stage

This stage is enabled by default

Basic Example

Given a tile index that was generated with the following scenario:

```
pdal tindex index.sqlite \
   "~/Users/hobu/dev/git/pdal/test/data/las/interesting.las" \
   -f "SQLite" \
   --lyr_name "pdal" \
   --t_srs "EPSG:4326"
```

Use the following pipeline (page 45) example to read and automatically merge the data.
{  
  {    
    "type": "readers.tindex",
    "filter_srs": "+proj=lcc +lat_1=43 +lat_2=45.5 +lat_0=41.75
    +lon_0=-120.5 +x_0=399999.9999999999 +y_0=0 +ellps=GRS80 +units=ft
    +no_defs",
    "filename": "index.sqlite",
    "where": "location LIKE '%interesting.las%'",
    "wkt": "POLYGON ((635629.85000000 848999.70000000, 635629.85000000 853535.43000000, 638982.55000000 853535.43000000, 638982.55000000 848999.70000000, 635629.85000000 848999.70000000, 635629.85000000 848999.70000000, 635629.85000000 848999.70000000, 635629.85000000 848999.70000000))"
  },
  {    
    "type": "writers.las",
    "filename": "outputfile.las"
  }
}

Options

data type  OGROpen’able raster file to read [Required]

count  Maximum number of points to read. [Default: unlimited]

override_srs  Spatial reference to apply to the data. Overrides any SRS in the input itself. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘default_srs’ [Default: none]

default_srs  Spatial reference to apply to the data if the input does not specify one. Can be specified as a WKT, proj.4 or EPSG string. Can’t use with ‘override_srs’ [Default: none]

lyr_name  The OGR layer name for the data source to use to fetch the tile index information.

tsrs_column  The column in the layer that provides the SRS information for the file. Use this if you wish to override or set coordinate system information for files.

tindex_name  The column name that defines the file location for the tile index file. [Default: location]

sql  OGR SQL (http://www.gdal.org/ogr_sql.html) to use to define the tile index layer.

bounds  A 2D box to pre-filter the tile index. If it is set, it will override any wkt (page 111) option.

wkt  A geometry to pre-filter the tile index using OGR.

t_srs  Reproject the layer SRS, otherwise default to the tile index layer’s SRS. [Default: “EPSG:4326”]
filter_srs  Transforms any wkt (page 111) or bounds (page 111) option to this coordinate system before filtering or reading data. [Default: “EPSG:4326”]

where  OGR SQL (http://www.gdal.org/ogr_sql.html) filter clause to use on the layer. It only works in combination with tile index layers that are defined with lyr_name (page 111)

dialect  OGR SQL (http://www.gdal.org/ogr_sql.html) dialect to use when querying tile index layer [Default: OGRSQL]

readers.bpf (page 54)  Read BPF files encoded as version 1, 2, or 3. BPF is an NGA specification for point cloud data.

readers.buffer (page 55)  Special stage that allows you to read data from your own PointView rather than fetching data from a specific reader.

readers.ept (page 56)  Used for reading Entwine Point Tile (https://entwine.io) format.

readers.e57 (page 59)  Read point clouds in the E57 format.

readers.faux (page 60)  Used for testing pipelines. It does not read from a file or database, but generates synthetic data to feed into the pipeline.

readers.gdal (page 62)  Read GDAL readable raster data sources as point clouds.

readers.geowave (page 63)  Read point cloud data from Accumulo.

readers.hdf (page 65)  Read data from files in the HDF5 format.

readers.i3s (page 67)  Read data stored in the Esri I3S format. The data is read from an appropriate server.

readers.ilvis2 (page 69)  Read from files in the ILVIS2 format.

readers.las (page 71)  Read ASPRS LAS versions 1.0 - 1.4. Does not support point formats containing waveform data. LASzip support is also enabled through this driver if LASzip or LAZperf are found during compilation.

readers.matlab (page 73)  Read point cloud data from MATLAB .mat files where dimensions are stored as arrays in a MATLAB struct.

readers.mbio (page 75)  Read sonar bathymetry data from formats supported by the MB-System library.

readers.memoryview (page 74)  Read data from memory where dimension data is arranged in rows. For use only with the PDAL API.

readers.mrsid (page 77)  Read data compressed by the MrSID 4.0 LiDAR Compressor. Requires the LizardTech Lidar_DSDK.

readers.nitf (page 78)  Read point cloud data (LAS or LAZ) wrapped in NITF 2.1 files.

readers.numpy (page 80)  Read point cloud data from Numpy .npy files.

readers.obj (page 85)  Read a mesh from an OBJ file.
**readers.oci** (page 86) Read data from Oracle point cloud databases. [deprecated]

*readers.optech* (page 87) Read Optech Corrected Sensor Data (.csd) files.

*readers.pcd* (page 88) Read files in the PCD format.

*readers.pgpointcloud* (page 89) Read point cloud data from a PostgreSQL database with the PostgreSQL Pointcloud extension enabled.

*readers.ply* (page 90) Read points and vertices from either ASCII or binary PLY files.

*readers.pts* (page 91) Read data from Leica Cyclone PTS files.

*readers.qfit* (page 92) Read data in the QFIT format originated for NASA’s Airborne Topographic Mapper project.

*readers.rxp* (page 100) Read data in the RXP format, the in-house streaming format used by RIEGL. The reader requires a copy of RiVLib during compilation.

*readers.rdb* (page 93) Read data in the RDB format, the in-house database format used by RIEGL. The reader requires a copy of rdblib during compilation and usage.

*readers.sbet* (page 102) Read the SBET format.

*readers.sqlite* (page 103) Read data stored in a SQLite database.

*readers.slpk* (page 104) Read data stored in an Esri SLPK file.

*readers.terrasolid* (page 106) TerraSolid Reader

*readers.text* (page 106) Read point clouds from ASCII text files.

*readers.tiledb* (page 109) Read point cloud data from a TileDB instance.

*readers.tindex* (page 110) The tindex (tile index) reader allows you to automatically merge and query data described in tile index files that have been generated using the PDAL tindex command.

### 7.3 Writers

Writers consume data provided by *Readers* (page 54). Some writers can consume any dimension type, while others only understand fixed dimension names.

**Note:** PDAL predefined dimension names can be found in the dimension registry: *Dimensions* (page 287)
7.3.1 writers.bpf

BPF is an NGA specification (https://nsgreg.nga.mil/doc/view?i=4202) for point cloud data. The PDAL BPF Writer only supports writing of version 3 BPF format files.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

```json
[
  {
    "type": "readers.bpf",
    "filename": "inputfile.las"
  },
  {
    "type": "writers.bpf",
    "filename": "outputfile.bpf"
  }
]
```

Options

- **filename** BPF file to write. The writer will accept a filename containing a single placeholder character (‘#’). If input to the writer consists of multiple PointViews, each will be written to a separate file, where the placeholder will be replaced with an incrementing integer. If no placeholder is found, all PointViews provided to the writer are aggregated into a single file for output. Multiple PointViews are usually the result of using filters.splitter (page 256), filters.chipper (page 252) or filters.divider (page 255). [Required]

- **compression** This option can be set to true to cause the file to be written with Zlib compression as described in the BPF specification. [Default: false]

- **format** Specifies the format for storing points in the file. [Default: dim]
  - **dim**: Dimension-major (non-interleaved). All data for a single dimension are stored contiguously.
• point: Point-major (interleaved). All data for a single point are stored contiguously.

• byte: Byte-major (byte-segregated). All data for a single dimension are stored contiguously, but bytes are arranged such that the first bytes for all points are stored contiguously, followed by the second bytes of all points, etc. See the BPF specification for further information.

**bundledfile** Path of file to be written as a bundled file (see specification). The path part of the filespec is removed and the filename is stored as part of the data. This option can be specified as many times as desired.

**header_data** Base64-encoded data that will be decoded and written following the standard BPF header.

**coord_id** The coordinate ID (UTM zone) of the data. Southern zones take negative values. A value of 0 indicates cartesian instead of UTM coordinates. A value of ‘auto’ will attempt to set the UTM zone from a suitable spatial reference, or set to 0 if no such SRS is set. [Default: 0]

**scale_x, scale_y, scale_z** Scale to be divided from the X, Y and Z nominal values, respectively, after the offset has been applied. The special value “auto” can be specified, which causes the writer to select a scale to set the stored values of the dimensions to range from [0, 2147483647]. [Default: .01]

Note: written value = (nominal value - offset) / scale.

**offset_x, offset_y, offset_z** Offset to be subtracted from the X, Y and Z nominal values, respectively, before the value is scaled. The special value “auto” can be specified, which causes the writer to set the offset to the minimum value of the dimension. [Default: auto]

Note: written value = (nominal value - offset) / scale.

---

**Note:** Because BPF data is always stored in UTM, the XYZ offsets are set to “auto” by default. This is to avoid truncation of the decimal digits (which may occur with offsets left at 0).

**output_dims** If specified, limits the dimensions written for each point. Dimensions are listed by name and separated by commas. X, Y and Z are required and must be explicitly listed.
7.3.2 writers.ept_addon

The EPT Addon Writer supports writing additional dimensions to Entwine Point Tile (https://entwine.io/entwine-point-tile.html) datasets. The EPT addon writer may only be used in a pipeline with an EPT reader (page 56), and it creates additional attributes for an existing dataset rather than creating a brand new one.

The addon dimensions created by this writer are stored independently from the corresponding EPT dataset, therefore write-access to the EPT resource itself is not required to create and use addon dimensions.

Default Embedded Stage

This stage is enabled by default

Example

This example downloads the Autzen dataset (10M points) and runs the SMRF filter (page 155), which populates the Classification dimension with ground values, and writes the resulting attribute to an EPT addon dataset on the local filesystem.

```json
[
  {
    "type": "readers.ept",
    "filename": "http://na.entwine.io/autzen/ept.json"
  },
  {
    "type": "filters.assign",
    "assignment": "Classification[:] = 0"
  },
  {
    "type": "filters.smrf"
  },
  {
    "type": "writers.ept_addon",
    "addons": {
      "~/entwine/addons/autzen/smrf": "Classification"
    }
  }
]
```

And here is a follow-up example of reading this dataset with the EPT reader (page 56) with the created addon overwriting the Classification value. The output is then written to a single file with the LAS writer (page 125).
This is an example of using multiple mappings in the `addons` option to apply a new color scheme with `filters.colorinterp` (page 175) mapping the Red, Green, and Blue dimensions to new values.

The following pipeline will read the data with the new colors:

(continues on next page)
Options

addons  A JSON object whose keys represent output paths for each addon dimension, and whose corresponding values represent the attributes to be written to these addon dimensions. [Required]

Note: The addons option is reversed between the EPT reader and addon-writer: in each case, the right-hand side represents an assignment to the left-hand side. In the writer, the dimension value is assigned to an addon path. In the reader, the addon path is assigned to a dimension.

threads  Number of worker threads used to write EPT addon data. A minimum of 4 will be used no matter what value is specified.

7.3.3 writers.e57

The E57 Writer supports writing to E57 files.

The writer supports E57 files with Cartesian point clouds.

Note: E57 files can contain multiple point clouds stored in a single file. The writer will only write a single cloud per file.

Note: Spherical format points are not supported.

Note: The E57 cartesianInvalidState dimension is mapped to the Omit PDAL dimension. A range filter can be used to filter out the invalid points.

Dynamic Plugin
This stage requires a dynamic plugin to operate

Streamable Stage
This stage supports streaming operations

Example

```json
[
  {
    "type": "readers.las",
    "filename": "inputfile.las"
  },
  {
    "type": "writers.e57",
    "filename": "outputfile.e57",
    "doublePrecision": false
  }
]
```

Options

- **filename**  E57 file to write [Required]
- **doublePrecision**  Use double precision for storage (false by default).

7.3.4 writers.gdal

The GDAL writer creates a raster from a point cloud using an interpolation algorithm. Output is produced using GDAL (http://gdal.org) and can use any driver that supports creation of rasters (http://www.gdal.org/formats_list.html). A data_type (page 121) can be specified for the raster (double, float, int32, etc.). If no data type is specified, the data type with the largest range supported by the driver is used.

The technique used to create the raster is a simple interpolation where each point that falls within a given radius (page 121) of a raster cell center potentially contributes to the raster’s value. If no radius is provided, it is set to the product of the resolution (page 121) and the square root of two. If a circle with the provided radius doesn’t encompass the entire cell, it is possible that some points will not be considered at all, including those that may be within the bounds of the raster cell.
The GDAL writer creates rasters using the data specified in the $dimension$ (page 122) option (defaults to $Z$). The writer creates up to six rasters based on different statistics in the output dataset. The order of the layers in the dataset is as follows:

**min**  Give the cell the minimum value of all points within the given radius.

**max**  Give the cell the maximum value of all points within the given radius.

**mean**  Give the cell the mean value of all points within the given radius.

**idw**  Cells are assigned a value based on Shepard’s inverse distance weighting (https://en.wikipedia.org/wiki/Inverse_distance_weighting) algorithm, considering all points within the given radius.

**count**  Give the cell the number of points that lie within the given radius.

**stdev**  Give the cell the population standard deviation of the points that lie within the given radius.

If no points fall within the circle about a raster cell, a secondary algorithm can be used to attempt to provide a value after the standard interpolation is complete. If the $window\_size$ (page 122) option is non-zero, the values of a square of rasters surrounding an empty cell is applied using inverse distance weighting of any non-empty cells. The value provided for $window\_size$ is the maximum horizontal or vertical distance that a donor cell may be in order to contribute to the subject cell (A $window\_size$ of 1 essentially creates a 3x3 array around the subject cell. A $window\_size$ of 2 creates a 5x5 array, and so on.)

Cells that have no value after interpolation are given a value specified by the $nodata$ (page 121) option.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Streamable Stage**

This stage supports streaming operations
Basic Example

This pipeline reads the file autzen_trim.las and creates a GeoTIFF dataset called outputfile.tif. Since output_type isn’t specified, it creates six raster bands (“min”, “max”, “mean”, “idx”, “count” and “stdev”) in the output dataset. The raster cells are 10x10 and the radius used to locate points whose values contribute to the cell value is 14.14.

```
[   
  "pdal/test/data/las/autzen_trim.las",
  
  {  
    "resolution": 10,
    "radius": 14.14,
    "filename":"outputfile.tif"
  }
]
```

Options

**filename** Name of output file. The writer will accept a filename containing a single placeholder character (#). If input to the writer consists of multiple PointViews, each will be written to a separate file, where the placeholder will be replaced with an incrementing integer. If no placeholder is found, all PointViews provided to the writer are aggregated into a single file for output. Multiple PointViews are usually the result of using `filters.splitter` (page 256), `filters.chipper` (page 252) or `filters.divider` (page 255). [Required]

**resolution** Length of raster cell edges in X/Y units. [Required]

**radius** Radius about cell center bounding points to use to calculate a cell value. [Default: \( \text{resolution} \) (page 121) * \( \sqrt{2} \)]

**power** Exponent of the distance when computing IDW. Close points have higher significance than far points. [Default: 1.0]

**gdaldriver** GDAL code of the GDAL driver (http://www.gdal.org/formats_list.html) to use to write the output. [Default: “GTiff”]

**gdalopts** A list of key/value options to pass directly to the GDAL driver. The format is name=value,name=value,… The option may be specified any number of times.

**Note:** The INTERLEAVE GDAL driver option is not supported. writers.gdal always uses BAND interleaving.

**data_type** The data type (page 293) to use for the output raster. Many GDAL drivers only support a limited set of output data types. [Default: depends on the driver]
**nodata**  The value to use for a raster cell if no data exists in the input data with which to compute an output cell value. [Default: depends on the *data_type* (page 121). -9999 for double, float, int and short, 9999 for unsigned int and unsigned short, 255 for unsigned char and -128 for char]

**output_type**  A comma separated list of statistics for which to produce raster layers. The supported values are “min”, “max”, “mean”, “idw”, “count”, “stdev” and “all”. The option may be specified more than once. [Default: “all”]

**window_size**  The maximum distance from a donor cell to a target cell when applying the fallback interpolation method. See the stage description for more information. [Default: 0]

**dimension**  A dimension name to use for the interpolation. [Default: “Z”]

**bounds**  The bounds of the data to be written. Points not in bounds are discarded. The format is ([minx, maxx],[miny, maxy]). [Optional]

**origin_x**  X origin (lower left corner) of the grid. [Default: None]

**origin_y**  Y origin (lower left corner) of the grid. [Default: None]

**width**  Number of cells in the X direction. [Default: None]

**height**  Number of cells in the Y direction. [Default: None]

---

**Note:** You may use the ‘bounds’ option, or ‘origin_x’, ‘origin_y’, ‘width’ and ‘height’, but not both.

---

### 7.3.5 *writers.geowave*

The GeoWave writer uses GeoWave (https://github.com/locationtech/geowave) to write to Accumulo. GeoWave entries are stored using EPSG:4326 (http://epsg.io/4326/).

**Dynamic Plugin**

This stage requires a dynamic plugin to operate
Example

```json
[
  {
    "type": "readers.qfit",
    "filename": "inputfile.qi",
    "flip_coordinates": "false",
    "scale_z": "1.0"
  },

  {
    "type": "writers.geowave",
    "zookeeper_url": "zookeeper1:2181,zookeeper2:2181,zookeeper3:2181",
    "instance_name": "GeoWave",
    "username": "user",
    "password": "pass",
    "table_namespace": "PDAL_Table",
    "feature_type_name": "PDAL_Point",
    "data_adapter": "FeatureCollectionDataAdapter",
    "points_per_entry": "5000u"
  }
]
```

Options

**zookeeper_url** The comma-delimited URLs for all zookeeper servers, this will be directly used to instantiate a ZookeeperInstance. [Required]

**instance_name** The zookeeper instance name, this will be directly used to instantiate a ZookeeperInstance. [Required]

**username** The username for the account to establish an Accumulo connector. [Required]

**password** The password for the account to establish an Accumulo connector. [Required]

**table_namespace** The table name to be used when interacting with GeoWave. [Required]

**feature_type_name** The feature type name to be used when interacting GeoWave. [Default: PDAL_Point]

**data_adapter** FeatureCollectionDataAdapter stores multiple points per Accumulo entry. FeatureDataAdapter stores a single point per Accumulo entry. [Default: FeatureCollectionDataAdapter]

**points_per_entry** Sets the maximum number of points per Accumulo entry when using FeatureCollectionDataAdapter. [Default: 5000u]
7.3.6 writers.gltf

GLTF is a file format specification (https://www.khronos.org/gltf/) for 3D graphics data. If a mesh has been generated for a PDAL point view, the **GLTF Writer** will produce simple output in the GLTF format. PDAL does not currently support many of the attributes that can be found in a GLTF file. This writer creates a *binary* GLTF.

**Default Embedded Stage**

This stage is enabled by default

**Example**

```json
[
  "infile.las",
  {
    "type": "filters.poisson",
    "depth": 12
  },
  {
    "type": "writers.gltf",
    "filename": "output.glb",
    "red": 0.8,
    "metallic": 0.5
  }
]
```

**Options**

- **filename** Name of the GLTF (.glb) file to be written. [Required]
- **metallic** The metallic factor of the faces. [Default: 0]
- **roughness** The roughness factor of the faces. [Default: 0]
- **red** The red component of the color applied to the faces. [Default: 0]
- **green** The green component of the color applied to the faces. [Default: 0]
- **blue** The blue component of the color applied to the faces. [Default: 0]
- **alpha** The alpha component to be applied to the faces. [Default: 1.0]
- **double_sided** Whether the faces are colored on both sides, or just the side visible from the initial observation point (positive normal vector). [Default: false]
7.3.7 writers.las

The LAS Writer supports writing to LAS format (http://asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) files, the standard interchange file format for LIDAR data.

**Warning:** Scale/offset are not preserved from an input LAS file. See below for information on the scale/offset options and the `forward` (page 126) option.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Streamable Stage**

This stage supports streaming operations

---

**VLRs**

VLRs can be created by providing a JSON node called `vlrs` with objects containing `user_id` and `data` (or `filename`) items.

```json
[
  {
    "type": "readers.las",
    "filename": "inputfile.las"
  },
  {
    "type": "writers.las",
    "vlrs": [{
      "description": "A description under 32 bytes",
      "record_id": 42,
      "user_id": "hobu",
      "data": "dGhpcyBpcyBzb21lIHRleHQ=
    },
    {
      "description": "A description under 32 bytes",
      "record_id": 43,
      "user_id": "hobu",
      "filename": "path-to-my-file.input"
    }
  ]
]
```

(continues on next page)
"filename":"outputfile.las"
]

Note: *data* or *filename* must be specified. Data must always be provided as base64 encoded strings, but content of *filename* is expected to be raw data.

Example

```
[
  {
    "type":"readers.las",
    "filename":"inputfile.las"
  },
  {
    "type":"writers.las",
    "filename":"outputfile.las"
  }
]
```

Options

**filename** Output filename. The writer will accept a filename containing a single placeholder character (#). If input to the writer consists of multiple PointViews, each will be written to a separate file, where the placeholder will be replaced with an incrementing integer. If no placeholder is found, all PointViews provided to the writer are aggregated into a single file for output. Multiple PointViews are usually the result of using *filters.splitter* (page 256), *filters.chipper* (page 252) or *filters.divider* (page 255). [Required]

**forward** List of header fields whose values should be preserved from a source LAS file. The option can be specified multiple times, which has the same effect as listing values separated by a comma. The following values are valid: major_version, minor_version, datasource_id, filesource_id, global_encoding, project_id, system_id, software_id, creation_doy, creation_year, scale_x, scale_y, scale_z, offset_x, offset_y, offset_z. In addition, the special value *header* can be specified, which is equivalent to specifying all the values EXCEPT the scale and offset values. Scale and offset values can be forwarded as a group by using the special values *scale* and *offset* respectively. The special value *all* is equivalent to specifying header, scale, offset and vlr (see below). If a header option is specified explicitly, it will override any forwarded header value. If a
LAS file is the result of multiple LAS input files, the header values to be forwarded must match or they will be ignored and a default will be used instead.

VLRs can be forwarded by using the special value vlr. VLRs containing the following User IDs are NOT forwarded: LASF_Projection, liblas, laszip encoded. VLRs with the User ID LASF_Spec and a record ID other than 0 or 3 are also not forwarded. These VLRs are known to contain information regarding the formatting of the data and will be rebuilt properly in the output file as necessary. Unlike header values, VLRs from multiple input files are accumulated and each is written to the output file. Forwarded VLRs may contain duplicate User ID/Record ID pairs.

**minor_version**  All LAS files are version 1, but the minor version (0 - 4) can be specified with this option. [Default: 2]

**software_id**  String identifying the software that created this LAS file. [Default: PDAL version num (build num)]”

**creation_doy**  Number of the day of the year (January 1 == 0, Dec 31 == 365) this file is being created.

**creation_year**  Year (Gregorian) this file is being created.

**dataformat_id**  Controls whether information about color and time are stored with the point information in the LAS file. [Default: 3]

- 0 == no color or time stored
- 1 == time is stored
- 2 == color is stored
- 3 == color and time are stored
- 4 [Not Currently Supported]
- 5 [Not Currently Supported]
- 6 == time is stored (version 1.4+ only)
- 7 == time and color are stored (version 1.4+ only)
- 8 == time, color and near infrared are stored (version 1.4+ only)
- 9 [Not Currently Supported]
- 10 [Not Currently Supported]

**system_id**  String identifying the system that created this LAS file. [Default: “PDAL”]

**a_srs**  The spatial reference system of the file to be written. Can be an EPSG string (e.g. “EPSG:26910”) or a WKT string. [Default: Not set]

**global_encoding**  Various indicators to describe the data. See the LAS documentation. Note that PDAL will always set bit four when creating LAS version 1.4 output. [Default: 0]
project_id UID reserved for the user [Default: Nil UID]

compression Set to “lazperf” or “laszip” to apply compression to the output, creating a LAZ file instead of an LAS file. “lazperf” selects the LazPerf compressor and “laszip” (or “true”) selects the LasZip compressor. PDAL must have been built with support for the requested compressor. [Default: “none”]

scale_x, scale_y, scale_z Scale to be divided from the X, Y and Z nominal values, respectively, after the offset has been applied. The special value auto can be specified, which causes the writer to select a scale to set the stored values of the dimensions to range from [0, 2147483647]. [Default: .01]

Note: written value = (nominal value - offset) / scale.

offset_x, offset_y, offset_z Offset to be subtracted from the X, Y and Z nominal values, respectively, before the value is scaled. The special value auto can be specified, which causes the writer to set the offset to the minimum value of the dimension. [Default: 0]

Note: written value = (nominal value - offset) / scale.

filesource_id The file source id number to use for this file (a value between 0 and 65535 - 0 implies “unassigned”) [Default: 0]

discard_high_return_numbers If true, discard all points with a return number greater than the maximum supported by the point format (5 for formats 0-5, 15 for formats 6-10). [Default: false]

extra_dims Extra dimensions to be written as part of each point beyond those specified by the LAS point format. The format of the option is <dimension_name>=<type> [, 

... ]. Any valid PDAL type (page 293) can be specified.

The special value all can be used in place of a dimension/type list to request that all dimensions that can’t be stored in the predefined LAS point record get added as extra data at the end of each point record.

PDAL writes an extra bytes VLR (User ID: LASF_Spec, Record ID: 4) when extra dims are written. The VLR describes the extra dimensions specified by this option. Note that reading of this VLR is only specified for LAS version 1.4, though some systems will honor it for earlier file formats. The LAS reader (page 71) requires the option use_eb_vlr in order to read the extra bytes VLR for files written with 1.1 - 1.3 LAS format.

Setting --verbose=Info will provide output on the names, types and order of dimensions being written as part of the LAS extra bytes.

pdal_metadata Write two VLRs containing JSON (http://www.json.org/) output with both the Metadata (page 467) and Pipeline (page 45) serialization. [Default: false]
7.3.8 writers.matlab

The Matlab Writer supports writing Matlab .mat files.

The produced files has a single variable, PDAL, an array struct.

Note: The Matlab writer requires the Mat-File API from MathWorks, and it must be explicitly enabled at compile time with the BUILD_PLUGIN_MATLAB=ON variable

Dynamic Plugin

This stage requires a dynamic plugin to operate
Example

```
[    
    {    
        "type": "readers.las",    
        "filename": "inputfile.las"
    },    
    {    
        "type": "writers.matlab",    
        "output_dims": "X,Y,Z,Intensity",    
        "filename": "outputfile.mat"
    }
]
```

Options

**filename**  Output file name [Required]

**output_dims**  A comma-separated list of dimensions to include in the output file. May also be specified as an array of strings. [Default: all available dimensions]

**struct**  Array structure name to read [Default: “PDAL”]

### 7.3.9 writers.nitf

The [NITF](http://en.wikipedia.org/wiki/National_Imagery_Transmission_Format) format is a US Department of Defense format for the transmission of imagery. It supports various formats inside a generic wrapper.

**Note:** LAS inside of NITF is widely supported by software that uses NITF for point cloud storage, and LAZ is supported by some softwares. No other content type beyond those two is widely supported as of January of 2016.

**Default Embedded Stage**

This stage is enabled by default

**Streamable Stage**

This stage supports streaming operations
Example

Example One

```
[
  {
    "type":"readers.las",
    "filename":"inputfile.las"
  },
  {
    "type":"writers.nitf",
    "compression":"laszip",
    "idatim":"20160102220000",
    "forward":"all",
    "acftb":"SENSOR_ID:LIDAR,SENSOR_ID_TYPE:LILN",
    "filename":"outputfile.ntf"
  }
]
```

Example Two

```
[
  {
    "type":"readers.las",
    "filename":"inputfile.las"
  },
  {
    "type":"writers.nitf",
    "compression":"laszip",
    "idatim":"20160102220000",
    "forward":"all",
    "acftb":"SENSOR_ID:LIDAR,SENSOR_ID_TYPE:LILN",
    "aimidb":"ACQUISITION_DATE:20160102235900",
    "filename":"outputfile.ntf"
  }
]
```

Options

**filename**  NITF file to write. The writer will accept a filename containing a single placeholder character (‘#’). If input to the writer consists of multiple PointViews, each will be written to a separate file, where the placeholder will be replaced with an incrementing integer. If no placeholder is found, all PointViews provided to the writer are aggregated into a single file for output. Multiple PointViews are usually the result of using filters.splitter (page 256), filters.chipper (page 252) or filters.divider (page 255).
**PDAL: Point cloud Data Abstraction Library, Release 2.2.0**

- **clevel**  File complexity level (2 characters) [Default: 03]
- **stype**  Standard type (4 characters) [Default: BF01]
- **ostaid**  Originating station ID (10 characters) [Default: PDAL]
- **ftitle**  File title (80 characters) [Default: <spaces>]
- **fsclas**  File security classification (‘T’, ‘S’, ‘C’, ‘R’ or ‘U’) [Default: U]
- **oname**  Originator name (24 characters) [Default: <spaces>]
- **ophone**  Originator phone (18 characters) [Default: <spaces>]
- **fsctlh**  File control and handling (2 characters) [Default: <spaces>]
- **fsclsy**  File classification system (2 characters) [Default: <spaces>]
- **idatim**  Image date and time (format: ‘CCYYMMDDhhmmss’). Required. [Default: AIMIDB.ACQUISITION_DATE if set or <spaces>]
- **iid2**  Image identifier 2 (80 characters) [Default: <spaces>]
- **fscltx**  File classification text (43 characters) [Default: <spaces>]
- **aimidb**  Comma separated list of name/value pairs to complete the AIMIDB (Additional Image ID) TRE record (format name:value). Required: ACQUISITION_DATE, will default to IDATIM value. [Default: NITF defaults]
- **acftb**  Comma separated list of name/value pairs to complete the ACFTB (Aircraft Information) TRE record (format name:value). Required: SENSOR_ID, SENSOR_ID_TYPE [Default: NITF defaults]

**7.3.10 writers.null**

The **null writer** discards its input. No point output is produced when using a null writer.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Streamable Stage**

This stage supports streaming operations
Example

```
[
  {
    "type": "readers.las",
    "filename": "inputfile.las"
  },
  {
    "type": "filters.hexbin"
  },
  {
    "type": "writers.null"
  }
]
```

When used with an option that forces metadata output, like –pipeline-serialization, this pipeline will create a hex boundary for the input file, but no output point data file will be produced.

Options

The null writer discards all passed options.

7.3.11 writers.oci

Note: The OCI writer is deprecated and will be removed in a future release.

The OCI writer is used to write data to Oracle point cloud (http://docs.oracle.com/cd/B28359_01/appdev.111/b28400/sdo_pc_pkg_ref.htm) databases.

Dynamic Plugin

This stage requires a dynamic plugin to operate
Example

```json
[
    {
        "type": "readers.las",
        "filename": "inputfile.las"
    },
    {
        "type": "writers.oci",
        "connection": "grid/grid@localhost/orcl",
        "block_table_name": "QFIT_BLOCKS",
        "base_table_name": "QFIT_CLOUD",
        "cloud_column_name": "CLOUD",
        "srid": "4269",
        "capacity": "5000"
    }
]
```

Options

- **connection**  Oracle connection string to connect to database
- **is3d** Should we use 3D objects (include the z dimension) for SDO_PC PC_EXTENT, BLK_EXTENT, and indexing [Default: false]
- **solid** Define the point cloud’s PC_EXTENT geometry gtype as (1,1007,3) instead of the normal (1,1003,3), and use gtype 3008/2008 vs 3003/2003 for BLK_EXTENT geometry values. [Default: false]
- **overwrite** Wipe the block table and recreate it before loading data [Default: false]
- **verbose** Wipe the block table and recreate it before loading data [Default: false]
- **srid** The Oracle numerical SRID value to use for PC_EXTENT, BLK_EXTENT, and indexing [Default: 0]
- **capacity** The block capacity or maximum number of points a block can contain. [Default: 0]
- **stream_output_precision** The number of digits past the decimal place for writing floats/doubles to streams. This is used for creating the SDO_PC object and adding the index entry to the USER_SDO_GEOM_METADATA for the block table. [Default: 8]
- **cloud_id** The point cloud id that links the point cloud object to the entries in the block table. [Default: -1]
- **block_table_name** The table in which block data for the created SDO_PC will be placed. [Default: “output”]
block_table_partition_column  The column name for which ‘block_table_partition_value’ will be placed in the ‘block_table_name’.

block_table_partition_value  Integer value to use to assing partition IDs in the block table. Used in conjunction with ‘block_table_partition_column’ [Default: 0]

base_table_name  The name of the table which will contain the SDO_PC object. [Default: “hobu”]

cloud_column_name  The column name in ‘base_table_name’ that will hold the SDO_PC object. [Default: “CLOUD”]

base_table_aux_columns  Quoted, comma-separated list of columns to add to the SQL that gets executed as part of the point cloud insertion into the ‘base_table_name’ table.

base_table_aux_values  Quoted, comma-separated values that correspond to ‘base_table_aux_columns’, entries that will get inserted as part of the creation of the SDO_PC entry in the ‘base_table_name’ table.

base_table_boundary_column  The SDO_GEOMETRY column in ‘base_table_name’ in which to insert the WKT in ‘base_table_boundary_wkt’ representing a boundary for the SDO_PC object. Note this is not the same as the ‘base_table_bounds’, which is just a bounding box that is placed on the SDO_PC object itself.

base_table_boundary_wkt  WKT, in the form of a string or a file location, to insert into the SDO_GEOMTRY column defined by ‘base_table_boundary_column’.

pre_block_sql  SQL, in the form of a string or file location, that is executed after the SDO_PC object has been created but before the block data in ‘block_table_name’ are inserted into the database.

pre_sql  SQL, in the form of a string or file location, that is executed before the SDO_PC object is created.

post_block_sql  SQL, in the form of a string or file location, that is executed after the block data in ‘block_table_name’ have been inserted

base_table_bounds  A bounding box, given in the Oracle SRID specified in ‘srid’ to set on the PC_EXTENT object of the SDO_PC. If none is specified, the cumulated bounds of all of the block data are used.

pc_id  Point Cloud id [Default: -1]

pack_ignored_fields  Pack ignored dimensions out of the data buffer that is written [Default: true]

do_trace  turn on server-side binds/waits tracing – needs ALTER SESSION privs. [Default: false]

stream_chunks  Stream block data chunk-wise by the DB’s chunk size rather than as an entire blob. [Default: false]

blob_chunk_count  When streaming, the number of chunks per write to use [Default: 16]
scale_x, scale_y, scale_z / offset_x, offset_y, offset_z  If ANY of these options are specified the X, Y and Z dimensions are adjusted by subtracting the offset and then dividing the values by the specified scaling factor before being written as 32-bit integers (as opposed to double precision values). If any of these options is specified, unspecified scale_<x,y,z> options are given the value of 1.0 and unspecified offset_<x,y,z> are given the value of 0.0.

output_dims  If specified, limits the dimensions written for each point. Dimensions are listed by name and separated by commas.

tolerance  Oracle geometry tolerance. X, Y, and Z dimensions are all currently specified as a single value [Default: 0.05]

### 7.3.12 writers.ogr

The **OGR Writer** will create files of various vector formats (http://www.gdal.org/ogr_formats.html) as supported by the OGR library. PDAL points are generally stored as points in the output format, though PDAL will create multipoint objects instead of point objects if the ‘multicount’ argument is set to a value greater than 1. Points can be written with a single additional value in addition to location if ‘measure_dim’ specifies a valid PDAL dimension and the output format supports measure point types.

By default, the OGR writer will create ESRI shapefiles. The particular OGR driver can be specified with the ‘ogrdriver’ option.

**Example**

```json
[
   "inputfile.las",
   {
      "type": "writers.ogr",
      "filename" "outfile.geojson",
      "measure_dim": "Compression"
   }
]
```
Options

filename  Output file to write. The writer will accept a filename containing a single placeholder character (#). If input to the writer consists of multiple PointViews, each will be written to a separate file, where the placeholder will be replaced with an incrementing integer. If no placeholder is found, all PointViews provided to the writer are aggregated into a single file for output. Multiple PointViews are usually the result of multiple input files, or using filters.splitter (page 256), filters.chipper (page 252) or filters.divider (page 255).

The driver will use the OGR GEOjson driver if the output filename extension is ‘geojson’, and the ESRI shapefile driver if the output filename extension is ‘shp’. If neither extension is recognized, the filename is taken to represent a directory in which ESRI shapefiles are written. The driver can be explicitly specified by using the ‘ogrdriver’ option.

multicount  If 1, point objects will be written. If greater than 1, specifies the number of points to group into a multipoint object. Not all OGR drivers support multipoint objects. [Default: 1]

measure_dim  If specified, points will be written with an extra data field, the dimension of which is specified by this option. Not all output formats support measure data. [Default: None]

Note: The measure_dim option is only supported if PDAL is built with GDAL version 2.1 or later.

ogrdriver  The OGR driver to use for output. This option overrides any inference made about output drivers from filename (page 137).

7.3.13 writers.pcd

The PCD Writer supports writing to Point Cloud Data (PCD) (https://pcl-tutorials.readthedocs.io/en/latest/pcd_file_format.html) formatted files, which are used by the Point Cloud Library (PCL) (http://pointclouds.org).

By default, compression is not enabled, and the PCD writer will output ASCII formatted data.

Default Embedded Stage

This stage is enabled by default

Streamable Stage
This stage supports streaming operations

Example

```
[
    {
        "type": "readers.pcd",
        "filename": "inputfile.pcd"
    },
    {
        "type": "writers.pcd",
        "filename": "outputfile.pcd"
    }
]
```

Options

**filename**  PCD file to write [Required]

**compression**  Level of PCD compression to use (ascii, binary, compressed) [Default: “ascii”]

**precision**  Decimal Precision for output of values. This can be overridden for individual dimensions using the order option. [Default: 3]

**order**  Comma-separated list of dimension names in the desired output order. For example “X,Y,Z,Red,Green,Blue”. Dimension names can optionally be followed by a PDAL type (e.g., Unsigned32) and dimension-specific precision (used only with “ascii” compression). Ex: “X=Float:2, Y=Float:2, Z=Float:3, Intensity=Unsigned32” If no precision is specified the value provided with the precision option is used. The default dimension type is double precision float. [Default: none]

**keep_unspecified**  If true, writes all dimensions. Dimensions specified with the order option precede those not specified. [Default: true]

7.3.14 writers.pgpointcloud

The **PostgreSQL Pointcloud Writer** allows you to write to PostgreSQL database that have the [PostgreSQL Pointcloud](http://github.com/pramsey/pointcloud) extension enabled. The Pointcloud extension stores point cloud data in tables that contain rows of patches. Each patch in turn contains a large number of spatially nearby points.

While you can theoretically store the contents of a whole file of points in a single patch, it is more practical to store a table full of smaller patches, where the patches are under the
PostgreSQL page size (8kb). For most LIDAR data, this practically means a patch size of between 400 and 600 points.

In order to create patches of the right size, the Pointcloud writer should be preceded in the pipeline file by filters.chipper (page 252).

The pgpointcloud format does not support WKT spatial reference specifications. A subset of spatial references can be stored by using the ‘srid’ option, which allows storage of an EPSG code (http://www.epsg.org) that covers many common spatial references. PDAL makes no attempt to reproject data to your specified srid. Use filters.reprojection (page 222) for this purpose.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
[  
  
  {  
    "type":"readers.las",
    "filename":"inputfile.las",
    "spatialreference":"EPSG:26916"
  },  
  
  {  
    "type":"filters.chipper",
    "capacity":400
  },  
  
  {  
    "type":"writers.pgpointcloud",
    "connection":"host='localhost' dbname='lidar' user='pramsey'",
    "table":"example",
    "compression":"dimensional",
    "srid":"26916"
  }
]
```
Options

connection  PostgreSQL connection string. In the form “host=hostname dbname=database user=username password=pw port=5432” [Required]

table  Database table to write to. [Required]

schema  Database schema to write to. [Default: “public”]

column  Table column to put patches into. [Default: “pa”]

compression  Patch compression type to use. [Default: “dimensional”]
  - none  applies no compression
  - dimensional  applies dynamic compression to each dimension separately
  - lazperf  applies a “laz” compression (using the laz-perf (https://github.com/hobu/laz-perf) library in PostgreSQL Pointcloud)

overwrite  To drop the table before writing set to ‘true’. To append to the table set to ‘false’. [Default: false]

srid  Spatial reference ID (relative to the spatial_ref_sys table in PostGIS) to store with the point cloud schema. [Default: 4326]

pcid  An optional existing PCID to use for the point cloud schema. If specified, the schema must be present. If not specified, a match will still be looked for, or a new schema will be inserted. [Default: 0]

pre_sql  SQL to execute before running the translation. If the value references a file, the file is read and any SQL inside is executed. Otherwise the value is executed as SQL itself. [Optional]

post_sql  SQL to execute after running the translation. If the value references a file, the file is read and any SQL inside is executed. Otherwise the value is executed as SQL itself. [Optional]

scale_x, scale_y, scale_z / offset_x, offset_y, offset_z  If ANY of these options are specified the X, Y and Z dimensions are adjusted by subtracting the offset and then dividing the values by the specified scaling factor before being written as 32-bit integers (as opposed to double precision values). If any of these options is specified, unspecified scale_<x,y,z> options are given the value of 1.0 and unspecified offset_<x,y,z> are given the value of 0.0.

output_dims  If specified, limits the dimensions written for each point. Dimensions are listed by name and separated by commas.
7.3.15 writers.ply

The ply writer writes the polygon file format (http://paulbourke.net/dataformats/ply/), a common file format for storing three dimensional models. The writer emits points as PLY vertices. The writer can also emit a mesh as a set of faces. filters.greedyprojection (page 270) and filters.poisson (page 271) create a mesh suitable for output as faces.

Default Embedded Stage

This stage is enabled by default

Example

```json
[
  {
    "type":"readers.pcd",
    "filename":"inputfile.pcd"
  },
  {
    "type":"writers.ply",
    "storage_mode":"little endian",
    "filename":"outputfile.ply"
  }
]
```

Options

filename  ply file to write [Required]

storage_mode  Type of ply file to write. Valid values are ‘ascii’, ‘little endian’, ‘big endian’.  [Default: “ascii”]

dims  List of dimensions (and Types (page 293)) in the format
  <dimension_name>[=<type>] [,...] to write as output. (e.g., “Y=int32_t, X,Red=char”)  [Default: All dimensions with stored types]

faces  Write a mesh as faces in addition to writing points as vertices.  [Default: false]

sized_types  PLY has variously been written with explicitly sized type strings (‘int8’, ‘float32’, ‘uint32’, etc.) and implied sized type strings (‘char’, ‘float’, ‘int’, etc.). If true, explicitly sized type strings are used. If false, implicitly sized type strings are used.  [Default: true]

precision  If specified, the number of digits to the right of the decimal place using f-style formatting. Only permitted when ‘storage_mode’ is ‘ascii’. See the printf
7.3.16 writers.raster

The Raster Writer writes an existing raster to a file. Output is produced using GDAL (http://gdal.org) and can use any driver that supports creation of rasters (http://www.gdal.org/formats_list.html). A data_type (page 143) can be specified for the raster (double, float, int32, etc.). If no data type is specified, the data type with the largest range supported by the driver is used.

Cells that have no value are given a value specified by the nodata (page 143) option.

Default Embedded Stage
This stage is enabled by default

Streamable Stage
This stage supports streaming operations

Basic Example

This pipeline reads the file autzen_trim.las, triangulates the data, creates a raster based on the Z dimension as determined by interpolation of the location and values of ‘Z’ of the vertices of a containing triangle, if any exists. The resulting raster is written to “outputfile.tif”.

```json
[  
    "pdal/test/data/las/autzen_trim.las",
    {
        "type": "filters.delaunay"
    },
    {
        "type": "filters.faceraster",
        "resolution": 1
    },
    {
        "type": "writers.raster",
        "filename": "outputfile.tif"
    }
]```
Options

filename  Name of output file. [Required]

gdaldriver GDAL code of the GDAL driver (http://www.gdal.org/formats_list.html) to use to write the output. [Default: “GTiff”]

gdalopts A list of key/value options to pass directly to the GDAL driver. The format is name=value,name=value,… The option may be specified any number of times.

Note: The INTERLEAVE GDAL driver option is not supported. writers.gdal always uses BAND interleaving.

rasters A comma-separated list of raster names to be written as bands of the raster. All rasters must have the same limits (origin/width/height). Rasters following the first that don’t have the same limits will be dropped. If no raster names are provided, only the first raster found will be placed into a single band for output.

data_type The data type (page 293) to use for the output raster. Many GDAL drivers only support a limited set of output data types. [Default: depends on the driver]

nodata The value to use for a raster cell if the raster contains no data in a cell. Note that the nodata written to the output may be different from that of the raster being written. [Default: depends on the data_type (page 143). -9999 for double, float, int and short, 9999 for unsigned int and unsigned short, 255 for unsigned char and -128 for char]

7.3.17 writers.sbet

The SBET writer writes files in the SBET format, used for exchange data from inertial measurement units (IMUs).

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations
Example

```json
[
    "input.sbet",
    "output.sbet"
]
```

Options

**filename**  File to write. [Required]

**angles_are_degrees**  Convert all angular values from degrees to radians before write. [Default: true]

### 7.3.18 writers.sqlite

The SQLite (http://sqlite.org) driver outputs point cloud data into a PDAL-specific scheme that matches the approach of `readers.pgpointcloud` (page 89) and `readers.oci` (page 86).

**Dynamic Plugin**

This stage requires a dynamic plugin to operate

Example

```json
[
    {
        "type": "readers.las",
        "filename": "inputfile.las"
    },
    {
        "type": "filters.chipper",
        "capacity": 50
    },
    {
        "type": "writers.sqlite",
        "connection": "output.sqlite",
        "cloud_table_name": "SIMPLE_CLOUD",
        "pre_sql": ",",
        "post_sql": ",",
        "block_table_name": "SIMPLE_BLOCKS",
    }
]
```
Options

connection  SQLlite filename [Required]
cloud_table_name  Name of table to store cloud (file) information [Required]
block_table_name  Name of table to store patch information [Required]
cloud_column_name  Name of column to store primary cloud_id key [Default: “cloud”]
compression  Use LAZperf (https://github.com/hobu/laz-perf) compression technique to store patches. [Default: false]
overwrite  Drop the table before writing. To append to the table set to false. [Default: true]
pre_sql  Optional SQL to execute before running the translation. If the value references a file, the file is read and any SQL inside is executed. Otherwise the value is executed as SQL itself.
post_sql  Optional SQL to execute after running the translation. If the value references a file, the file is read and any SQL inside is executed. Otherwise the value is executed as SQL itself.
scale_x, scale_y, scale_z / offset_x, offset_y, offset_z  If ANY of these options are specified the X, Y and Z dimensions are adjusted by subtracting the offset and then dividing the values by the specified scaling factor before being written as 32-bit integers (as opposed to double precision values). If any of these options is specified, unspecified scale_<x,y,z> options are given the value of 1.0 and unspecified offset_<x,y,z> are given the value of 0.0.
output_dims  If specified, limits the dimensions written for each point. Dimensions are listed by name and separated by commas.

7.3.19 writers.text

The text writer writes out to a text file. This is useful for debugging or getting smaller files into an easily parseable format. The text writer supports both GeoJSON (http://geojson.org) and CSV (http://en.wikipedia.org/wiki/Comma-separated_values) output.
This stage is enabled by default

Streamable Stage
This stage supports streaming operations

Example

```
[
  {
    "type": "readers.las",
    "filename": "inputfile.las"
  },
  {
    "type": "writers.text",
    "format": "geojson",
    "order": "X,Y,Z",
    "keep_unspecified": "false",
    "filename": "outputfile.txt"
  }
]
```

Options

- **filename**  File to write to, or “STDOUT” to write to standard out [Required]
- **format**  Output format to use. One of geojson or csv. [Default: “csv”]
- **precision**  Decimal Precision for output of values. This can be overridden for individual dimensions using the order option. [Default: 3]
- **order**  Comma-separated list of dimension names in the desired output order. For example “X,Y,Z,Red,Green,Blue”. Dimension names can optionally be followed with a colon (‘:’) and an integer to indicate the precision to use for output. Ex: “X:3, Y:5,Z:0” If no precision is specified the value provided with the precision (page 146) option is used. [Default: none]
- **keep_unspecified**  If true, writes all dimensions. Dimensions specified with the order (page 146) option precede those not specified. [Default: true]
- **jscallback**  When producing GeoJSON, the callback allows you to wrap the data in a function, so the output can be evaluated in a <script> tag.
quote_header  When producing CSV, should the column header named by quoted? [Default: true]
write_header  Whether a header should be written. [Default: true]
newline  When producing CSV, what newline character should be used? (For Windows, \\r\\n is common.) [Default: “\n”]
delimiter  When producing CSV, what character to use as a delimiter? [Default: “;”]

7.3.20 writers.tiledb

Implements TileDB (https://tiledb.io) 1.4.1+ reads from an array.

Dynamic Plugin
This stage requires a dynamic plugin to operate

Streamable Stage
This stage supports streaming operations

Example

```json
[
  {
    "type": "readers.las",
    "array_name": "input.las"
  },
  {
    "type": "writers.tiledb",
    "array_name": "output_array"
  }
]
```
Options

array_name  TileDB (https://tiledb.io) array to write to. [Required]
config_file  TileDB (https://tiledb.io) configuration file [Optional]
tile_data_capacity  Number of points per tile [Optional]
x_tile_size  Tile size (x) in a Cartesian projection [Optional]
y_tile_size  Tile size (y) in a Cartesian projection [Optional]
z_tile_size  Tile size (z) in a Cartesian projection [Optional]
chunk_size  Point cache size for chunked writes [Optional]
compression  TileDB compression type for attributes, default is None [Optional]
compression_level  TileDB compression level for chosen compression [Optional]
append  Append to an existing TileDB array with the same schema [Optional]
stats  Dump query stats to stdout [Optional]
filters  JSON array or object of compression filters for either coords or attributes of the form
{coords/attributename : {“compression”: name, compression_options: value, . . . }} [Optional]

By default TileDB will use the following set of compression filters for coordinates and attributes;

```json
{
  "coords": [
    {"compression": "bit-shuffle"},
    {"compression": "gzip", "compression_level": 9}
  ],
  "Intensity": {"compression": "bzip2", "compression_level": 5},
  "ReturnNumber": {"compression": "zstd", "compression_level": 75},
  "NumberOfReturns": {"compression": "zstd", "compression_level": 75},
  "ScanDirectionFlag": {"compression": "bzip2", "compression_level": 5},
  "EdgeOfFlightLine": {"compression": "bzip2", "compression_level": 5},
  "Classification": {"compression": "gzip", "compression_level": 9}
  "ScanAngleRank": {"compression": "bzip2", "compression_level": 5},
  "UserData": {"compression": "gzip", "compression_level": 9},
  "PointSourceId": {"compression": "bzip2"},
  "Red": {"compression": "rle"},
  ...
}
```

(continues on next page)
"Green": {"compression": "rle"},
"Blue": {"compression": "rle"},
"Gpstime": [
    {"compression": "bit-shuffle"},
    {"compression": "zstd", "compression_level": 75}
]
}

**writers.bpf** (page 114) Write BPF version 3 files. BPF is an NGA specification for point cloud data.

**writers.ept_addon** (page 116) Append additional dimensions to Entwine resources.

**writers.e57** (page 118) Write data in the E57 format.

**writers.gdal** (page 119) Create a raster from a point cloud using an interpolation algorithm.

**writers.geowave** (page 122) Write point cloud data to Accumulo.

**writers.glTF** (page 124) Write mesh data in GLTF format. Point clouds without meshes cannot be written.

**writers.las** (page 125) Write ASPRS LAS versions 1.0 - 1.4 formatted data. LAZ support is also available if enabled at compile-time.

**writers.matlab** (page 129) Write MATLAB .mat files. The output has a single array struct.

**writers.nitf** (page 130) Write LAS and LAZ point cloud data, wrapped in a NITF 2.1 file.

**writers.null** (page 132) Provides a sink for points in a pipeline. It’s the same as sending pipeline output to /dev/null.

**writers.oci** (page 133) Write data to Oracle point cloud databases. [deprecated]

**writers.ogr** (page 136) Write a point cloud as a set of OGR points/multipoints.

**writers.pcd** (page 137) Write PCD-formatted files in the ASCII, binary, or compressed format.

**writers.pgpointcloud** (page 138) Write to a PostgreSQL database that has the PostgreSQL Pointcloud extension enabled.

**writers.ply** (page 141) Write points as PLY vertices. Can also emit a mesh as a set of faces.

**writers.raster** (page 142) Writes rasters using GDAL. Rasters must be created using a PDAL filter.

**writers.sbet** (page 143) Write data in the SBET format.

**writers.sqlite** (page 144) Write point cloud data in a scheme that matches the approach used in the PostgreSQL Pointcloud and OCI readers.

**writers.text** (page 145) Write points in a text file. GeoJSON and CSV formats are supported.
7.4 Filters

Filters operate on data as inline operations. They can remove, modify, reorganize, and add points to the data stream as it goes by. Some filters can only operate on dimensions they understand (consider filters.reprojection (page 222) doing geographic reprojection on XYZ coordinates), while others do not interrogate the point data at all and simply reorganize or split data.

7.4.1 Create

PDAL filters commonly create new dimensions (e.g., HeightAboveGround) or alter existing ones (e.g., Classification). These filters will not invalidate an existing KD-tree.

Note: We treat those filters that alter XYZ coordinates separately.

Note: When creating new dimensions, be mindful of the writer you are using and whether or not the custom dimension can be written to disk if that is the desired behavior.

Classification

Ground/Unclassified

filters.csf

The Cloth Simulation Filter (CSF) classifies ground points based on the approach outlined in [Zhang2016].

Default Embedded Stage

This stage is enabled by default
Example

The sample pipeline below uses CSF to segment ground and non-ground returns, using default options, and writing only the ground returns to the output file.

```json
[
  "input.las",
  {
    "type": "filters.csf"
  },
  {
    "type": "filters.range",
    "limits": "Classification[2:2]"
  },
  "output.laz"
]
```

Options

- **resolution** Cloth resolution. [Default: 1.0]
- **ignore** A range (page 249) of values of a dimension to ignore.
- **returns** Return types to include in output. Valid values are “first”, “last”, “intermediate” and “only”. [Default: “last, only”]
- **threshold** Classification threshold. [Default: 0.5]
- **smooth** Perform slope post-processing? [Default: true]
- **step** Time step. [Default: 0.65]
- **rigidness** Rigidness. [Default: 3]
- **iterations** Maximum number of iterations. [Default: 500]
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.pmf

The Progressive Morphological Filter (PMF) is a method of segmenting ground and non-ground returns. This filter is an implementation of the method described in [Zhang2003].

Default Embedded Stage

This stage is enabled by default

Example

```
[  
  "input.las",
  {
    "type": "filters.pmf"
  },
  "output.las"
]
```

Notes

- `slope` (page 153) controls the height threshold at each iteration. A slope of 1.0 represents a 1:1 or 45°.

- `initial_distance` (page 153) is _intended_ to be set to account for z noise, so for a flat surface if you have an uncertainty of around 15 cm, you set `initial_distance` (page 153) large enough to not exclude these points from the ground.

- For a given iteration, the height threshold is determined by multiplying slope by `cell_size` (page 153) by the difference in window size between the current and last iteration, plus the `initial_distance` (page 153). This height threshold is constant across all cells and is maxed out at the `max_distance` (page 153) value. If the difference in elevation between a point and its “opened” value (from the morphological operator) exceeds the height threshold, it is treated as non-ground. So, bigger slope leads to bigger height thresholds, and these grow with each iteration (not to exceed the max). With flat terrain, keep this low, the thresholds are small, and stuff is more aggressively dumped into non-ground class. In rugged terrain, open things up a little, but then you can start missing buildings, veg, etc.

- Very large `max_window_size` (page 153) values will result in a lot of potentially extra iteration. This parameter can have a strongly negative impact on computation performance.
• *exponential* (page 153) is used to control the rate of growth of morphological window sizes toward *max_window_size* (page 153). Linear growth preserves gradually changing topographic features well, but demands considerable compute time. The default behavior is to grow the window sizes exponentially, thus reducing the number of iterations.

• This filter will mark all returns deemed to be ground returns with a classification value of 2 (per the LAS specification). To extract only these returns, users can add a *range filter* (page 248) to the pipeline.

```json
{
    "type": "filters.range",
    "limits": "Classification[2:2]"
}
```

**Note:** [Zhang2003] describes the consequences and relationships of the parameters in more detail and is the canonical resource on the topic.

**Options**

**cell_size** Cell Size. [Default: 1]

**exponential** Use exponential growth for window sizes? [Default: true]

**ignore** Range of values to ignore. [Optional]

**initial_distance** Initial distance. [Default: 0.15]

**returns** Comma-separated list of return types into which data should be segmented. Valid groups are “last”, “first”, “intermediate” and “only”. [Default: “last, only”]

**max_distance** Maximum distance. [Default: 2.5]

**max_window_size** Maximum window size. [Default: 33]

**slope** Slope. [Default: 1.0]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.skewnessbalancing

Skewness Balancing classifies ground points based on the approach outlined in [Bartels2010].

Default Embedded Stage
This stage is enabled by default

Note: For Skewness Balancing to work well, the scene being processed needs to be quite flat, otherwise many above ground features will begin to be included in the ground surface.

Example
The sample pipeline below uses the Skewness Balancing filter to segment ground and non-ground returns, using default options, and writing only the ground returns to the output file.

```
[  
   "input.las",
   {  
      "type": "filters.skewnessbalancing"
   },
   {  
      "type": "filters.range",
      "limits": "Classification[2:2]"
   },
   "output.laz"
]
```

Options

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
Note: The Skewness Balancing method is touted as being threshold-free. We may still in the future add convenience parameters that are common to other ground segmentation filters, such as returns or ignore to limit the points under consideration for filtering.

filters.smrf

The Simple Morphological Filter (SMRF) classifies ground points based on the approach outlined in [Pingel2013].

Default Embedded Stage

This stage is enabled by default

Example #1

The sample pipeline below uses the SMRF filter to segment ground and non-ground returns, using default options, and writing only the ground returns to the output file.

```json
[
   "input.las",
   {
      "type": "filters.smrf"
   },
   {
      "type": "filters.range",
      "limits": "Classification[2:2]"
   },
   "output.laz"
]
```

Example #2

A more complete example, specifying some options. These match the optimized parameters for Sample 1 given in Table 3 of [Pingel2013].

```json
[
   "input.las",
   {
      "type": "filters.smrf",
      ...
```
"scalar":1.2,
"slope":0.2,
"threshold":0.45,
"window":16.0
},
{
    "type": "filters.range",
    "limits": "Classification[2:2]"
},
"output.laz"

Options

cell Cell size. [Default: 1.0]
classbits Selectively ignore points marked as “synthetic”, “keypoint”, or “withheld”. [Default: empty string, use all points]
cut Cut net size (cut=0 skips the net cutting step). [Default: 0.0]
dir Optional output directory for debugging intermediate rasters.
ignore A range (page 249) of values of a dimension to ignore.
returns Return types to include in output. Valid values are “first”, “last”, “intermediate” and “only”. [Default: “last, only”]
scalar Elevation scalar. [Default: 1.25]
slope Slope (rise over run). [Default: 0.15]
threshold Elevation threshold. [Default: 0.5]
window Max window size. [Default: 18.0]
where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.csf (page 150) Label ground/non-ground returns using [Zhang2016].
filters.pmf (page 152) Label ground/non-ground returns using [Zhang2003].
**filters.skewnessbalancing** (page 154) Label ground/non-ground returns using [Bartels2010].

**filters.smrf** (page 155) Label ground/non-ground returns using [Pingel2013].

**Noise**

**filters.elm**

The Extended Local Minimum (ELM) filter marks low points as noise. This filter is an implementation of the method described in [Chen2012].

ELM begins by rasterizing the input point cloud data at the given cell size. Within each cell, the lowest point is considered noise if the next lowest point is a given threshold above the current point. If it is marked as noise, the difference between the next two points is also considered, marking points as noise if needed, and continuing until another neighbor is found to be within the threshold. At this point, iteration for the current cell stops, and the next cell is considered.

**Default Embedded Stage**

This stage is enabled by default

**Example #1**

The following PDAL pipeline applies the ELM filter, using a cell size of 20 and applying the classification code of 18 to those points determined to be noise.

```json
{
   "pipeline": [
      "input.las",
      {
         "type": "filters.elm",
         "cell": 20.0,
         "class": 18
      },
      "output.las"
   ]
}
```
Example #2

This variation of the pipeline begins by assigning a value of 0 to all classifications, thus resetting any existing classifications. It then proceeds to compute ELM with a threshold (page 158) value of 2.0, and finishes by extracting all returns that are not marked as noise.

```
[
    "input.las",
    {
      "type": "filters.assign",
      "assignment": "Classification[:] = 0"
    },
    {
      "type": "filters.elm",
      "threshold": 2.0
    },
    {
      "type": "filters.range",
      "limits": "Classification![:]=0"
    },
    "output.las"
]
```

Options

cell  Cell size. [Default: 10.0]

class  Classification value to apply to noise points. [Default: 7]

threshold  Threshold value to identify low noise points. [Default: 1.0]

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.outlier

The outlier filter provides two outlier filtering methods: radius and statistical. These two approaches are discussed in further detail below.

It is worth noting that both filtering methods simply apply a classification value of 7 to the noise points (per the LAS specification (http://www.asprs.org/a/society/committees/standards/LAS_1_4_r13.pdf)). To remove the noise points altogether, users can add a range filter (page 248) to their pipeline, downstream from the outlier filter.

Default Embedded Stage

This stage is enabled by default

```json
{
    "type": "filters.range",
    "limits": "Classification![7:7]"
}
```

Statistical Method

The default method for identifying outlier points is the statistical outlier method. This method requires two passes through the input PointView, first to compute a threshold value based on global statistics, and second to identify outliers using the computed threshold.

In the first pass, for each point \(p_i\) in the input PointView, compute the mean distance \(\mu_i\) to each of the \(k\) nearest neighbors (where \(k\) is configurable and specified by \(\text{mean}_k\) (page 162)). Then,

\[
\overline{\mu} = \frac{1}{N} \sum_{i=1}^{N} \mu_i
\]

\[
\sigma = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (\mu_i - \overline{\mu})^2}
\]

A global mean \(\overline{\mu}\) of these mean distances is then computed along with the standard deviation \(\sigma\). From this, the threshold is computed as

\[
t = \mu + m\sigma
\]

where \(m\) is a user-defined multiplier specified by \(\text{multiplier}\) (page 162).
We now iterate over the pre-computed mean distances $\mu_i$ and compare to computed threshold value. If $\mu_i$ is greater than the threshold, it is marked as an outlier.

$$\text{outlier}_i = \begin{cases} 
\text{true}, & \text{if } \mu_i \geq t \\
\text{false}, & \text{otherwise}
\end{cases}$$

Before outlier removal, noise points can be found both above and below the scene. After outlier removal, the noise points are removed.

See [Rusu2008] for more information.
Example

In this example, points are marked as outliers if the average distance to each of the 12 nearest neighbors is below the computed threshold.

```json
[
  "input.las",
  {
    "type": "filters.outlier",
    "method": "statistical",
    "mean_k": 12,
    "multiplier": 2.2
  },
  "output.las"
]
```

Radius Method

For each point \( p_i \) in the input \texttt{PointView}, this method counts the number of neighboring points \( k_i \) within radius \( r \) (specified by \texttt{radius} (page 162)). If \( k_i < k_{\text{min}} \), where \( k_{\text{min}} \) is the minimum number of neighbors specified by \texttt{min_k} (page 162), it is marked as an outlier.

\[
outlier_i = \begin{cases} 
  \text{true}, & \text{if } k_i < k_{\text{min}} \\
  \text{false}, & \text{otherwise}
\end{cases}
\]
Example

The following example will mark points as outliers when there are fewer than four neighbors within a radius of 1.0.

```json
[
    "input.las",
    {
        "type": "filters.outlier",
        "method": "radius",
        "radius": 1.0,
        "min_k": 4
    },
    "output.las"
]
```

Options

class  The classification value to apply to outliers. [Default: 7]

method  The outlier removal method (either “statistical” or “radius”). [Default: “statistical”]

min_k  Minimum number of neighbors in radius (radius method only). [Default: 2]

radius  Radius (radius method only). [Default: 1.0]

mean_k  Mean number of neighbors (statistical method only). [Default: 8]

multiplier  Standard deviation threshold (statistical method only). [Default: 2.0]

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.elm  (page 157)  Marks low points as noise.

filters.outlier  (page 159)  Label noise points using either a statistical or radius outlier detection.
Consensus

**filters.neighborclassifier**

The **neighborclassifier filter** allows you to update the value of the classification for specific points to a value determined by a K-nearest neighbors vote. For each point, the \( k \) (page 164) nearest neighbors are queried and if more than half of them have the same value, the filter updates the selected point accordingly.

For example, if an automated classification procedure put/left erroneous vegetation points near the edges of buildings which were largely classified correctly, you could try using this filter to fix that problem.

Similarly, some automated classification processes result in predictions for only a subset of the original point cloud. This filter could be used to extrapolate those predictions to the original.

**Default Embedded Stage**

This stage is enabled by default.

**Example 1**

This pipeline updates the Classification of all points with classification 1 (unclassified) based on the consensus (majority) of its nearest 10 neighbors.

```json
[  
  "autzen_class.las",
  {
    "type" : "filters.neighborclassifier",
    "domain" : "Classification[1:1]",
    "k" : 10
  },
  "autzen_class_refined.las"
]
```
Example 2

This pipeline moves all the classifications from “pred.txt” to src.las. Any points in src.las that are not in pred.txt will be assigned based on the closest point in pred.txt.

```
[
  "src.las",
  {
    "type" : "filters.neighborclassifier",
    "k" : 1,
    "candidate" : "pred.txt"
  },
  "dest.las"
]
```

Options

candidate A filename which points to the point cloud containing the points which will do the voting. If not specified, defaults to the input of the filter.

domain A range (page 249) which selects points to be processed by the filter. Can be specified multiple times. Points satisfying any range will be processed.

k An integer which specifies the number of neighbors which vote on each selected point.

where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.neighborclassifier (page 163) Update pointwise classification using k-nearest neighbor consensus voting.
Height Above Ground

filters.hag_delaunay

The **Height Above Ground Delaunay filter** takes as input a point cloud with Classification set to 2 for ground points. It creates a new dimension, HeightAboveGround, that contains the normalized height values.

**Note:** We expect ground returns to have the classification value of 2 in keeping with the ASPRS Standard LIDAR Point Classes (http://www.asprs.org/a/society/committees/standards/LAS_1_4_r13.pdf).

Ground points may be generated by `filters.pmf` (page 152) or `filters.smrf` (page 155), but you can use any method you choose, as long as the ground returns are marked.

Normalized heights are a commonly used attribute of point cloud data. This can also be referred to as *height above ground* (HAG) or *above ground level* (AGL) heights. In the end, it is simply a measure of a point’s relative height as opposed to its raw elevation value.

The filter creates a delaunay triangulation of the count (page 167) ground points closest to the non-ground point in question. If the non-ground point is within the triangulated area, the assigned HeightAboveGround is the difference between its Z value and a ground height interpolated from the three vertices of the containing triangle. If the non-ground point is outside of the triangulated area, its HeightAboveGround is calculated as the difference between its Z value and the Z value of the nearest ground point.

Choosing a value for count (page 167) is difficult, as placing the non-ground point in the triangulated area depends on the layout of the nearby points. If, for example, all the ground points near a non-ground point lay on one side of that non-ground point, finding a containing triangle will fail.

**Default Embedded Stage**

This stage is enabled by default

**Example #1**

Using the autzen dataset (here shown colored by elevation), which already has points classified as ground
we execute the following pipeline

```json
[
    "autzen.laz",
    {
      "type":"filters.hag_delaunay"
    },
    {
      "type":"writers.laz",
      "filename":"autzen_hag_delaunay.laz",
      "extra_dims":"HeightAboveGround=float32"
    }
]
```

which is equivalent to the `pdal translate` command

```bash
$ pdal translate autzen.laz autzen_hag_delaunay.laz hag_delaunay
   --writers.las.extra_dims="HeightAboveGround=float32"
```

In either case, the result, when colored by the normalized height instead of elevation is
Options

count The number of ground neighbors to consider when determining the height above ground for a non-ground point. [Default: 10]

allow_extrapolation If false and a non-ground point lies outside of the bounding box of all ground points, its HeightAboveGround is set to 0. If true and delaunay is set, the HeightAboveGround is set to the difference between the heights of the non-ground point and nearest ground point. [Default: false]

where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.hag_dem

The **Height Above Ground (HAG) Digital Elevation Model (DEM) filter** loads a GDAL-readable raster image specifying the DEM. The $z$ value of each point in the input is compared against the value at the corresponding X,Y location in the DEM raster. It creates a new dimension, HeightAboveGround, that contains the normalized height values.

Normalized heights are a commonly used attribute of point cloud data. This can also be referred to as *height above ground* (HAG) or *above ground level* (AGL) heights. In the end, it is simply a measure of a point’s relative height as opposed to its raw elevation value.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Streamable Stage**

This stage supports streaming operations

---

**Example #1**

Using the autzen dataset (here shown colored by elevation)
we generate a DEM based on the points already classified as ground

```bash
$ pdal translate autzen.laz autzen_dem.tif range \
    --filters.range.limits="Classification[2:2]" \
    --writers.gdal.output_type="idw" \
    --writers.gdal.resolution=6 \
    --writers.gdal.window_size=24
```

and execute the following pipeline

```json
[
    "autzen.laz",
    {
      "type": "filters.hag_dem",
      "raster": "autzen_dem.tif"
    },
    {
      "type": "writers.las",
      "filename": "autzen_hag_dem.laz",
      "extra_dims": "HeightAboveGround=float32"
    }
]
```

which is equivalent to the `pdal translate` command

---

**7.4. Filters**
In either case, the result, when colored by the normalized height instead of elevation is

![Image of colored point cloud]

**Options**

- **raster**  GDAL-readable raster to use for DEM.
- **band**  GDAL Band number to read (count from 1). [Default: 1]
- **zero_ground**  If true, set HAG of ground-classified points to 0 rather than comparing \( Z \) value to raster DEM. [Default: true]
- **where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
The **Height Above Ground Nearest Neighbor filter** takes as input a point cloud with `Classification` set to 2 for ground points. It creates a new dimension, `HeightAboveGround`, that contains the normalized height values.

**Note:** We expect ground returns to have the classification value of 2 in keeping with the ASPRS Standard LIDAR Point Classes (http://www.asprs.org/a/society/committees/standards/LAS_1_4_r13.pdf).

Ground points may be generated by `filters.pmf` (page 152) or `filters.smrf` (page 155), but you can use any method you choose, as long as the ground returns are marked.

Normalized heights are a commonly used attribute of point cloud data. This can also be referred to as *height above ground* (HAG) or *above ground level* (AGL) heights. In the end, it is simply a measure of a point’s relative height as opposed to its raw elevation value.

The filter finds the `count` (page 174) ground points nearest the non-ground point under consideration. It calculates an average ground height weighted by the distance of each ground point from the non-ground point. The `HeightAboveGround` is the difference between the Z value of the non-ground point and the interpolated ground height.

**Default Embedded Stage**

This stage is enabled by default.

**Example #1**

Using the autzen dataset (here shown colored by elevation), which already has points classified as ground
we execute the following pipeline

```json
[
  "autzen.laz",
  {
    "type": "filters.hag_nn"
  },
  {
    "type": "writers.laz",
    "filename": "autzen_hag_nn.laz",
    "extra_dims": "HeightAboveGround=float32"
  }
]
```

which is equivalent to the `pdal translate` command

```
$ pdal translate autzen.laz autzen_hag_nn.laz hag_nn \ 
   --writers.las.extra_dims="HeightAboveGround=float32"
```

In either case, the result, when colored by the normalized height instead of elevation is
Example #2

In the previous example, we chose to write HeightAboveGround using the extra_dims option of writers.las (page 125). If you’d instead like to overwrite your Z values, then follow the height filter with filters.ferry (page 208) as shown

```json
[
  "autzen.laz",
  {
    "type": "filters.hag_nn"
  },
  {
    "type": "filters.ferry",
    "dimensions": "HeightAboveGround=>Z"
  },
  "autzen-height-as-Z.laz"
]
```
Example #3

If you don’t yet have points classified as ground, start with `filters.pmf` (page 152) or `filters.smrf` (page 155) to label ground returns, as shown

```json
[  
  "autzen.laz",
  {  
    "type": "filters.smrf"
  },
  {  
    "type": "filters.hag_nn"
  },
  {  
    "type": "filters.ferry",
    "dimensions": "HeightAboveGround=>Z"
  },
  "autzen-height-as-Z-smrf.laz"
]
```

Options

- **count**  The number of ground neighbors to consider when determining the height above ground for a non-ground point. [Default: 1]
- **max_distance**  Use only ground points within `max_distance` of non-ground point when performing neighbor interpolation. [Default: None]
- **allow_extrapolation**  If false and a non-ground point lies outside of the bounding box of all ground points, its HeightAboveGround is set to 0. If true, extrapolation is used to assign the HeightAboveGround value. [Default: false]
- **where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge**  A strategy for merging points skipped by a `where` option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

- **filters.hag_delaunay** (page 165)  Compute pointwise height above ground using triangulation. Requires points to classified as ground/non-ground prior to estimating.

- **filters.hag_dem** (page 168)  Compute pointwise height above GDAL-readable DEM raster.
**filters.hag_nn (page 171)** Compute pointwise height above ground estimate. Requires points to be classified as ground/non-ground prior to estimating.

### Colorization

**filters.colorinterp**

The color interpolation filter assigns scaled RGB values from an image based on a given dimension. It provides three possible approaches:

1. You provide a minimum (page 177) and maximum (page 177), and the data are scaled for the given dimension (page 177) accordingly.
2. You provide a $k$ (page 178) and a mad (page 178) setting, and the scaling is set based on Median Absolute Deviation.
3. You provide a $k$ (page 178) setting and the scaling is set based on the $k$ (page 178)-number of standard deviations from the median.

You can provide your own GDAL (http://www.gdal.org)-readable image for the scale color factors, but a number of pre-defined ramps are embedded in PDAL. The default ramps provided by PDAL are 256x1 RGB images, and might be a good starting point for creating your own scale factors. See Default Ramps (page 176) for more information.

---

**Note:** filters.colorinterp (page 175) will use the entire band to scale the colors.

---

**Default Embedded Stage**

This stage is enabled by default.

---

**Example**

```json
{
    "uncolored.las",
    {
        "type": "filters.colorinterp",
        "ramp": "pestel_shades",
        "mad": true,
        "k": 1.8,
        "dimension": "Z"
    }
}
```

(continues on next page)
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

"colorized.las"

Fig. 4: Image data with interpolated colors based on Z dimension and `pestel_shades` ramp.

**Default Ramps**

PDAL provides a number of default color ramps you can use in addition to providing your own. Give the ramp name as the `ramp` (page 177) option to the filter and it will be used. Otherwise, provide a GDAL (http://www.gdal.org)-readable raster filename.

`awesome_green`

`black_orange`
**Options**

**ramp**  The raster file to use for the color ramp. Any format supported by GDAL (http://www.gdal.org) may be read. Alternatively, one of the default color ramp names can be used. [Default: “pestel_shades”]

**dimension**  A dimension name to use for the values to interpolate colors. [Default: “Z”]

**minimum**  The minimum value to use to scale the data. If none is specified, one is computed from the data. If one is specified but a $k$ (page 178) value is also provided, the $k$ (page 178) value will be used.

**maximum**  The maximum value to use to scale the data. If none is specified, one is computed from the data. If one is specified but a $k$ (page 178) value is also provided, the $k$ (page 178) value will be used.

**invert**  Invert the direction of the ramp? [Default: false]
k  Color based on the given number of standard deviations from the median. If set, minimum (page 177) and maximum (page 177) will be computed from the median and setting them will have no effect.

mad  If true, minimum (page 177) and maximum (page 177) will be computed by the median absolute deviation. See filters.mad (page 237) for discussion. [Default: false]

mad_multiplier  MAD threshold multiplier. Used in conjunction with k (page 178) to threshold the differencing. [Default: 1.4862]

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.colorization

The colorization filter populates dimensions in the point buffer using input values read from a raster file. Commonly this is used to add Red/Green/Blue values to points from an aerial photograph of an area. However, any band can be read from the raster and applied to any dimension name desired.

![Image of a stadium with colors provided by the input image](image)

Fig. 5: After colorization, points take on the colors provided by the input image

Note:  GDAL (http://www.gdal.org) is used to read the color information and any
GDAL-readable supported format (https://www.gdal.org/formats_list.html) can be read.

The bands of the raster to apply to each are selected using the “band” option, and the values of the band may be scaled before being written to the dimension. If the band range is 0-1, for example, it might make sense to scale by 256 to fit into a traditional 1-byte color value range.

**Default Embedded Stage**

This stage is enabled by default

**Streamable Stage**

This stage supports streaming operations

**Example**

```json
[
  "uncolored.las",
  {
    "type": "filters.colorization",
    "dimensions": "Red:1:1.0, Blue, Green::256.0",
    "raster": "aerial.tif"
  },
  "colorized.las"
]
```

**Considerations**

Certain data configurations can cause degenerate filter behavior. One significant knob to adjust is the GDAL_CACHEMAX environment variable. One driver which can have issues is when a TIFF (http://www.gdal.org/frmt_gtiff.html) file is striped vs. tiled. GDAL’s data access in that situation is likely to cause lots of re-reading if the cache isn’t large enough.

Consider a striped TIFF file of 286mb:

```
-rw-r-----@ 1 hobu staff  286M Oct 29 16:58 orth-striped.tif
```

```json
[
  "colourless.laz",
  {
  }
]
```
Simple application of the `filters.colorization` (page 178) using the striped TIFF (http://www.gdal.org/frmt_gtiff.html) with a 268mb `readers.las` (page 71) file will take nearly 1:54.

```
[hobu@pyro knudsen (master)]$ time ~/dev/git/pdal/bin/pdal pipeline -i striped.json
real 1m53.477s
user 1m20.018s
sys 0m33.397s
```

Setting the `GDAL_CACHEMAX` variable to a size larger than the TIFF file dramatically speeds up the color fetching:

```
[hobu@pyro knudsen (master)]$ export GDAL_CACHEMAX=500
[hobu@pyro knudsen (master)]$ time ~/dev/git/pdal/bin/pdal pipeline -i striped.json
real 0m19.034s
user 0m15.557s
sys 0m1.102s
```

### Options

- **raster** The raster file to read the band from. Any format (https://www.gdal.org/formats_list.html) supported by GDAL (http://www.gdal.org) may be read.

- **dimensions** A comma separated list of dimensions to populate with values from the raster file. Dimensions will be created if they don’t already exist. The format of each dimension is `<name>`:<band_number>:<scale_factor>. Either or both of band number and scale factor may be omitted as may ‘:’ separators if the data is not ambiguous. If not supplied, band numbers begin at 1 and increment from the band number of the previous dimension. If not supplied, the scaling factor is 1.0. [Default: “Red:1:1.0, Green:2:1.0, Blue:3:1.0”]

- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in
standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: `auto`]

`filters.colorinterp` (page 175) Assign RGB colors based on a dimension and a ramp

`filters.colorization` (page 178) Fetch and assign RGB color information from a GDAL-readable datasource.

**Clustering**

`filters.cluster`

The Cluster filter first performs Euclidean Cluster Extraction on the input `PointView` and then labels each point with its associated cluster ID. It creates a new dimension `ClusterID` that contains the cluster ID value. Cluster IDs start with the value 1. Points that don’t belong to any cluster will be given a cluster ID of 0.

**Default Embedded Stage**

This stage is enabled by default

**Example**

```json
[
  { "input.las",
    { "type":"filters.cluster"
    },
    { "type":"writers.bpf",
      "filename":"output.bpf",
      "output_dims":"X,Y,Z,ClusterID"
    }
  ]
```
Options

**min_points** Minimum number of points to be considered a cluster. [Default: 1]

**max_points** Maximum number of points to be considered a cluster. [Default: $2^{64} - 1$]

**tolerance** Cluster tolerance - maximum Euclidean distance for a point to be added to the cluster. [Default: 1.0]

**is3d** By default, clusters are formed by considering neighbors in a 3D sphere, but if **is3d** is set to false, it will instead consider neighbors in a 2D cylinder (XY plane only). [Default: true]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a `where` option when running in standard mode. If **true**, the skipped points are added to the first point view returned by the skipped filter. If **false**, skipped points are placed in their own point view. If **auto**, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.dbscan**

The DBSCAN filter performs Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [Ester1996] and labels each point with its associated cluster ID. Points that do not belong to a cluster are given a Cluster ID of -1. The remaining clusters are labeled as integers starting from 0.

Default Embedded Stage

This stage is enabled by default

New in version 2.1.

Example

```json
[
    "input.las",
    {
        "type": "filters.dbscan",
        "min_points": 10,
        "eps": 2.0,
    }
]```

(continues on next page)
Options

**min_points**  The minimum cluster size `min_points` should be greater than or equal to the number of dimensions (e.g., X, Y, and Z) plus one. As a rule of thumb, two times the number of dimensions is often used. [Default: 6]

**eps**  The epsilon parameter can be estimated from a k-distance graph (for k = \( \text{min_points} \) minus one). `eps` defines the Euclidean distance that will be used when searching for neighbors. [Default: 1.0]

**dimensions**  Comma-separated string indicating dimensions to use for clustering. [Default: X,Y,Z]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.litree**

The purpose of the Li tree filter is to segment individual trees from an input PointView. In the output PointView points that are deemed to be part of a tree are labeled with a TreeID. Tree IDs start at 1, with non-tree points given a TreeID of 0.

**Note:**  The filter differs only slightly from the paper in the addition of a few conditions on size of tree, minimum height above ground for tree seeding, and flexible radius for non-tree seed insertion.
Default Embedded Stage

This stage is enabled by default

Example

```json
[
  "input.las",
  {
    "type": "filters.litree",
    "min_points": 50,
    "min_height": 10.0,
    "radius": 200.0
  },
  {
    "type": "writers.las",
    "filename": "output.laz",
    "minor_version": 1.4,
    "extra_dims": "all"
  }
]
```

Options

- **min_points** Minimum number of points in a tree cluster. [Default: 10]
- **min_height** Minimum height above ground to start a tree cluster. [Default: 3.0]
- **radius** The seed point for the non-tree cluster is the farthest point in a 2D Euclidean sense from the seed point for the current tree. [Default: 100.0]
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.lloydkmeans

K-means clustering using Lloyd’s algorithm labels each point with its associated cluster ID (starting at 0).

Default Embedded Stage

This stage is enabled by default

New in version 2.1.

Example

```
[  
  "input.las",
  {
    "type":"filters.lloydkmeans",
    "k":10,
    "maxiters":20,
    "dimensions":"X,Y,Z"
  },
  {
    "type":"writers.las",
    "filename":"output.laz",
    "minor_version":4,
    "extra_dims":"all"
  }
]
```

Options

- **k** The desired number of clusters. [Default: 10]
- **maxiters** The maximum number of iterations. [Default: 10]
- **dimensions** Comma-separated string indicating dimensions to use for clustering. [Default: X,Y,Z]
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto,
skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.cluster** (page 181) Extract and label clusters using Euclidean distance metric. Returns a new dimension `ClusterID` that indicates the cluster that a point belongs to. Points not belonging to a cluster are given a cluster ID of 0.

**filters.dbscan** (page 182) Perform Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [Ester1996].

**filters.litree** (page 183) Segment and label individual trees. Returns a new dimension `TreeID` that indicates the tree that a point belongs to. `TreeID` starts at 1, with non-tree points given a `TreeID` of 0. [Li2012].

**filters.lloydkmeans** (page 185) Perform K-means clustering using Lloyd’s algorithm. Returns a new dimension `ClusterID` with each point being assigned to a cluster. `ClusterID` starts at 0. [Lloyd1982].

**Pointwise Features**

**filters.approximatecoplanar**

The **approximate coplanar filter** implements a portion of the algorithm presented in [Limberger2015]. Prior to clustering points, the authors first apply an approximate coplanarity test, where points that meet the following criteria are labeled as approximately coplanar.

\[
\lambda_2 > (s_\alpha \lambda_1) \& \& (s_\beta \lambda_2) > \lambda_3
\]

\(\lambda_1, \lambda_2, \lambda_3\) are the eigenvalues of a neighborhood of points (defined by knn nearest neighbors) in ascending order. The threshold values \(s_\alpha\) and \(s_\beta\) are user-defined and default to 25 and 6 respectively.

The filter returns a point cloud with a new dimension `Coplanar` that indicates those points that are part of a neighborhood that is approximately coplanar (1) or not (0).

**Default Embedded Stage**

This stage is enabled by default
Example

The sample pipeline presented below estimates the planarity of a point based on its eight nearest neighbors using the approximate coplanar filter. A filters.range (page 248) stage then filters out any points that were not deemed to be coplanar before writing the result in compressed LAZ.

```
[  
    "input.las",
    {
      "type": "filters.approximatecoplanar",
      "knn": 8,
      "thresh1": 25,
      "thresh2": 6
    },
    {
      "type": "filters.range",
      "limits": "Coplanar[1:1]"
    },
    "output.laz"
]
```

Options

**knn** The number of k-nearest neighbors. [Default: 8]

**thresh1** The threshold to be applied to the smallest eigenvalue. [Default: 25]

**thresh2** The threshold to be applied to the second smallest eigenvalue. [Default: 6]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.covariancefeatures

This filter implements various local feature descriptors that are based on the covariance matrix of a point’s neighborhood.

The user can pick a set of feature descriptors by setting the feature_set option. The *dimensionality* (page 190) set of feature descriptors introduced below is the default. The user can also provide a comma-separated list of features to explicitly itemize those covariance features they wish to be computed. This can be combined with any supported presets like “Dimensionality”. Specifying “all” will compute all available features.

Supported features include:

- Anisotropy
- DemantkeVerticality
- Density
- Eigenentropy
- Linearity
- Omnivariance
- Planarity
- Scattering
- EigenvalueSum
- SurfaceVariation
- Verticality

**Note:** Density requires both OptimalKNN and OptimalRadius which can be computed by running `filters.optimalneighborhood` (page 199) prior to `filters.covariancefeatures`.

**Example #1**

```json
[
    "input.las",
    {
        "type": "filters.covariancefeatures",
        "knn": 8,
        "threads": 2,
        "feature_set": "Dimensionality"
    }
]
```
Example #2

```json
[
  "input.las",
  {
    "type": "filters.optimalneighborhood"
  },
  {
    "type": "filters.covariancefeatures",
    "knn": 8,
    "threads": 2,
    "optimized": true,
    "feature_set": "Linearity, Omnipervariance, Density"
  },
  {
    "type": "writers.las",
    "minor_version": 4,
    "extra_dims": "all",
    "forward": "all",
    "filename": "output.las"
  }
]
```

Options

**knn**  The number of k nearest neighbors used for calculating the covariance matrix. [Default: 10]

**threads**  The number of threads used for computing the feature descriptors. [Default: 1]

**feature_set**  A comma-separated list of individual features or feature presets (e.g., “Dimensionality”) to be computed. To compute all available features, specify “all”. [Default: “Dimensionality”]
**stride**  When finding k nearest neighbors, stride determines the sampling rate. A stride of 1 retains each neighbor in order. A stride of two selects every other neighbor and so on. [Default: 1]

**min_k**  Minimum number of neighbors in radius (radius search only). [Default: 3]

**radius**  If radius is specified, neighbors will be obtained by radius search rather than k nearest neighbors, subject to meeting the minimum number of neighbors (min_k).

**mode**  By default, features are computed using the standard deviation along each eigenvector, i.e., using the square root of the computed eigenvalues (mode="SQRT"). mode also accepts “Normalized” which normalizes eigenvalue such that they sum to one, or “Raw” such that the eigenvalues are used directly. [Default: “SQRT”]

**optimized**  optimized can be set to true to enable computation of features using precomputed optimal neighborhoods (found in the OptimalKNN dimension). Requires filters.optimalneighborhood (page 199) be run prior to this stage. [Default: false]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

### Dimensionality feature set

The features introduced in [Demantke2011] describe the shape of the neighborhood, indicating whether the local geometry is more linear (1D), planar (2D) or volumetric (3D) while the one introduced in [Guinard2017] adds the idea of a structure being vertical.

The dimensionality filter introduces the following four descriptors that are computed from the covariance matrix of a point’s neighbors (as defined by knn or radius):

- **linearity** - higher for long thin strips
- **planarity** - higher for planar surfaces
- **scattering** - higher for complex 3d neighbourhoods
- **verticality** - higher for vertical structures, highest for thin vertical strips

It introduces four new dimensions that hold each one of these values: **Linearity**, **Planarity**, **Scattering** and **Verticality**.

---

190  Chapter 7. Drivers
filters.eigenvalues

The eigenvalue filter returns the eigenvalues for a given point, based on its k-nearest neighbors. The filter produces three new dimensions (Eigenvalue0, Eigenvalue1, and Eigenvalue2), which can be analyzed directly, or consumed by downstream stages for more advanced filtering. The eigenvalues are sorted in ascending order.

The eigenvalue decomposition is performed using Eigen’s SelfAdjointEigenSolver (https://eigen.tuxfamily.org/dox/classEigen_1_1SelfAdjointEigenSolver.html).

Default Embedded Stage

This stage is enabled by default

Example

This pipeline demonstrates the calculation of the eigenvalues. The newly created dimensions are written out to BPF for further inspection.

```json
[
  "input.las",
  {
    "type": "filters.eigenvalues",
    "knn": 8
  },
  {
    "type": "writers.bpf",
    "filename": "output.bpf",
    "output_dims": "X,Y,Z,Eigenvalue0,Eigenvalue1,Eigenvalue2"
  }
]
```

Options

**knn** The number of k-nearest neighbors. [Default: 8]

**normalize** Normalize eigenvalues such that the sum is 1. [Default: false]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by
the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.estimaterank

The rank estimation filter uses singular value decomposition (SVD) to estimate the rank of a set of points. Point sets with rank 1 correspond to linear features, while sets with rank 2 correspond to planar features. Rank 3 corresponds to a full 3D feature. In practice this can be used alone, or possibly in conjunction with other filters to extract features (e.g., buildings, vegetation).

Two parameters are required to estimate rank (though the default values will be suitable in many cases). First, the knn (page 193) parameter defines the number of points to consider when computing the SVD and estimated rank. Second, the thresh (page 193) parameter is used to determine when a singular value shall be considered non-zero (when the absolute value of the singular value is greater than the threshold).

The rank estimation is performed on a pointwise basis, meaning for each point in the input point cloud, we find its knn (page 193) neighbors, compute the SVD, and estimate rank. The filter creates a new dimension called Rank that can be used downstream of this filter stage in the pipeline. The type of writer used will determine whether or not the Rank dimension itself can be saved to disk.

Default Embedded Stage

This stage is enabled by default

Example

This sample pipeline estimates the rank of each point using this filter and then filters out those points where the rank is three using filters.range (page 248).

```json
[  
  "input.las",
  {
    "type":"filters.estimaterank",
    "knn":8,
    "thresh":0.01
  },
  {
    "type":"filters.range",
    "values":{0,3}
  }
]
```

(continues on next page)
Options

**knn** The number of k-nearest neighbors. [Default: 8]

**thresh** The threshold used to identify nonzero singular values. [Default: 0.01]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.lof**

The **Local Outlier Factor (LOF) filter** was introduced as a method of determining the degree to which an object is an outlier. This filter is an implementation of the method described in [Breunig2000].

The filter creates three new dimensions, **KDistance**, **LocalReachabilityDistance** and **LocalOutlierFactor**, all of which are double-precision floating values. The **KDistance** dimension records the Euclidean distance between a point and it’s k-th nearest neighbor (the number of k neighbors is set with the **minpts** (page 194) option). The **LocalReachabilityDistance** is the inverse of the mean of all reachability distances for a neighborhood of points. This reachability distance is defined as the max of the Euclidean distance to a neighboring point and that neighbor’s own previously computed **KDistance**. Finally, each point has a **LocalOutlierFactor** which is the mean of all **LocalReachabilityDistance** values for the neighborhood. In each case, the neighborhood is the set of k nearest neighbors.

In practice, setting the **minpts** (page 194) parameter appropriately and subsequently filtering outliers based on the computed **LocalOutlierFactor** can be difficult. The authors present some work on establishing upper and lower bounds on LOF values, and provide some guidelines on selecting **minpts** (page 194) values, which users of this filter should find instructive.
Note: To inspect the newly created, non-standard dimensions, be sure to write to an output format that can support arbitrary dimensions, such as BPF.

Default Embedded Stage

This stage is enabled by default

Example

The sample pipeline below computes the LOF with a neighborhood of 20 neighbors, followed by a range filter to crop out points whose LocalOutlierFactor exceeds 1.2 before writing the output.

```json
[  
  "input.las",
  
  
  
  "type": "filters.lof",
  "minpts": 20
  
  
  
  },

  
  
  "type": "filters.range",
  "limits": "LocalOutlierFactor[:1.2]"
  

  ],
  "output.laz"
]
```

Options

minpts The number of k nearest neighbors. [Default: 10]

where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.miniball

The Miniball Criterion was introduced in [Weyrich2004] and is based on the assumption that points that are distant to the cluster built by their k-neighborhood are likely to be outliers. First, the smallest enclosing ball is computed for the k-neighborhood, giving a center point and radius [Fischer2010]. The miniball criterion is then computed by comparing the distance (from the current point to the miniball center) to the radius of the miniball.

The author suggests that the Miniball Criterion is more robust than the Plane Fit Criterion (page 200) around high-frequency details, but demonstrates poor outlier detection for points close to a smooth surface.

The filter creates a single new dimension, Miniball, that records the Miniball criterion for the current point.

Note: To inspect the newly created, non-standard dimensions, be sure to write to an output format that can support arbitrary dimensions, such as BPF.

Default Embedded Stage

This stage is enabled by default

Example

The sample pipeline below computes the Miniball criterion with a neighborhood of 8 neighbors. We do not apply a fixed threshold to single out outliers based on the Miniball criterion as the range of values can vary from one dataset to another. In general, higher values indicate the likelihood of a point being an outlier.

```json
[
    "input.las",
    {
        "type": "filters.miniball",
        "knn": 8
    },
    "output.laz"
]
```
Options

**knn**  The number of k nearest neighbors. [Default: 8]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: `auto`]

**filters.nndistance**

The NNDistance filter runs a 3-D nearest neighbor algorithm on the input cloud and creates a new dimension, `NNDistance`, that contains a distance metric described by the `mode` (page 197) of the filter.

**Default Embedded Stage**

This stage is enabled by default

**Example**

```json
[
    "input.las",
    {
        "type": "filters.nndistance",
        "k": 8
    },
    {
        "type": "writers.bpf",
        "filename": "output.las",
        "output_dims": "X,Y,Z,NNDistance"
    }
]
```
Options

**mode**  The mode of operation. Either “kth”, in which the distance is the euclidian distance of the subject point from the kth remote point or “avg” in which the distance is the average euclidian distance from the \( k \) (page 197) nearest points. [Default: ‘kth’]

**k**  The number of \( k \) nearest neighbors to consider. [Default: 10]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.normal**

The **normal filter** returns the estimated normal and curvature for a collection of points. The algorithm first computes the eigenvalues and eigenvectors of the collection of points, which is comprised of the \( k \)-nearest neighbors. The normal is taken as the eigenvector corresponding to the smallest eigenvalue. The curvature is computed as

\[
\text{curvature} = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2}
\]

where \( \lambda_i \) are the eigenvalues sorted in ascending order.

The filter produces four new dimensions (NormalX, NormalY, NormalZ, and Curvature), which can be analyzed directly, or consumed by downstream stages for more advanced filtering.

The eigenvalue decomposition is performed using Eigen’s [SelfAdjointEigenSolver](https://eigen.tuxfamily.org/dox/classEigen_1_1SelfAdjointEigenSolver.html).

normals will be automatically flipped towards positive \( Z \), unless the **always_up** (page 198) flag is set to false. Users can optionally set any of the XYZ coordinates to specify a custom viewpoint (page 198) or set them all to zero to effectively disable the normal flipping.

**Note:** By default, the Normal filter will invert normals such that they are always pointed “up” (positive \( Z \)). If the user provides a viewpoint (page 198), normals will instead be inverted such that they are oriented towards the viewpoint, regardless of the always_up (page 198) flag. To disable all normal flipping, do not provide a viewpoint (page 198) and set always_up (page 198) to false.

7.4. Filters
In addition to *always_up* (page 198) and *viewpoint* (page 198), users can run a refinement step (off by default) that propagates normals using a minimum spanning tree. The propagated normals can lead to much more consistent results across the dataset.

**Note:** To enable normal propagation, users can set *refine* (page 198) to *true*.

**Default Embedded Stage**

This stage is enabled by default

**Example**

This pipeline demonstrates the calculation of the normal values (along with curvature). The newly created dimensions are written out to BPF for further inspection.

```json
[
    "input.las",
    {
        "type": "filters.normal",
        "knn": 8
    },
    {
        "type": "writers.bpf",
        "filename": "output.bpf",
        "output_dims": "X,Y,Z,NormalX,NormalY,NormalZ,Curvature"
    }
]
```

**Options**

- **knn** The number of k-nearest neighbors. [Default: 8]
- **viewpoint** A single WKT or GeoJSON 3D point. Normals will be inverted such that they are all oriented towards the viewpoint.
- **always_up** A flag indicating whether or not normals should be inverted only when the Z component is negative. [Default: true]
- **refine** A flag indicating whether or not to reorient normals using minimum spanning tree propagation. [Default: false]
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
where_merge A strategy for merging points skipped by a ‘where’ option when running in
standard mode. If true, the skipped points are added to the first point view returned by
the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view
is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.optimalneighborhood

The Optimal Neighborhood filter computes the eigenentropy (defined as the Shannon
entropy of the normalized eigenvalues) for a neighborhood of points in the range min_k to
max_k. The neighborhood size that minimizes the eigenentropy is saved to a new dimension
OptimalKNN. The corresponding radius of the neighborhood is saved to OptimalRadius.
These dimensions can be written to an output file or utilized directly by
filters.covariancefeatures (page 188).

Default Embedded Stage

This stage is enabled by default

Example

```json
[  
    "input.las",
    
    {  
        "type": "filters.optimalneighborhood",
        "min_k": 8,
        "max_k": 50
    },
    
    {  
        "type": "writers.las",
        "minor_version": 4,
        "extra_dims": "all",
        "forward": "all",
        "filename": "output.las"
    }
]
```
Options

**min_k**  The minimum number of k nearest neighbors to consider for optimal neighborhood selection. [Default: 10]

**max_k**  The maximum number of k nearest neighbors to consider for optimal neighborhood selection. [Default: 100]

**filters.planefit**

The **Plane Fit Criterion** was introduced in [Weyrich2004] and computes the deviation of a point from a manifold approximating its neighbors. First, a plane is fit to each point’s k-neighborhood by performing an eigenvalue decomposition. Next, the mean point to plane distance is computed by considering all points within the neighborhood. This is compared to the point to plane distance of the current point giving rise to the k-neighborhood. As the mean distance of the k-neighborhood approaches 0, the Plane Fit criterion will tend toward 1. As point to plane distance of the current point approaches 0, the Plane Fit criterion will tend toward 0.

The author suggests that the Plane Fit Criterion is well suited to outlier detection when considering noisy reconstructions of smooth surfaces, but produces poor results around small features and creases.

The filter creates a single new dimension, PlaneFit, that records the Plane Fit criterion for the current point.

---

**Note:** To inspect the newly created, non-standard dimensions, be sure to write to an output format that can support arbitrary dimensions, such as BPF.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Example**

The sample pipeline below computes the Plane Fit criterion with a neighborhood of 8 neighbors. We do not apply a fixed threshold to single out outliers based on the Plane Fit criterion as the range of values can vary from one dataset to another. In general, higher values indicate the likelihood of a point being an outlier.
Options

**knn**  The number of k nearest neighbors. [Default: 8]

**threads**  The number of threads used for computing the plane fit criterion. [Default: 1]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.radialdensity**

The **Radial Density filter** creates a new attribute `RadialDensity` that contains the density of points in a sphere of given radius.

The density at each point is computed by counting the number of points falling within a sphere of given `radius` (page 202) (default is 1.0) and centered at the current point. The number of neighbors (including the query point) is then normalized by the volume of the sphere, defined as

\[ V = \frac{4}{3} \pi r^3 \]

The radius \( r \) can be adjusted by changing the `radius` (page 202) option.

**Default Embedded Stage**

This stage is enabled by default

---

7.4. Filters
Example

```json
[
  "input.las",
  {
    "type": "filters.radialdensity",
    "radius": 2.0
  },
  {
    "type": "writers.bpf",
    "filename": "output.bpf",
    "output_dims": "X,Y,Z,RadialDensity"
  }
]
```

Options

**radius** Radius. [Default: 1.0]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a `where` option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.reciprocity**

The Nearest-Neighbor Reciprocity Criterion was introduced in [Weyrich2004] and is based on a simple assumption, that valid points may be in the k-neighborhood of an outlier, but the outlier will most likely not be part of the valid point’s k-neighborhood.

The author suggests that the Nearest-Neighbor Reciprocity Criterion is more robust than both the Plane Fit (page 200) and Miniball (page 195) Criterion, being equally sensitive around smooth and detailed regions. The criterion does however produce invalid results near manifold borders.

The filter creates a single new dimension, Reciprocity, that records the percentage of points (in the range 0 to 100) that are considered uni-directional neighbors of the current point.

**Note:** To inspect the newly created, non-standard dimensions, be sure to write to an output
format that can support arbitrary dimensions, such as BPF.

**Default Embedded Stage**

This stage is enabled by default

**Example**

The sample pipeline below computes reciprocity with a neighborhood of 8 neighbors, followed by a range filter to crop out points whose Reciprocity percentage is less than 98% before writing the output.

```json
[
  "input.las",
  {
    "type"="filters.reciprocity",
    "knn":8
  },
  {
    "type"="filters.range",
    "limits":"Reciprocity[:98.0]"
  },
  "output.laz"
]
```

**Options**

- **knn** The number of k nearest neighbors. [Default: 8]
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.approximatecoplanar** (page 186) Estimate pointwise planarity, based on k-nearest neighbors. Returns a new dimension Coplanar where a value of 1 indicates that a point is part of a coplanar neighborhood (0 otherwise).
**filters.covariancefeatures** *(page 188)* Filter that calculates local features based on the covariance matrix of a point’s neighborhood.

**filters.eigenvalues** *(page 191)* Compute pointwise eigenvalues, based on k-nearest neighbors.

**filters.estimaterank** *(page 192)* Compute pointwise rank, based on k-nearest neighbors.

**filters.lof** *(page 193)* Compute pointwise Local Outlier Factor (along with K-Distance and Local Reachability Distance).

**filters.miniball** *(page 195)* Compute a criterion for point neighbors based on the miniball algorithm.

**filters.nndistance** *(page 196)* Compute a distance metric based on nearest neighbors.

**filters.normal** *(page 197)* Compute pointwise normal and curvature, based on k-nearest neighbors.

**filters.optimalneighborhood** *(page 199)* Compute optimal k nearest neighbors and corresponding radius by minimizing pointwise eignentropy. Creates two new dimensions **OptimalKNN** and **OptimalRadius**.

**filters.planefit** *(page 200)* Compute a deviation of a point from a manifold approximating its neighbors.

**filters.radialdensity** *(page 201)* Compute pointwise density of points within a given radius.

**filters.reciprocity** *(page 202)* Compute the percentage of points that are considered uni-directional neighbors of a point.

**Assignment**

**filters.assign**

The assign filter allows you set the value of a dimension for all points to a provided value that pass a range filter.

**Default Embedded Stage**

This stage is enabled by default

**Streamable Stage**

This stage supports streaming operations
Note: The assignment and condition options are deprecated and may be removed in a future release.

Options

assignment A range (page 249) followed by an assignment of a value (see example). Can be specified multiple times. The assignments are applied sequentially to the dimension value as set when the filter began processing. [Required]

condition A list of ranges (page 249) that a point’s values must pass in order for the assignment to be performed. [Default: none]

value A list of assignment expressions (page 205) to be applied to points. The list of values is evaluated in order. [Default: none]

where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

Assignment Expressions

The assignment expression syntax is an expansion on the PDAL expression syntax that provides for assignment of values to points. The generic expression is:

```
"value" : "Dimension = ValueExpression [WHERE ConditionalExpression]"
```

Dimension is the name of a PDAL dimension.

A ValueExpression consists of constants, dimension names and mathematical operators that evaluates to a numeric value. The supported mathematical operations are addition(+), subtraction(-), multiplication(*) and division(/).

A ConditionalExpression is an optional boolean value that must evaluate to true for the ValueExpression to be applied.
Example 1

"value" : "Red = Red / 256"

This scales the Red value by 1/256. If the input values are in the range 0 - 65535, the output value will be in the range 0 - 255.

Example 2

"value" :
[
   "Classification = 2 WHERE HeightAboveGround < 5",
   "Classification = 1 WHERE HeightAboveGround >= 5"
]

This sets the classification of points to either Ground or Unassigned depending on the value of the HeightAboveGround dimension.

Example 3

"value" :
[
   "X = 1",
   "X = 2 WHERE X > 10"
]

This sets the value of X for all points to 1. The second statement is essentially ignored since the first statement sets the X value of all points to 1 and therefore no points the ConditionalExpression of the second statement.

defilters.overlay

The overlay filter allows you to set the values of a selected dimension based on an OGR-readable polygon or multi-polygon.

Default Embedded Stage

This stage is enabled by default
OGR SQL support

You can limit your queries based on OGR’s SQL support. If the filter has both a `datasource` (page 208) and a `query` (page 208) option, those will be used instead of the entire OGR data source. At this time it is not possible to further filter the OGR query based on a geometry but that may be added in the future.

**Note:** The OGR SQL support follows the rules specified in [ExecuteSQL](http://www.gdal.org/ogr__api_8h.html#a9892ecb0bf61add295bd9decdb13797a) documentation, and it will pass SQL down to the underlying datasource if it can do so.

**Example 1**

In this scenario, we are altering the attributes of the dimension `Classification`. Points from `autzen-dd.las` that lie within a feature will have their classification to match the `CLS` field associated with that feature.

```
[  
  "autzen-dd.las",
  
  
  {
    "type":"filters.overlay",
    "dimension":"Classification",
    "datasource":"attributes.shp",
    "layer":"attributes",
    "column":"CLS"
  },

  
  {
    "filename":"attributed.las",
    "scale_x":0.0000001,
    "scale_y":0.0000001
  }
]
```

**Example 2**

This example sets the Intensity attribute to `CLS` values read from the OGR SQL ([http://www.gdal.org/ogr_sql_sqlite.html](http://www.gdal.org/ogr_sql_sqlite.html)) query.

```
[  
  "autzen-dd.las",
  
  
]
```
"type": "filters.overlay",
"dimension": "Intensity",
"datasource": "attributes.shp",
"query": "SELECT CLS FROM attributes where cls!=6",
"column": "CLS"
},
"attributed.las"
]

Options

dimension Name of the dimension whose value should be altered. [Required]
datasource OGR-readable datasource for Polygon or MultiPolygon data. [Required]
column The OGR datasource column from which to read the attribute. [Default: first column]
query OGR SQL query to execute on the datasource to fetch geometry and attributes. The entire layer is fetched if no query is provided. [Default: none]
layer The data source’s layer to use. [Default: first layer]
where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.assign (page 204) Assign values for a dimension range to a specified value.

filters.overlay (page 206) Assign values to a dimension based on the extent of an OGR-readable data source or an OGR SQL query.

Dimension Create/Copy

filters.ferry

The ferry filter copies data from one dimension to another, creates new dimensions or both.

The filter is guided by a list of ‘from’ and ‘to’ dimensions in the format <from>=><to>. Data from the ‘from’ dimension is copied to the ‘to’ dimension. The ‘from’ dimension must exist. The ‘to’ dimension can be pre-existing or will be created by the ferry filter.
Alternatively, the format `=><to>` can be used to create a new dimension without copying data from any source. The values of the ‘to’ dimension are default initialized (set to 0).

### Default Embedded Stage

This stage is enabled by default

### Streamable Stage

This stage supports streaming operations

### Example 1

In this scenario, we are making copies of the X and Y dimensions into the dimensions `StatePlaneX` and `StatePlaneY`. Since the reprojection filter will modify the dimensions X and Y, this allows us to maintain both the pre-reprojection values and the post-reprojection values.

```
[  
    "uncompressed.las",
      
    {  
      "type": "readers.las",
         "spatialreference": "EPSG:2993",
         "filename": "/las/1.2-with-color.las"
     },
     
    {  
      "type": "filters.ferry",
         "dimensions": "X => StatePlaneX, Y=>StatePlaneY"
     },
     
    {  
      "type": "filters.reprojection",
         "out_srs": "EPSG:4326+4326"
     },
     
    {  
      "type": "writers.las",
         "scale_x": "0.0000001",
         "scale_y": "0.0000001",
         "filename": "colorized.las"
    }
]
```
Example 2

The ferry filter is being used to add a dimension Classification to points so that the value can be set to ‘2’ and written as a LAS file.

```json
[
  {
    "type": "readers.gdal",
    "filename": "somefile.tif"
  },
  {
    "type": "filters.ferry",
    "dimensions": "=>Classification"
  },
  {
    "type": "filters.assign",
    "assignment": "Classification[:]=2"
  },
  "out.las"
]
```

Options

dimensions  A list of dimensions whose values should be copied. The format of the option is `<from>=<to>, <from>=<to>,...` Spaces are ignored. ‘from’ can be left empty, in which case the ‘to’ dimension is created and default-initialized. ‘to’ dimensions will be created if necessary.

Note: the old syntax that used ‘=’ instead of ‘=>’ between dimension names is still supported.

where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.ferry (page 208) Copy data from one dimension to another.
7.4.2 Order

There are currently three PDAL filters that can be used to reorder points. These filters will invalidate an existing KD-tree.

**filters.mortonorder**


It’s also possible to compute a reverse Morton code by reading the binary representation from the end to the beginning. This way, points are sorted with a good dispersement. For example, by successively selecting N representative points within tiles:

See also:

See LOPoCS (https://github.com/Oslandia/lopocs) and pgmorton (https://github.com/Oslandia/pgmorton) for some use case examples of the Reverse Morton algorithm.

**Default Embedded Stage**

This stage is enabled by default
Example

```
[  
    "uncompressed.las",
    {
      "type": "filters.mortonorder",
      "reverse": "false"
    },
    {
      "type": "writers.las",
      "filename": "compressed.laz",
      "compression": "true"
    }
  ]
```

Options

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.randomize**

The randomize filter reorders the points in a point view randomly.

Default Embedded Stage

This stage is enabled by default
Example

```
[  
    "input.las",
    {  
        "type": "filters.randomize"
    },
    {  
        "type": "writers.las",
        "filename": "output.las"
    }
]
```

Options

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.sort**

The sort filter orders a point view based on the values of a *dimension* (page 214). The sorting can be done in increasing (ascending) or decreasing (descending) *order* (page 214).

---

**Default Embedded Stage**

This stage is enabled by default
Example

```json
[
    "unsorted.las",
    {
        "type": "filters.sort",
        "dimension": "X",
        "order": "ASC"
    },
    "sorted.las"
]
```

Options

dimension  The dimension on which to sort the points. [Required]

order  The order in which to sort, ASC or DESC [Default: “ASC”]

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.mortonorder (page 211)  Sort XY data using Morton ordering (aka Z-order/Z-curve).

filters.randomize (page 212)  Randomize points in a view.

filters.sort (page 213)  Sort data based on a given dimension.

7.4.3 Move

PDAL filters that move XYZ coordinates will invalidate an existing KD-tree.
Registration

filters.cpd

The Coherent Point Drift (CPD) filter uses the algorithm of [[MS10]] algorithm to compute a rigid, nonrigid, or affine transformation between datasets. The rigid and affine are what you’d expect; the nonrigid transformation uses Motion Coherence Theory [[YG88]] to “bend” the points to find a best alignment.

Note: CPD is computationally intensive and can be slow when working with many points (i.e. > 10,000). Nonrigid is significantly slower than rigid and affine.

The first input to the change filter are considered the “fixed” points, and all subsequent inputs are “moving” points. The output from the change filter are the “moving” points after the calculated transformation has been applied, one point view per input. Any additional information about the cpd registration, e.g. the rigid transformation matrix, will be placed in the stage’s metadata.

When to use CPD vs ICP

Summarized from the Non-rigid point set registration: Coherent Point Drift (http://graphics.stanford.edu/courses/cs468-07-winter/Papers/nips2006_0613.pdf) paper.

• CPD outperforms the ICP in the presence of noise and outliers by the use of a probabilistic assignment of correspondences between pointsets, which is innately more robust than the binary assignment used in ICP.
• CPD does not work well for large in-plane rotation, such transformation can be first compensated by other well known global registration techniques before CPD algorithm is carried out
• CPD is most effective when estimating smooth non-rigid transformations.

Dynamic Plugin

This stage requires a dynamic plugin to operate
Examples

```json
[
  "fixed.las",
  "moving.las",
  {
    "type": "filters.cpd",
    "method": "rigid"
  },
  "output.las"
]
```

If `method` (page 217) is not provided, the `cpd` filter will default to using the rigid registration method. To get the transform matrix, you’ll need to use the “metadata” option of the pipeline command:

```
$ pdal pipeline cpd-pipeline.json --metadata cpd-metadata.json
```

The metadata output might start something like:

```json
{
  "stages":
  {
    "filters.cpd":
    {
      "iterations": 10,
      "method": "rigid",
      "runtime": 0.003839,
      "sigma2": 5.684342128e-16,
      "transform": 
        1 -6.21722e-17 1.30104e-18 5.
        29303e-11 -8.99346e-17 1 2.60209e-19 1.73472e-18 1 -1.53477e-12 0
        0 0 1
    },
  }
}
```

See also:

* `filters.transformation` (page 224) to apply a transform to other points. `filters.icp` (page 217) for deterministic binary point pair assignments.
Options

**method** Change detection method to use. Valid values are “rigid”, “affine”, and “nonrigid”.  
[Default: “rigid”]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression  
skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in  
standard mode. If true, the skipped points are added to the first point view returned by  
the skipped filter. If false, skipped points are placed in their own point view. If auto,  
skipped points are merged into the returned point view provided that only one point view  
is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.icp

The ICP filter uses the Iterative Closest Point (ICP) algorithm to calculate a rigid (rotation and translation) transformation that best aligns two datasets. The first input to the ICP filter is considered the “fixed” points, and all subsequent points are “moving” points. The output from the filter are the “moving” points after the calculated transformation has been applied, one point view per input. The transformation matrix is inserted into the stage’s metadata.

**Note:** ICP requires the initial pose of the two point sets to be adequately close, which is not always possible, especially when the transformation is non-rigid. ICP can handle limited non-rigid transformations but be aware ICP may be unable to escape a local minimum. Consider using CPD instead.

From [[LLW+19]]:

ICP starts with an initial guess of the transformation between the two point sets and then iterates between finding the correspondence under the current transformation and updating the transformation with the newly found correspondence. ICP is widely used because it is rather straightforward and easy to implement in practice; however, its biggest problem is that it does not guarantee finding the globally optimal transformation. In fact, ICP converges within a very small basin in the parameter space, and it easily becomes trapped in local minima. Therefore, the results of ICP are very sensitive to the initialization, especially when high levels of noise and large proportions of outliers exist.
Examples

```
[
  "fixed.las",
  "moving.las",
  {
    "type": "filters.icp"
  },
  "output.las"
]
```

To get the transform matrix, you’ll need to use the --metadata option from the pipeline command:

```
$ pdal pipeline icp-pipeline.json --metadata icp-metadata.json
```

The metadata output might start something like:

```
{
  "stages": {
    "filters.icp": {
      "centroid": "583394 5.2831e+06 498.152",
      "composed": "1 2.60209e-18 -1.97906e-09
                  -0.374999 8.9407e-08 1 5.58794e-09 -0.614662
                  98492e -10 -5.58794e-09 1 0.033234 0
                  0 0 1",
      "converged": true,
      "fitness": 0.01953125097,
      "transform": "1 2.60209e-18 -1.97906e-09
                   -0.375 8.9407e-08 1 5.58794e-09 -0.5625
                   98492e -10 -5.58794e-09 1 0.00411987 0
                   0 0 1"
    }
  }
```

To apply this transformation to other points, the centroid and transform metadata items can be used with filters.transformation in another pipeline. First, move the centroid of the points to (0,0,0), then apply the transform, then move the points back to the original location. For the above metadata, the pipeline would be similar to:

```
[
  {
    "type": "readers.las",
    "filename": "in.las"
  }
]
```
Note: The composed metadata matrix is a composition of the three transformation steps outlined above, and can be used in a single call to filters.transformation as opposed to the three separate calls.

See also:

filters.transformation (page 224) to apply a transform to other points. filters.cpd (page 215) for the use of a probabilistic assignment of correspondences between pointsets.

Options

max_iter Maximum number of iterations. [Default: 100]
max_similar Max number of similar transforms to consider converged. [Default: 0]
mse_abs Absolute threshold for MSE. [Default: 1e-12]
rt Rotation threshold. [Default: 0.99999]
tt Translation threshold. [Default: 9e-8]
**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

*filters.cpd* (page 215)  Compute and apply transformation between two point clouds using the Coherent Point Drift algorithm.

*filters.icp* (page 217)  Compute and apply transformation between two point clouds using the Iterative Closest Point algorithm.

**Predefined**

*filters.projpipeline*

The projpipeline filter applies a coordinates transformation pipeline. The pipeline could be specified as PROJ string (single step operation or multiple step string starting with +proj=pipeline), a WKT2 string describing a CoordinateOperation, or a “urn:ogc:def:coordinateOperation:EPSG::XXXX” URN.

**Note:** The projpipeline filter does not consider any spatial reference information. However user could specify an output srs, but no check is done to ensure the compliance with the provided transformation pipeline.

**Note:** The projpipeline filter is enabled if the version of GDAL is superior or equal to 3.0

**Streamable Stage**

This stage supports streaming operations
Example

This example shift point on the z-axis.

```
[  
  "untransformed.las",
  {
    "type":"filters.projpipeline",
    "coord_op":"+proj=affine +zoff=100"
  },
  {
    "type":"writers.las",
    "filename":"transformed.las"
  }
]
```

This example apply a shift on the z-axis then reproject from utm 10 to WGS84, using the `reverse_transfo` flag. It also set the output srs

```
[  
  "utm10.las",
  {
    "type":"filters.projpipeline",
    "coord_op":"+proj=pipeline +step +proj=unitconvert +xy_in=deg +xy_out=rad +step +proj=utm +zone=10 +step +proj=affine, +zoff=100",
    "reverse_transfo": "true",
    "out_srs": "EPSG:4326"
  },
  {
    "type":"writers.las",
    "filename":"wgs84.las"
  }
]
```

**Note:** PDAL use the GDAL `OGRCoordinateTransformation` class to transform coordinates. By default output angular unit are in radians. To change to degrees we need to apply a unit conversion step.
Options

**coord_op** The coordinate operation string. Could be specified as PROJ string (single step operation or multiple step string starting with +proj=pipeline), a WKT2 string describing a CoordinateOperation, or a “urn:ogc:def:coordinateOperation:EPSG::XXXX” URN.

**reverse_transfo** Boolean, Whether the coordinate operation should be evaluated in the reverse path [Default: false]

**out_srs** The spatial reference system of the file to be written. Can be an EPSG string (e.g. “EPSG:26910”) or a WKT string. No check is done to ensure the compliance with the specified coordinate operation [Default: Not set]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.reprojection**

The **reprojection filter** converts the X, Y and/or Z dimensions to a new spatial reference system. The old coordinates are replaced by the new ones. If you want to preserve the old coordinates for future processing, use a **filters.ferry** (page 208) to create copies of the original dimensions before reprojecting.

**Note:** When coordinates are reprojected, it may significantly change the precision necessary to represent the values in some output formats. Make sure that you’re familiar with any scaling necessary for your output format based on the projection you’ve used.

**Default Embedded Stage**

This stage is enabled by default

**Streamable Stage**

This stage supports streaming operations
Example 1

This pipeline reprojects terrain points with $Z$-values between 0 and 100 by first applying a range filter and then specifying both the input and output spatial reference as EPSG-codes. The $X$ and $Y$ dimensions are scaled to allow enough precision in the output coordinates.

```json
[  
  {  
    "filename":"input.las",  
    "type":"readers.las",  
    "spatialreference":"EPSG:26916"  
  },  
  {  
    "type":"filters.range",  
    "limits":"Z[0:100],Classification[2:2]"  
  },  
  {  
    "type":"filters.reprojection",  
    "in_srs":"EPSG:26916",  
    "out_srs":"EPSG:4326"  
  },  
  {  
    "type":"writers.las",  
    "scale_x":"0.0000001",  
    "scale_y":"0.0000001",  
    "scale_z":"0.01",  
    "offset_x":"auto",  
    "offset_y":"auto",  
    "offset_z":"auto",  
    "filename":"example-geog.las"  
  }]
```

Example 2

In some cases it is not possible to use a EPSG-code as a spatial reference. Instead Proj.4 (http://proj4.org) parameters can be used to define a spatial reference. In this example the vertical component of points in a laz file is converted from geometric (ellipsoidal) heights to orthometric heights by using the `geoidgrids` parameter from Proj.4. Here we change the vertical datum from the GRS80 ellipsoid to DVR90, the vertical datum in Denmark. In the writing stage of the pipeline the spatial reference of the file is set to EPSG:7416. The last step is needed since PDAL will otherwise reference the vertical datum as “Unnamed Vertical Datum” in the spatial reference VLR.
Options

**in_srs** Spatial reference system of the input data. Express as an EPSG string (eg “EPSG:4326” for WGS84 geographic), Proj.4 string or a well-known text string. [Required if not part of the input data set]

**out_srs** Spatial reference system of the output data. Express as an EPSG string (eg “EPSG:4326” for WGS84 geographic), Proj.4 string or a well-known text string. [Required]

**in_axis_ordering** An array of numbers that override the axis order for the in_srs (or if not specified, the inferred SRS from the previous Stage). “2, 1” for example would swap X and Y, which may be commonly needed for something like “EPSG:4326”.

**out_axis_ordering** An array of numbers that override the axis order for the out_srs. “2, 1” for example would swap X and Y, which may be commonly needed for something like “EPSG:4326”.

**filters.transformation**

The transformation filter applies an arbitrary rotation+translation transformation, represented as a 4x4 matrix (page 225), to each xyz triplet.

The filter does no checking to ensure the matrix is a valid affine transformation.

**Note**: The transformation filter does not apply or consider any spatial reference information.

Default Embedded Stage
This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

This example rotates the points around the z-axis while translating them.

```
[{
    "untransformed.las",
    {
        "type": "filters.transformation",
        "matrix": "0 -1 0 1 1 0 0 2 0 0 1 3 0 0 0 1"
    },
    {
        "type": "writers.las",
        "filename": "transformed.las"
    }
}
```

Options

invert  If set to true, applies the inverse of the provided transformation matrix. [Default: false]

matrix  A whitespace-delimited transformation matrix. The matrix is assumed to be presented in row-major order. Only matrices with sixteen elements are allowed.

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
Further details

A full tutorial about transformation matrices is beyond the scope of this documentation. Instead, we will provide a few pointers to introduce core concepts, especially as pertains to PDAL’s handling of the matrix argument.

Transformations in a 3-dimensional coordinate system can be represented as an affine transformation using homogeneous coordinates. This 4x4 matrix can represent transformations describing operations like translation, rotation, and scaling of coordinates.

The transformation filter’s matrix argument is a space delimited, 16 element string. This string is simply a row-major representation of the 4x4 matrix (i.e., first four elements correspond to the top row of the transformation matrix and so on).

In the event that readers are accustomed to an alternate representation of the transformation matrix, we provide some simple examples in the form of pure translations, rotations, and scaling, and show the corresponding matrix string.

Translation

A pure translation by \( t_x \), \( t_y \), and \( t_z \) in the X, Y, and Z dimensions is represented by the following matrix.

\[
\begin{bmatrix}
1 & 0 & 0 & t_x \\
0 & 1 & 0 & t_y \\
0 & 0 & 1 & t_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The JSON syntax required for such a translation is written as follows for \( t_x = 7 \), \( t_y = 8 \), and \( t_z = 9 \).

```json
[
  {
    "type": "filters.transformation",
    "matrix": "1 0 0 7 0 1 0 8 0 0 1 9 0 0 0 1"
  }
]
```
Scaling

Scaling of coordinates is also possible using a transformation matrix. The matrix shown below will scale the X coordinates by \( s_x \), the Y coordinates by \( s_y \), and Z by \( s_z \).

\[
\begin{pmatrix}
 s_x & 0 & 0 & 0 \\
 0 & s_y & 0 & 0 \\
 0 & 0 & s_z & 0 \\
 0 & 0 & 0 & 1
\end{pmatrix}
\]

We again provide an example JSON snippet to demonstrate the scaling transformation. In the example, X and Y are not scaled at all (i.e., \( s_x = s_y = 1 \)) and Z is magnified by a factor of 2 (\( s_z = 2 \)).

```json
[
  {
    "type": "filters.transformation",
    "matrix": "1 0 0 0 0 1 0 0 0 0 2 0 0 0 0 1"
  }
]
```

Rotation

A rotation of coordinates by \( \theta \) radians counter-clockwise about the z-axis is accomplished with the following matrix.

\[
\begin{pmatrix}
 \cos \theta & -\sin \theta & 0 & 0 \\
 \sin \theta & \cos \theta & 0 & 0 \\
 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 1
\end{pmatrix}
\]

In JSON, a rotation of 90 degrees (\( \theta = 1.57 \) radians) takes the form shown below.

```json
[
  {
    "type": "filters.transformation",
    "matrix": "0 0 -1 0 0 0 0 1 0 0 0 0 1"
  }
]
```

Similarly, a rotation about the x-axis by \( \theta \) radians is represented as

\[
\begin{pmatrix}
 1 & 0 & 0 & 0 \\
 0 & \cos \theta & -\sin \theta & 0 \\
 0 & \sin \theta & \cos \theta & 0 \\
 0 & 0 & 0 & 1
\end{pmatrix}
\]

which takes the following form in JSON for a rotation of 45 degrees (\( \theta = 0.785 \) radians)
Finally, a rotation by \( \theta \) radians about the y-axis is accomplished with the matrix

\[
\begin{pmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

and the JSON string for a rotation of 10 degrees (\( \theta = 0.175 \) radians) becomes

```
[
  {
    "type": "filters.transformation",
    "matrix": "0.985 0 0.174 0 0 1 0 0 -0.174 0 0.985 -> 0 0 0 1"
  }
]
```

**filters.proj pipeline** (page 220)  Apply coordinates operation on point triplets, based on PROJ pipeline string, WKT2 coordinates operations or URN definitions.

**filters.reprojection** (page 222)  Reproject data using GDAL from one coordinate system to another.

**filters.transformation** (page 224)  Transform each point using a 4x4 transformation matrix.
7.4.4 Cull

Some PDAL filters will cull points, returning a point cloud that is smaller than the input. These filters will invalidate an existing KD-tree.

Spatial

filters.crop

The crop filter removes points that fall outside or inside a cropping bounding box (2D or 3D), polygon, or point+distance. If more than one bounding region is specified, the filter will pass all input points through each bounding region, creating an output point set for each input crop region.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

The provided bounding regions are assumed to have the same spatial reference as the points unless the option `a_srs` (page 231) provides an explicit spatial reference for bounding regions. If the point input consists of multiple point views with differing spatial references, one is chosen at random and assumed to be the spatial reference of the input bounding region. In this case a warning will be logged.

Example 1

This example crops an input point cloud using a square polygon.

```
[
    "file-input.las",
    {
        "type": "filters.crop",
        "bounds": "([0,1000000],[0,1000000])"
    },
    {
        "type": "writers.las",
        "filename": "file-cropped.las"
    }
]
```
Example 2

This example crops all points more than 500 units in any direction from a point.

```json
[
   "file-input.las",
   {
      "type": "filters.crop",
      "point": "POINT(0 0 0)",
      "distance": 500
   },
   {
      "type": "writers.las",
      "filename": "file-cropped.las"
   }
]
```

Options

**bounds** The extent of the clipping rectangle in the format "([xmin, xmax], [ymin, ymax])". This option can be specified more than once by placing values in an array.

---

**Note:** 3D bounds can be given in the form ([xmin, xmax], [ymin, ymax], [zmin, zmax]).

---

**Warning:** If a 3D bounds is given to the filter, a 3D crop will be attempted, even if the Z values are invalid or inconsistent with the data.

**polygon** The clipping polygon, expressed in a well-known text string, eg: "POLYGON((0 0, 5000 10000, 10000 0, 0 0))". This option can be specified more than once by placing values in an array.

**outside** Invert the cropping logic and only take points outside the cropping bounds or polygon. [Default: false]

**point** An array of WKT or GeoJSON 2D or 3D points (eg: "POINT(0 0 0)"). Requires `distance` (page 231).
**distance** Distance (radius) in units of common X, Y, and Z Dimensions (page 287) in combination with point (page 230). Passing a 2D point will crop using a circle. Passing a 3D point will crop using a sphere.

**a_srs** Indicates the spatial reference of the bounding regions. If not provided, it is assumed that the spatial reference of the bounding region matches that of the points.

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**Notes**

1. See *Clipping data with polygons* (page 374): and *Clipping with Geometries* (page 327) for example usage scenarios for filters.crop (page 229).

**filters.crop** (page 229) Filter points inside or outside a bounding box or a polygon

**Resampling**

**filters.decimation**

The decimation filter retains every Nth point from an input point view.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Streamable Stage**

This stage supports streaming operations

---

7.4. Filters
Example

```json
[
    {
        "type": "readers.las",
        "filename": "larger.las"
    },
    {
        "type": "filters.decimation",
        "step": 10
    },
    {
        "type": "writers.las",
        "filename": "smaller.las"
    }
]
```

See also:

filters.voxelgrid provides grid-style point decimation.

**Options**

**step**  Number of points to skip between each sample point. A step of 1 will skip no points. A step of 2 will skip every other point. A step of 100 will reduce the input by ~99%. [Default: 1]

**offset**  Point index to start sampling. Point indexes start at 0. [Default: 0]

**limit**  Point index at which sampling should stop (exclusive). [Default: No limit]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: `auto`]
filters.farthestpointsampling

The **Farthest Point Sampling Filter** adds points from the input to the output `PointView` one at a time by selecting the point from the input cloud that is farthest from any point currently in the output.

**See also:**

`filters.sample` (page 233) produces a similar result, but while `filters.sample` allows us to target a desired separation of points via the `radius` parameter at the expense of knowing the number of points in the output, `filters.farthestpointsampling` allows us to specify exactly the number of output points at the expense of knowing beforehand the spacing between points.

**Default Embedded Stage**

This stage is enabled by default

**Options**

- **count** Desired number of output samples. [Default: 1000]
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: `auto`]

**filters.sample**

The **Sample Filter** performs Poisson sampling of the input `PointView`. The practice of performing Poisson sampling via “Dart Throwing” was introduced in the mid-1980’s by [Cook1986] and [Dippe1985], and has been applied to point clouds in other software [Mesh2009].

The sampling can be performed in a single pass through the point cloud. To begin, each input point is assumed to be kept. As we iterate through the kept points, we retrieve all neighbors within a given `radius`, and mark these neighbors as points to be discarded. All remaining kept points are appended to the output `PointView`. The full layout (i.e., the dimensions) of the input `PointView` is kept in tact (the same cannot be said for `filters.voxelgrid`).

7.4. Filters
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

See also:

*filters.decimation* (page 231) and *filters.voxelgrid* also perform decimation.

**Default Embedded Stage**

This stage is enabled by default

**Options**

**radius**  Minimum distance between samples. [Default: 1.0]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

*filters.decimation* (page 231)  Keep every Nth point.

*filters.farthestpointsampling* (page 233)  The Farthest Point Sampling Filter adds points from the input to the output PointView one at a time by selecting the point from the input cloud that is farthest from any point currently in the output.

*filters.sample* (page 233)  Perform Poisson sampling and return only a subset of the input points.

**Conditional**

*filters.dem*

The **DEM filter** uses a source raster to keep point cloud data within a each cell within a computed range. For example, atmospheric or MTA noise in a scene can be quickly removed by keeping all data within 100m above and 20m below a preexisting elevation model.

**Default Embedded Stage**

This stage is enabled by default
Example

```json
[
  {
    "type": "filters.dem",
    "raster": "dem.tif",
    "limits": "Z[20:100]"
  }
]
```

Options

limits  A range (page 249) that defines the dimension and the magnitude above and below the value of the given dimension to filter.

For example “Z[20:100]” would keep all Z point cloud values that are within 100 units above and 20 units below the elevation model value at the given X and Y value.

raster  GDAL readable raster (http://www.gdal.org/formats_list.html) data to use for filtering.

band  GDAL Band number to read (count from 1) [Default: 1]

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.iqr

The Interquartile Range Filter automatically crops the input point cloud based on the distribution of points in the specified dimension. The Interquartile Range (IQR) is defined as the range between the first and third quartile (25th and 75th percentile). Upper and lower bounds are determined by adding 1.5 times the IQR to the third quartile or subtracting 1.5 times the IQR from the first quartile. The multiplier, which defaults to 1.5, can be adjusted by the user.

Note: This method can remove real data, especially ridges and valleys in rugged terrain, or tall features such as towers and rooftops in flat terrain. While the number of deviations can be
adjusted to account for such content-specific considerations, it must be used with care.

**Default Embedded Stage**

This stage is enabled by default

**Example**

The sample pipeline below uses the filter to automatically crop the Z dimension and remove possible outliers. The multiplier to determine high/low thresholds has been adjusted to be less aggressive and to only crop those outliers that are greater than the third quartile plus 3 times the IQR or are less than the first quartile minus 3 times the IQR.

```
[  
    "input.las",
    {
      "type": "filters.iqr",
      "dimension": "Z",
      "k": 3.0  
    },
    "output.laz"
]
```

**Options**

- **k** The IQR multiplier used to determine upper/lower bounds. [Default: 1.5]
- **dimension** The name of the dimension to filter.
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If *true*, the skipped points are added to the first point view returned by the skipped filter. If *false*, skipped points are placed in their own point view. If *auto*, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
**filters.mad**

The **MAD filter** filter crops the input point cloud based on the distribution of points in the specified *dimension* (page 237). Specifically, we choose the method of median absolute deviation from the median (commonly referred to as MAD), which is robust to outliers (as opposed to mean and standard deviation).

**Note:** This method can remove real data, especially ridges and valleys in rugged terrain, or tall features such as towers and rooftops in flat terrain. While the number of deviations can be adjusted to account for such content-specific considerations, it must be used with care.

### Default Embedded Stage

This stage is enabled by default.

### Example

The sample pipeline below uses filters.mad to automatically crop the **Z** dimension and remove possible outliers. The number of deviations from the median has been adjusted to be less aggressive and to only crop those outliers that are greater than four deviations from the median.

```json
[  
  "input.las",
  {
    "type":"filters.mad",
    "dimension":"Z",
    "k":4.0
  },
  "output.laz"
]
```

### Options

**k** The number of deviations from the median. [Default: 2.0]

**dimension** The name of the dimension to filter.

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘**where**’ option when running in standard mode. If **true**, the skipped points are added to the first point view returned by...
the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.dem** *(page 234)* Remove points that are in a raster cell but have a value far from the value of the raster.

**filters.iqr** *(page 235)* Cull points falling outside the computed Interquartile Range for a given dimension.

**filters.mad** *(page 237)* Cull points falling outside the computed Median Absolute Deviation for a given dimension.

**Voxel**

**filters.voxelcenternearestneighbor**

The **VoxelCenterNearestNeighbor filter** is a voxel-based sampling filter. The input point cloud is divided into 3D voxels at the given cell size. For each populated voxel, the coordinates of the voxel center are used as the query point in a 3D nearest neighbor search. The nearest neighbor is then added to the output point cloud, along with any existing dimensions.

**Note:** This is similar to the existing filters.voxelgrid. However, in the case of the VoxelGrid, the centroid of the points falling within the voxel is added to the output point cloud. The drawback with this approach is that all dimensional data is lost, and the sampled cloud now consists of only XYZ coordinates.

**Default Embedded Stage**

This stage is enabled by default

**Example**

```json
[
    "input.las",
    {
        "type": "filters.voxelcenternearestneighbor",
        "cell": 10.0
    }
]
```

(continues on next page)
"output.las"
[
]

See also:

`filters.voxelcentroidnearestneighbor` (page 239) offers a similar solution, using as the query point the centroid of all points falling within the voxel as opposed to the voxel center coordinates.

**Options**

cell  Cell size in the $X$, $Y$, and $Z$ dimension. [Default: 1.0]

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.voxelcentroidnearestneighbor**

The **VoxelCentroidNearestNeighbor Filter** is a voxel-based sampling filter. The input point cloud is divided into 3D voxels at the given cell size. For each populated voxel, we apply the following ruleset. For voxels with only one point, the point is passed through to the output. For voxels with exactly two points, the point closest the voxel center is returned. Finally, for voxels with more than two points, the centroid of the points within that voxel is computed. This centroid is used as the query point in a 3D nearest neighbor search (considering only those points lying within the voxel). The nearest neighbor is then added to the output point cloud, along with any existing dimensions.

**Default Embedded Stage**

This stage is enabled by default
Example

```
[  "input.las",
   {
      "type": "filters.voxelcentroidnearestneighbor",
      "cell": 10.0
   },
   "output.las"
]
```

See also:

`filters.voxelcenternearestneighbor` (page 238) offers a similar solution, using the voxel center as opposed to the voxel centroid for the query point.

Options

- **cell**  Cell size in the X, Y, and Z dimension. [Default: 1.0]
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
**filters.voxeldownsize**

The **voxeldownsize filter** is a voxel-based sampling filter. The input point cloud is divided into 3D voxels at the given cell size. For each populated voxel, either first point entering in the voxel or center of a voxel (depending on mode argument) is accepted and voxel is marked as populated. All other points entering in the same voxel are filtered out.

**Example**

```json
[  
    "input.las",
    {
      "type": "filters.voxeldownsize",
      "cell": 1.0,
      "mode": "center"
    },
    "output.las"
]
```

See also:

*filters.voxelcenternearestneighbor* (page 238) offers a similar solution, using the coordinates of the voxel center as the query point in a 3D nearest neighbor search. The nearest neighbor is then added to the output point cloud, along with any existing dimensions.

**Options**

- **cell**  Cell size in the X, Y, and Z dimension. [Default: 0.001]
- **mode**  Mode for voxel based filtering. [Default: center] center: Coordinates of the first point found in each voxel will be modified to be the center of the voxel. first: Only the first point found in each voxel is retained.
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
Warning: If you choose center mode, you are overwriting the X, Y and Z values of retained points. This may invalidate other dimensions of the point if they depend on this location or the location of other points in the input.

`filters.voxelcenternearestneighbor` (page 238) Return the point within each voxel that is nearest the voxel center.

`filters.voxelcentroidnearestneighbor` (page 239) Return the point within each voxel that is nearest the voxel centroid.

`filters.voxeldownsize` (page 241) Retain either first point detected in each voxel or center of a populated voxel, depending on mode argument.

### Position

`filters.head`

The **Head filter** returns a specified number of points from the beginning of a `PointView`.

**Note:** If the requested number of points exceeds the size of the point cloud, all points are passed with a warning.

### Default Embedded Stage

This stage is enabled by default

### Example #1

Thin a point cloud by first shuffling the point order with `filters.randomize` (page 212) and then picking the first 10000 using the HeadFilter.

```json
[
  {
    "type": "filters.randomize"
  },
  {
    "type": "filters.head",
    "count": 10000
  }
]
```
Example #2

Compute height above ground and extract the ten highest points.

```
[
  {
    "type":"filters.smrf"
  },
  {
    "type":"filters.hag_nn"
  },
  {
    "type":"filters.sort",
    "dimension":"HeightAboveGround",
    "order":"DESC"
  },
  {
    "type":"filters.head",
    "count":10
  }
]
```

See also:

`filters.tail` (page 250) is the dual to `filters.head` (page 242).

Options

`count` Number of points to return. [Default: 10]

`where` An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

`where_merge` A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: `auto`]
filters.locate

The Locate filter searches the specified dimension (page 244) for the minimum or maximum value and returns a single point at this location. If multiple points share the min/max value, the first will be returned. All dimensions of the input PointView will be output, subject to any overriding writer options.

Default Embedded Stage

This stage is enabled by default

Example

This example returns the point at the highest elevation.

```json
[
    "input.las",
    {
        "type": "filters.locate",
        "dimension": "Z",
        "minmax": "max"
    },
    "output.las"
]
```

Options

dimension Name of the dimension in which to search for min/max value.

minmax Whether to return the minimum or maximum value in the dimension.

where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.mongo

The **Mongo Filter** applies query logic to the input point cloud based on a MongoDB-style query expression using the point cloud attributes.

**Default Embedded Stage**

This stage is enabled by default.

**Streamable Stage**

This stage supports streaming operations.

**Example**

This example passes through only the points whose Classification is non-zero.

```
[  
  "input.las",
  {
    "type": "filters.mongo",
    "expression": {
      "Classification": { "$ne": 0 }
    }
  },
  "filtered.las"
]
```

This example passes through only the points whose ReturnNumber is equal to the NumberOfReturns and the NumberOfReturns is greater than 1.

```
[  
  "input.las",
  {
    "type": "filters.mongo",
    "expression": { "$and": [
      { "ReturnNumber": "NumberOfReturns" },
      { "NumberOfReturns": { "$gt": 1 } }
    ] }
  },
  "filtered.las"
]
```
Options

**expression** A JSON query expression (page 246) containing a combination of query comparisons and logical operators.

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: `auto`]

Expression

A query expression is a combination of comparison and logical operators that define a query which can be used to select matching points by their attribute values.

Comparison operators

There are 8 valid query comparison operators:

- `$eq`: Matches values equal to a specified value.
- `$gt`: Matches values greater than a specified value.
- `$gte`: Matches values greater than or equal to a specified value.
- `$lt`: Matches values less than a specified value.
- `$lte`: Matches values less than or equal to a specified value.
- `$ne`: Matches values not equal to a specified value.
- `$in`: Matches any of the values specified in the array.
- `$nin`: Matches none of the values specified in the array.

Comparison operators compare a point cloud attribute with an operand or an array of operands. An *operand* is either a numeric constant or a string representing a dimension name. For all comparison operators except for `$in` and `$nin`, the comparison value must be a single operand. For `$in` and `$nin`, the value must be an array of operands.

Comparison operator specifications must be contained within an object whose key is the dimension name to be compared.
The `$eq` comparison operator may be implicitly invoked by setting an attribute name directly to a value.

Logical operators

There are 4 valid logical operators:

- `$and`: Applies a logical and on the expressions of the array and returns a match only if all expressions match.
- `$not`: Inverts the value of the single sub-expression.
- `$nor`: Applies a logical nor on the expressions of the array and returns a match only if all expressions fail to match.
- `$or`: Applies a logical or on the expressions of the array and returns a match if any of the expressions match.

Logical operators are used to logically combine sub-expressions. All logical operators except for `$not` are applied to arrays of expressions. `$not` is applied to a single expression and negates its result.

Logical operators may be applied directly to comparison expressions or may contain further nested logical operators. For example:

```json
{ "$or": [ 
    { "Classification": 2 },
    { "Intensity": { "$gt": 0 } }
] }
```

```json
{ "$or": [ 
    { "Classification": 2 },
    { "$and": [ 
        { "ReturnNumber": "NumberOfReturns" },
        { "NumberOfReturns": { "$gt": 1 } }
    ] }
] }
```
For any individual dimension, the logical **and** may be implicitly invoked via multiple comparisons within the comparison object. For example:

```json
{ "$or": [
    { "Classification": 2 },
    { "$and": [
        { "ReturnNumber": { "$gt": 0 } },
        { "Z": { "$lte": 42 } }
    ] }
]
}
```

**filters.range**

The **Range Filter** applies rudimentary filtering to the input point cloud based on a set of criteria on the given dimensions.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Streamable Stage**

This stage supports streaming operations

---

**Example**

This example passes through all points whose \( Z \) value is in the range \([0,100]\) and whose \( Classification \) equals 2 (corresponding to ground in LAS).

```json
[
    "input.las",
    {
        "type":"filters.range",
        "limits":"Z[0:100],Classification[2:2]"
    },
]
```

(continues on next page)
The equivalent pipeline invoked via the PDAL translate command would be

```bash
$ pdal translate -i input.las -o filtered.las -f range --filters.
  \ overrides=range.limits="Z[0:100],Classification[2:2]"
```

### Options

**limits**  A comma-separated list of **Ranges** (page 249). If more than one range is specified for a dimension, the criteria are treated as being logically ORed together. Ranges for different dimensions are treated as being logically ANDed.

Example:

```plaintext
Classification[1:2], Red[1:50], Blue[25:75], Red[75:255],
  Classification[6:7]
```

This specification will select points that have the classification of 1, 2, 6 or 7 and have a blue value or 25-75 and have a red value of 1-50 or 75-255. In this case, all values are inclusive.

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

### Ranges

A range specification is a dimension name, followed by an optional negation character (‘!’), and a starting and ending value separated by a colon, surrounded by parentheses or square brackets. Either the starting or ending values can be omitted. Parentheses indicate an open endpoint that doesn’t include the adjacent value. Square brackets indicate a closed endpoint that includes the adjacent value.
Example 1:

\texttt{Z[10:]} 

Selects all points with a Z value greater than or equal to 10.

Example 2:

\texttt{Classification[2:2]} 

Selects all points with a classification of 2.

Example 3:

\texttt{Red!(20:40]} 

Selects all points with red values less than or equal to 20 and those with values greater than 40.

Example 4:

\texttt{Blue[:255)} 

Selects all points with a blue value less than 255.

Example 5:

\texttt{Intensity![25:25]} 

Selects all points with an intensity not equal to 25.

\texttt{filters.tail}

The \texttt{Tail Filter} returns a specified number of points from the end of the \texttt{PointView}.

\textbf{Note:} If the requested number of points exceeds the size of the point cloud, all points are passed with a warning.
**Default Embedded Stage**

This stage is enabled by default

---

**Example**

Sort and extract the 100 lowest intensity points.

```json
[
  {
    "type": "filters.sort",
    "dimension": "Intensity",
    "order": "DESC"
  },
  {
    "type": "filters.tail",
    "count": 100
  }
]
```

See also:

*filters.head* (page 242) is the dual to *filters.tail* (page 250).

**Options**

- **count**  Number of points to return. [Default: 10]
- **where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

*filters.head* (page 242)  Return N points from beginning of the point cloud.

*filters.locate* (page 244)  Return a single point with min/max value in the named dimension.

*filters.mongo* (page 245)  Cull points using MongoDB-style expression syntax.

*filters.range* (page 248)  Pass only points given a dimension/range.

---

7.4. Filters
filters.tail (page 250) Return N points from end of the point cloud.

7.4.5 New

PDAL filters can be used to split the incoming point cloud into subsets. These filters will invalidate an existing KD-tree.

Spatial

filters.chipper

The Chipper Filter takes a single large point cloud and converts it into a set of smaller clouds, or chips. The chips are all spatially contiguous and non-overlapping, so the result is an irregular tiling of the input data.

Note: Each chip will have approximately, but not exactly, the capacity (page 255) point count specified.

See also:

The PDAL split command (page 35) utilizes the filters.chipper (page 252) to split data by capacity.

Chipping is usually applied to data read from files (which produce one large stream of points) before the points are written to a database (which prefer data segmented into smaller blocks).

Default Embedded Stage

This stage is enabled by default

Example

```
[
   "example.las",
   {
      "type":"filters.chipper",
      "capacity":"400"
   },
   {
      "type":"writers.pgpointcloud",
```

(continues on next page)
Fig. 6: Before chipping, the points are all in one collection.
Fig. 7: After chipping, the points are tiled into smaller contiguous chips.
Options

capacity  How many points to fit into each chip. The number of points in each chip will not exceed this value, and will sometimes be less than it. [Default: 5000]

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.divider

The Divider Filter breaks a point view into a set of smaller point views based on simple criteria. The number of subsets can be specified explicitly, or one can specify a maximum point count for each subset. Additionally, points can be placed into each subset sequentially (as they appear in the input) or in round-robin fashion.

Normally points are divided into subsets to facilitate output by writers that support creating multiple output files with a template (LAS and BPF are notable examples).

Default Embedded Stage

This stage is enabled by default

Example

This pipeline will create 10 output files from the input file readers.las.

```json
[
    "example.las",
    
]  
```
"type": "filters.divider",
"count": "10"
},
{
"type": "writers.las",
"filename": "out_.las"
}
]

Options

**mode** A mode of “partition” will write sequential points to an output view until the view meets its predetermined size. “round_robin” mode will iterate through the output views as it writes sequential points. [Default: “partition”]

**count** Number of output views. [Default: none]

**capacity** Maximum number of points in each output view. Views will contain approximately equal numbers of points. [Default: none]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**Warning:** You must specify exactly one of either **count** (page 256) or **capacity** (page 256).

**filters.splitter**

The **Splitter Filter** breaks a point cloud into square tiles of a specified size. The origin of the tiles is chosen arbitrarily unless specified with the **origin_x** (page 257) and **origin_y** (page 257) option.

The splitter takes a single **PointView** as its input and creates a **PointView** for each tile as its output.

Splitting is usually applied to data read from files (which produce one large stream of points) before the points are written to a database (which prefer data segmented into smaller blocks).
Default Embedded Stage

This stage is enabled by default

Example

```
[  
  "input.las",  
  {  
    "type":"filters.splitter",  
    "length":"100",  
    "origin_x":"638900.0",  
    "origin_y":"835500.0"  
  },  
  {  
    "type":"writers.pgpointcloud",  
    "connection":"dbname='lidar' user='user'"  
  }  
]
```

Options

**length**  Length of the sides of the tiles that are created to hold points. [Default: 1000]

**origin_x**  X Origin of the tiles. [Default: none (chosen arbitrarily)]

**origin_y**  Y Origin of the tiles. [Default: none (chosen arbitrarily)]

**buffer**  Amount of overlap to include in each tile. This buffer is added onto length in both the x and the y direction. [Default: 0]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.chipper**  Organize points into spatially contiguous, squarish, and non-overlapping chips.
filters.divider (page 255) Divide points into approximately equal sized groups based on a simple scheme.

filters.splitter (page 256) Split data based on a X/Y box length.

Dimension

filters.groupby

The Groupby Filter takes a single PointView as its input and creates a PointView for each category in the named *dimension* (page 258) as its output.

Default Embedded Stage

This stage is enabled by default

Example

The following pipeline will create a set of LAS files, where each file contains only points of a single Classification.

```json
[  
    "input.las",
    
    
    "type":"filters.groupby",
    "dimension":"Classification"
  ],
  "output_#.las"
]
```

Options

dimension The dimension containing data to be grouped.

where An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
filters.returns

The **Returns Filter** takes a single PointView as its input and creates a PointView for each of the user-specified groups (page 259) defined below.

“first” is defined as those points whose `ReturnNumber` is 1 when the `NumberOfReturns` is greater than 1.

“intermediate” is defined as those points whose `ReturnNumber` is greater than 1 and less than `NumberOfReturns` when `NumberOfReturns` is greater than 2.

“last” is defined as those points whose `ReturnNumber` is equal to `NumberOfReturns` when `NumberOfReturns` is greater than 1.

“only” is defined as those points whose `NumberOfReturns` is 1.

---

**Default Embedded Stage**

This stage is enabled by default

---

**Example**

This example creates two separate output files for the “last” and “only” returns.

```json
[
  "input.las",
  {
    "type": "filters.returns",
    "groups": "last,only"
  },
  "output_#.las"
]
```

**Options**

- **groups** Comma-separated list of return number groupings. Valid options are “first”, “last”, “intermediate” or “only”. [Default: “last”]
- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]
- **where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view
is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.separatescanline**

The **Separate scan line Filter** takes a single PointView as its input and creates a PointView for each scan line as its output. PointView must contain the EdgeOfFlightLine dimension.

### Default Embedded Stage

This stage is enabled by default

### Example

The following pipeline will create a set of text files, where each file contains only 10 scan lines.

```json
[
    "input.text",
    {
        "type": "filters.separatescanline",
        "groupby": 10
    },
    "output_#.text"
]
```

### Options

**groupby**  The number of lines to be grouped by. [Default : 1]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.groupby** (page 258) Split data categorically by dimension.

**filters.returns** (page 259) Split data by return order (e.g., ‘first’, ‘last’, ‘intermediate’, ‘only’).
7.4.6 Join

Multiple point clouds can be joined to form a single point cloud. These filters will invalidate an existing KD-tree.

filters.merge

The Merge Filter combines input from multiple sources into a single output. In most cases, this happens automatically on output and use of the merge filter is unnecessary. However, there may be special cases where merging points prior to a particular filter or writer is necessary or desirable.

The merge filter will log a warning if its input point sets are based on different spatial references. No checks are made to ensure that points from various sources being merged have similar dimensions or are generally compatible.

Default Embedded Stage

This stage is enabled by default

Example 1

This pipeline will create an output file “output.las” that concatenates the points from “file1”, “file2” and “file3”. Note that the explicit use of the merge filter is unnecessary in this case (removing the merge filter will yield the same result).

```json
[
    "file1",
    "file2",
    "file3",
    {
        "type": "filters.merge"
    },
    "output.las"
]
```
Example 2

Here are a pair of unlikely pipelines that show one way in which a merge filter might be used. The first pipeline simply reads the input files “utm1.las”, “utm2.las” and “utm3.las”. Since the points from each input set are carried separately through the pipeline, three files are created as output, “out1.las”, “out2.las” and “out3.las”. “out1.las” contains the points in “utm1.las”. “out2.las” contains the points in “utm2.las” and “out3.las” contains the points in “utm3.las”.

```
[  
  "utm1.las",
  "utm2.las",
  "utm3.las",
  "out#.las"
]
```

Here is the same pipeline with a merge filter added. The merge filter will combine the points in its input: “utm1.las” and “utm2.las”. Then the result of the merge filter is passed to the writer along with “utm3.las”. This results in two output files: “out1.las” contains the points from “utm1.las” and “utm2.las”, while “out2.las” contains the points from “utm3.las”.

```
[  
  "utm1.las",
  "utm2.las",
  {  
    "type" : "filters.merge"
  },
  "utm3.las",
  "out#.las"
]
```

Options

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.merge** (page 261) Merge data from two different readers into a single stream.
7.4.7 Metadata

PDAL filters can be used to create new metadata. These filters will not invalidate an existing KD-tree.

Note: filters.cpd (page 215) and filters.icp (page 217) can optionally create metadata as well, inserting the computed transformation matrix.

filters.hexbin

A common questions for users of point clouds is what the spatial extent of a point cloud collection is. Files generally provide only rectangular bounds, but often the points inside the files only fill up a small percentage of the area within the bounds.

Fig. 8: Hexbin output shows boundary of actual points in point buffer, not just rectangular extents.

The hexbin filter reads a point stream and writes out a metadata record that contains a boundary, expressed as a well-known text polygon. The filter counts the points in each hexagonal area to determine if that area should be included as part of the boundary. In order to write out the metadata record, the pdal pipeline command must be invoked using the “–pipeline-serialization” option:

Streamable Stage

This stage supports streaming operations
Example 1

The following pipeline file and command produces an JSON output file containing the pipeline’s metadata, which includes the result of running the hexbin filter:

```
[  
  "/Users/me/pdal/test/data/las/autzen_trim.las",
  {  
    "type" : "filters.hexbin"
  }
]
```

```
$ pdal pipeline hexbin-pipeline.json --metadata hexbin-out.json
```

```
{  
  "stages":  
  {  
    "filters.hexbin":  
    {  
      "area": 746772.7543,  
      "avg_pt_per_sq_unit": 22.43269935,  
      "avg_pt_spacing": 2.605540869,  
      "boundary": "MULTIPOLYGON (((636274.38924399 848834.99817891,  
      →637242.52219686 848834.99817891, 637274.79329529 849226.26445367,  
      →637145.70890157 849338.05481789, 637242.52219686 849505.74036422,  
      →636016.22045656 849505.74036422, 635983.94935813 849114.47408945,  
      →636113.03375184 848890.89336102, 636274.38924399 848834.99817891)))
      →",  
      "boundary_json": { "type": "MultiPolygon", "coordinates": [ [  
      →[ 636274.38924399, 848834.99817891 ], [ 637242.52219686, 848834.  
      →99817891 ], [ 637274.79329529, 849226.26445367 ], [ 637145.  
      →70890157, 849338.05481789 ], [ 637242.52219686, 849505.74036422 ],  
      →[ 636016.22045656, 849505.74036422 ], [ 635983.94935813, 849114.  
      →47408945 ], [ 636113.03375184, 848890.89336102 ], [ 636274.  
      →38924399, 848834.99817891 ] ] },  
      "density": 0.1473004999,  
      "edge_length": 0,  
      "estimated_edge": 111.7903642,  
      "hex_offsets": "MULTIPOINT (0 0, -32.2711 55.8952, 0 111.79,  
      →64.5422 111.79, 96.8133 55.8952, 64.5422 0)",  
      "sample_size": 5000,  
      "threshold": 15
    }
  }
}
```
Example 2

As a convenience, the `pdal info` command will produce similar output:

```bash
$ pdal info --boundary /Users/me/test/data/las/autzen_trim.las
```

```
{}
  "boundary": {
    "area": 746772.7543,
    "avg_pt_per_sq_unit": 22.43269935,
    "avg_pt_spacing": 2.605540869,
    "boundary": "MULTIPOLYGON (((636274.38924399 848834.99817891, ...
    "boundary_json": { "type": "MultiPolygon", "coordinates": [ [ [ ...
    "density": 0.1473004999,
    "edge_length": 0,
    "estimated_edge": 111.7903642,
    "hex_offsets": "MULTIPOINT (0 0, -32.2711 55.8952, 0 111.79, 64.5422 111.79, 96.8133 55.8952, 64.5422 0)"
    "sample_size": 5000,
    "threshold": 15
  },
  "filename": "\Users/\acbell/\pdal/\test/\data/\las/\autzen_trim.las",
  "pdal_version": "1.6.0 (git-version: 675afe)"
}
```
Options

**edge_size** If not set, the hexbin filter will estimate a hex size based on a sample of the data. If set, hexbin will use the provided size in constructing the hexbins to test.

**sample_size** How many points to sample when automatically calculating the edge size? Only applies if `edge_size` (page 266) is not explicitly set. [Default: 5000]

**threshold** Number of points that have to fall within a hexagon boundary before it is considered “in” the data set. [Default: 15]

**precision** Minimum number of significant digits to use in writing out the well-known text of the boundary polygon. [Default: 8]

**preserve_topology** Use GEOS SimplifyPreserveTopology instead of Simplify for polygon simplification with `smooth` option. [Default: true]

**smooth** Use GEOS simplify operations to smooth boundary to a tolerance [Default: true]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.info**

The **Info filter** provides simple information on a point set as metadata. It is usually invoked by the info command, rather than by user code. The data provided includes bounds, a count of points, dimension names, spatial reference, and points meeting a query criteria.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

```python
[
    "input.las",
]
```

(continues on next page)
Options

**point** A comma-separated list of single point IDs or ranges of points. For example “2-6, 10, 25” selects eight points from the input set. The first point has an ID of 0. The `point` (page 267) option can’t be used with the `query` (page 267) option. [Default: no points are selected.]

**query** A specification to retrieve points near a location. Syntax of the query is X,Y[Z]/[count] where ‘X’, ‘Y’ and ‘Z’ are coordinate locations mapping to the X, Y and Z point dimension and ‘count’ is the number of points to return. If ‘count’ isn’t specified, the 10 points nearest to the location are returned. The `query` (page 267) option can’t be used with the `point` (page 267) option. [Default: no points are selected.]

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: `auto`]

**filters.stats**

The Stats Filter calculates the minimum, maximum and average (mean) values of dimensions. On request it will also provide an enumeration of values of a dimension and skewness and kurtosis.

The output of the stats filter is metadata that can be stored by writers or used through the PDAL API. Output from the stats filter can also be quickly obtained in JSON format by using the command “pdal info –stats”.

**Note:** The filter can compute both sample and population statistics. For kurtosis, the filter can also compute standard and excess kurtosis. However, only a single value is reported for each statistic type in metadata, and that is the sample statistic, rather than the population statistic.
For kurtosis the sample excess kurtosis is reported. This seems to match the behavior of many other software packages.

**Example**

```json
[
    "input.las",
    {
        "type": "filters.stats",
        "dimensions": "X,Y,Z,Classification",
        "enumerate": "Classification"
    },
    {
        "type": "writers.las",
        "filename": "output.las"
    }
]
```

**Options**

- **dimensions** A comma-separated list of dimensions whose statistics should be processed. If not provided, statistics for all dimensions are calculated.

- **enumerate** A comma-separated list of dimensions whose values should be enumerated. Note that this list does not add to the list of dimensions that may be provided in the `dimensions` option.

- **count** Identical to the `enumerate` option, but provides a count of the number of points in each enumerated category.

- **global** A comma-separated list of dimensions for which global statistics (median, mad, mode) should be calculated.

- **advanced** Calculate advanced statistics (skewness, kurtosis). [Default: false]

- **where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

- **where_merge** A strategy for merging points skipped by a `where` option when running in standard mode. If `true`, the skipped points are added to the first point view returned by the skipped filter. If `false`, skipped points are placed in their own point view. If `auto`, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]
**filters.hexbin** *(page 263)*  
Tessellate XY domain and determine point density and/or point boundary.

**filters.info** *(page 266)*  
Generate metadata about the point set, including a point count and spatial reference information.

**filters.stats** *(page 267)*  
Compute statistics about each dimension (mean, min, max, etc.).

### 7.4.8 Mesh

Meshes can be computed from point clouds. These filters will invalidate an existing KD-tree.

**filters.delaunay**

The Delaunay Filter creates a triangulated mesh fulfilling the Delaunay condition from a collection of points.


The filter currently only supports 2D Delaunay triangulation, using the $x$ and $y$ dimensions of the point cloud.

---

**Default Embedded Stage**

This stage is enabled by default.

---

**Example**

```json
[

    "input.las",

    {
        "type": "filters.delaunay"
    },

    {
        "type": "writers.ply",
        "filename": "output.ply",
        "faces": true
    }

]
```

---

**7.4. Filters**
Options

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.greedyprojection**

The **Greedy Projection Filter** creates a mesh (triangulation) in an attempt to reconstruct the surface of an area from a collection of points.

GreedyProjectionTriangulation is an implementation of a greedy triangulation algorithm for 3D points based on local 2D projections. It assumes locally smooth surfaces and relatively smooth transitions between areas with different point densities. The algorithm itself is identical to that used in the PCL (http://www.pointclouds.org/documentation/tutorials/greedy_projection.php) library.

**Default Embedded Stage**

This stage is enabled by default

**Example**

```json
[
  "input.las",
  {
    "type": "filters.greedyprojection",
    "multiplier": 2,
    "radius": 10
  },
  {
    "type": "writers.ply",
    "faces": true,
    "filename": "output.ply"
  }
]
```
Options

**multiplier**  Nearest neighbor distance multiplier. [Required]

**radius**  Search radius for neighbors. [Required]

**num_neighbors**  Number of nearest neighbors to consider. [Required]

**min_angle**  Minimum angle for created triangles. [Default: 10 degrees]

**max_angle**  Maximum angle for created triangles. [Default: 120 degrees]

**eps_angle**  Maximum normal difference angle for triangulation consideration. [Default: 45 degrees]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.poisson**

The Poisson Filter passes data Mischa Kazhdan’s poisson surface reconstruction algorithm. [Kazhdan2006] It creates a watertight surface from the original point set by creating an entirely new point set representing the imputed isosurface. The algorithm requires normal vectors to each point in order to run. If the x, y and z normal dimensions are present in the input point set, they will be used by the algorithm. If they don’t exist, the poisson filter will invoke the PDAL normal filter to create them before running.

The poisson algorithm will usually create a larger output point set than the input point set. Because the algorithm constructs new points, data associated with the original points set will be lost, as the algorithm has limited ability to impute associated data. However, if color dimensions (red, green and blue) are present in the input, colors will be reconstructed in the output point set.

This integration of the algorithm with PDAL only supports a limited set of the options available to the implementation. If you need support for further options, please let us know.

**Default Embedded Stage**

This stage is enabled by default

---

7.4. Filters 271
Example

```json
[  "dense.las",
   {
      "type": "filters.poisson"
   },
   {
      "type": "writers.ply",
      "filename": "isosurface.ply"
   }
]
```

Note: The algorithm is slow. On a reasonable desktop machine, the surface reconstruction shown below took about 15 minutes.

Options

density  Write an estimate of neighborhood density for each point in the output set.

depth  Maximum depth of the tree used for reconstruction. The output is sensitive to this parameter. Increase if the results appear unsatisfactory. [Default: 8]

where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.delaunay (page 269)  Create mesh using Delaunay triangulation.

filters.greedyprojection (page 270)  Create mesh using the Greedy Projection Triangulation approach.

filters.poisson (page 271)  Create mesh using the Poisson surface reconstruction algorithm [Kazhdan2006].
Fig. 9: Point cloud (800,000 points)
Fig. 10: Reconstruction (1.8 million vertices, 3.7 million faces)
7.4.9 Languages

PDAL has three filters than can be used to pass point clouds to other languages. These filters will invalidate an existing KD-tree.

**filters.matlab**

The **Matlab Filter** allows Matlab (https://www.mathworks.com/products/matlab.html) software to be embedded in a **Pipeline** (page 45) that interacts with a struct array of the data and allows you to modify those points. Additionally, some global **Metadata** (page 467) is also available that Matlab functions can interact with.

The Matlab interpreter must exit and always set “ans==true” upon success. If “ans==false”, an error would be thrown and the **Pipeline** (page 45) exited.

**See also:**

**writers.matlab** (page 129) can be used to write .mat files.

---

**Note:** **filters.matlab** (page 275) embeds the entire Matlab interpreter, and it will require a fully licensed version of Matlab to execute your script.

---

**Dynamic Plugin**

This stage requires a dynamic plugin to operate

---

**Example**

```json
[
    {
        "filename": "test\data\las\1.2-with-color.las",
        "type": "readers.las"
    },
    {
        "type": "filters.matlab",
        "script": "matlab.m"
    },
    {
        "filename": "out.las",
        "type": "writers.las"
    }
]
```

(continues on next page)
Options

**script**  When reading a function from a separate Matlab (https://www.mathworks.com/products/matlab.html) file, the file name to read from. [Example: “functions.m”]

**source**  The literal Matlab (https://www.mathworks.com/products/matlab.html) code to execute, when the script option is not being used.

**add_dimension**  The name of a dimension to add to the pipeline that does not already exist.

**struct**  Array structure name to read [Default: “PDAL”]

**where**  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge**  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

**filters.python**

The **Python Filter** allows Python (http://python.org/) software to be embedded in a **Pipeline** (page 45) that allows modification of PDAL points through a **NumPy** (http://www.numpy.org/) array. Additionally, some global **Metadata** (page 467) is also available that Python functions can interact with.

The function must have two **NumPy** (http://www.numpy.org/) arrays as arguments, ins and outs. The ins array represents the points before the filters.python filter and the outs array represents the points after filtering.

**Warning:** Make sure **NumPy** (http://www.numpy.org/) is installed in your Python (http://python.org/) environment.

```
$ python3 -c "import numpy; print(numpy.__version__)"
1.18.1
```
**Warning:** Each array contains all the *Dimensions* (page 287) of the incoming `ins` point schema. Each array in the `outs` list matches the NumPy (http://www.numpy.org/) array of the same type as provided as `ins` for shape and type.

---

**Dynamic Plugin**

This stage requires a dynamic plugin to operate.

```python
import numpy as np

def multiply_z(ins, outs):
    Z = ins['Z']
    Z = Z * 10.0
    outs['Z'] = Z
    return True
```

1) The function must always return *True* upon success. If the function returned *False*, an error would be thrown and the *Pipeline* (page 45) exited.

2) If you want write a dimension that might not be available, you can specify it with the *add_dimension* (page 281) option:

```
"add_dimension": "NewDimensionOne"
```

To create more than one dimension, this option also accepts an array:

```
"add_dimension": [ "NewDimensionOne", "NewDimensionTwo", "NewDimensionThree" ]
```

You can also specify the *type* (page 293) of the dimension using an `=`.

```
"add_dimension": "NewDimensionOne=uint8"
```

---

**Modification Example**

```
[
    "file-input.las",
    {
        "type":"filters.smrf"
    },
    {
        "type":"filters.python",
    }
]
```

(continues on next page)
The JSON pipeline file referenced the external `multiply_z.py` Python script, which scales the $Z$ coordinate by a factor of 10.

```python
import numpy as np

def multiply_z(ins, outs):
    Z = ins['Z']
    Z = Z * 10.0
    outs['Z'] = Z
    return True
```

**Predicates**

Points can be retained/removed from the stream by setting true/false values into a special “Mask” dimension in the output point array.

The example above sets the “mask” to true for points that are in classifications 1 or 2 and to false otherwise, causing points that are not classified 1 or 2 to be dropped from the point stream.

```python
import numpy as np

def filter(ins, outs):
    cls = ins['Classification']
    keep_classes = [1, 2]

    # Use the first test for our base array.
    keep = np.equal(cls, keep_classes[0])

    # For 1:n, test each predicate and join back
    # to our existing predicate array
    for k in range(1, len(keep_classes)):
        t = np.equal(cls, keep_classes[k])
        keep = np.logical_or(keep, t)
    outs['Mask'] = keep
```
keep = keep + t
outs['Mask'] = keep
return True

Note: filters.range (page 248) is a specialized filter that implements the exact functionality described in this Python operation. It is likely to be much faster than Python, but not as flexible. filters.python (page 276) is the tool you can use for prototyping point stream processing operations.

See also:
If you want to read a Pipeline (page 45) of operations into a numpy array, the PDAL Python extension (https://pypi.python.org/pypi/PDAL) is available.

Example pipeline

```json
[
  "file-input.las",
  {
    "type": "filters.smrf"
  },
  {
    "type": "filters.python",
    "script": "filter_pdal.py",
    "function": "filter",
    "module": "anything"
  },
  {
    "type": "writers.las",
    "filename": "file-filtered.las"
  }
]```
Module Globals

Three global variables are added to the Python module as it is run to allow you to get
*Dimensions* (page 287), *Metadata* (page 467), and coordinate system information.
Additionally, the *metadata* object can be set by the function to modify metadata for the
in-scope *filters.python* (page 276) *pdal::Stage* (page 553).

```python
def myfunc(ins, outs):
    print('schema: ', schema)
    print('srs: ', spatialreference)
    print('metadata: ', metadata)
    outs = ins
    return True
```

Updating metadata

The filter can update the global *metadata* dictionary as needed, define it as a *global* Python
variable for the function’s scope, and the updates will be reflected back into the pipeline from
that stage forward.

```python
def myfunc(ins, outs):
    global metadata
    metadata = {'name': 'root', 'value': 'a string', 'type': 'string',
                'description': 'a description', 'children': [{'name': 'filters.python',
                                                           'value': 52, 'type': 'integer', 'description': 'a filter description',
                                                           'children': []}, {'name': 'readers.faux', 'value':
                                                           'another string', 'type': 'string', 'description': 'a reader description',
                                                           'children': []}]
    return True
```

Passing Python objects

An JSON-formatted option can be passed to the filter representing a Python dictionary
containing objects you want to use in your function. This feature is useful in situations where
you wish to call *pipeline* (page 32) with substitutions.

If we needed to be able to provide the Z scaling factor of *Example Pipeline* (page 279) with a
Python argument, we can place that in a dictionary and pass that to the filter as a separate
argument. This feature allows us to be able easily reuse the same basic Python function while
substituting values as necessary.

```json
[
    "input.las",
```

(continues on next page)
With that option set, you can now fetch the \textit{pdalargs} (page 281) dictionary in your Python script and use it:

```python
import numpy as np

def multiply_z(ins, outs):
    Z = ins['Z']
    Z = Z * float(pdalargs['factor'])
    outs['Z'] = Z
    return True
```

### Standard output and error

A \texttt{redirector} module is available for scripts to output to PDAL’s log stream explicitly. The module handles redirecting \texttt{sys.stderr} and \texttt{sys.stdout} for you transparently, but it can be used directly by scripts. See the PDAL source code for more details.

### Options

- **script** When reading a function from a separate Python (http://python.org/) file, the file name to read from.
- **source** The literal Python (http://python.org/) code to execute, when the script option is not being used.
- **module** The Python module that is holding the function to run. [Required]
- **function** The function to call. [Required]
- **add_dimension** A dimension name or an array of dimension names to add to the pipeline that do not already exist.
- **pdalargs** A JSON dictionary of items you wish to pass into the modules globals as the \texttt{pdalargs} object.
where  An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

where_merge  A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

filters.julia

The Julia Filter allows Julia (https://julialang.org/) software to be embedded in a Pipeline (page 45) that allows modification of PDAL points through a TypedTables (https://github.com/JuliaData/TypedTables.jl) datatype.

The supplied julia function must take a TypedTables (https://github.com/JuliaData/TypedTables.jl) FlexTable as an argument and return the same object (with modifications).

```
Warning: The returned Table contains all the Dimensions (page 287) of the incoming ins Table
```

Dynamic Plugin

This stage requires a dynamic plugin to operate

```
module MyModule
  using TypedTables

  function multiply_z(ins)
    for n in 1:length(ins)
      ins[n] = merge(ins[n], (; :Z => row.Z * 10.0))
    end
    return ins
  end
end
```

If you want write a dimension that might not be available, you can specify it with the add_dimension_ option:

(continues on next page)
To create more than one dimension, this option also accepts an array:

```json
"add_dimension": [ "NewDimensionOne", "NewDimensionTwo",
"NewDimensionThree" ]
```

You can also specify the :ref:`type <types>` of the dimension using an `=`.

```json
"add_dimension": "NewDimensionOne=uint8"
```

## Filter Example

```json
[
    "file-input.las",
    {
        "type": "filters.smrf"
    },
    {
        "type": "filters.julia",
        "script": "filter_z.jl",
        "function": "filter_z",
        "module": "MyModule"
    },
    {
        "type": "writers.las",
        "filename": "file-filtered.las"
    }
]
```

The JSON pipeline file referenced the external *filter_z.jl* Julia (https://julialang.org/) script, which removes points with the Z coordinate by less than 420.

```julia
module MyModule
    using TypedTables
```

(continues on next page)
function filter_z(ins)
    return filter(p -> p.Z > 420, ins)
end
end

Modification Example

The JSON pipeline file referenced the external multiply_z.jl Julia (https://julialang.org/) script, which scales the Z coordinate by a factor of 10.

module MyModule
    using TypedTables

    function multiply_z(ins)
        for n in 1:length(ins)
            ins[n] = merge(ins[n], (; :Z => row.Z * 10.0))
        end
        return ins
    end
end
Options

**script** When reading a function from a separate Julia (https://julialang.org/) file, the file name to read from.

**source** The literal Julia (https://julialang.org/) code to execute, when the script option is not being used.

**module** The Julia module that is holding the function to run. [Required]

**function** The function to call. [Required]

**add_dimension** A dimension name or an array of dimension names to add to the pipeline that do not already exist.

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If true, the skipped points are added to the first point view returned by the skipped filter. If false, skipped points are placed in their own point view. If auto, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: auto]

*filters.matlab* (page 275) Embed MATLAB software in a pipeline.

*filters.python* (page 276) Embed Python software in a pipeline.

*filters.julia* (page 282) Embed Julia software in a pipeline.

7.4.10 Other

**filters.streamcallback**

The Stream Callback Filter provides a simple hook for a user-specified action to occur for each point. The stream callback filter is for use by C++ programmers extending PDAL functionality and isn’t useful to end users.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations
Options

**where** An expression that limits points passed to a filter. Points that don’t pass the expression skip the stage but are available to subsequent stages in a pipeline. [Default: no filtering]

**where_merge** A strategy for merging points skipped by a ‘where’ option when running in standard mode. If **true**, the skipped points are added to the first point view returned by the skipped filter. If **false**, skipped points are placed in their own point view. If **auto**, skipped points are merged into the returned point view provided that only one point view is returned and it has the same point count as it did when the filter was run. [Default: **auto**]

*filters.streamcallback* (page 285) Provide a hook for a simple point-by-point callback.
8.1 Dimensions

All point data in PDAL is stored as a set of dimensions. Dimensions have a name and a data type. The data type is determined at runtime, but a default data type for each dimension is listed below, along with the name of the dimension and its description.

The following table provides a list of known dimension names you can use in Filters (page 150), Writers (page 113), and Readers (page 54).

Note: Types are default types. Stage developers should set the dimension type explicitly if the default dimension isn’t suitable.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>uint16</td>
<td>Alpha</td>
</tr>
<tr>
<td>Amplitude</td>
<td>float</td>
<td>This is the ratio of the received power to the power received at the detection limit expressed in dB</td>
</tr>
<tr>
<td>Anisotropy</td>
<td>double</td>
<td>Anisotropy of a point; larger values indicate strong variance in multiple dimensions.</td>
</tr>
<tr>
<td>Azimuth</td>
<td>double</td>
<td>Scanner azimuth</td>
</tr>
<tr>
<td>BackgroundRadiation</td>
<td>float</td>
<td>A measure of background radiation.</td>
</tr>
<tr>
<td>Blue</td>
<td>uint16</td>
<td>Blue image channel value</td>
</tr>
<tr>
<td>ClassFlags</td>
<td>uint8</td>
<td>Class Flags</td>
</tr>
<tr>
<td>Classification</td>
<td>uint8</td>
<td>ASPRS classification. 0 for no classification. See LAS specification for details.</td>
</tr>
<tr>
<td>ClusterID</td>
<td>int64_t</td>
<td>ID assigned to a point by a point-clustering algorithm.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Curvature</td>
<td>double</td>
<td>Curvature of surface at this point</td>
</tr>
<tr>
<td>DemantkeVerticality</td>
<td>double</td>
<td>Verticality of a point; larger values indicate vertical structure (Demantke’s variation)</td>
</tr>
<tr>
<td>Density</td>
<td>double</td>
<td>Estimate of point density</td>
</tr>
<tr>
<td>Deviation</td>
<td>float</td>
<td>Difference between the shape of the reference pulse and the return pulse. A larger value for deviation indicates larger distortion.</td>
</tr>
<tr>
<td>EchoRange</td>
<td>double</td>
<td>Echo Range</td>
</tr>
<tr>
<td>EdgeOfFlightLine</td>
<td>uint8</td>
<td>Indicates the end of scanline before a direction change with a value of 1 - 0 otherwise</td>
</tr>
<tr>
<td>Eigenentropy</td>
<td>double</td>
<td>Eigenentropy of a point; small values indicate more ordered regions, while large values indicate disorder.</td>
</tr>
<tr>
<td>Eigenvalue-Sum</td>
<td>double</td>
<td>Sum of computed eigenvalues.</td>
</tr>
<tr>
<td>ElevationCentroid</td>
<td>double</td>
<td>Elevation Centroid</td>
</tr>
<tr>
<td>ElevationHigh</td>
<td>double</td>
<td>Elevation High</td>
</tr>
<tr>
<td>ElevationLow</td>
<td>double</td>
<td>Elevation Low</td>
</tr>
<tr>
<td>Flag</td>
<td>uint8</td>
<td>Flag</td>
</tr>
<tr>
<td>GpsTime</td>
<td>double</td>
<td>GPS time that the point was acquired</td>
</tr>
<tr>
<td>Green</td>
<td>uint16</td>
<td>Green image channel value</td>
</tr>
<tr>
<td>HeightAbove-Ground</td>
<td>double</td>
<td>Height Above Ground</td>
</tr>
<tr>
<td>Infrared</td>
<td>uint16</td>
<td>Infrared</td>
</tr>
<tr>
<td>Intensity</td>
<td>uint16</td>
<td>Representation of the pulse return magnitude</td>
</tr>
<tr>
<td>InternalTime</td>
<td>double</td>
<td>Scanner’s internal time when the point was acquired, in seconds</td>
</tr>
<tr>
<td>IsPpsLocked</td>
<td>uint8</td>
<td>The external PPS signal was found to be synchronized at the time of the current laser shot.</td>
</tr>
<tr>
<td>LatitudeCentroid</td>
<td>double</td>
<td>Latitude Centroid</td>
</tr>
<tr>
<td>LatitudeHigh</td>
<td>double</td>
<td>Latitude High</td>
</tr>
<tr>
<td>LatitudeLow</td>
<td>double</td>
<td>Latitude Low</td>
</tr>
</tbody>
</table>
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linearity</td>
<td>double</td>
<td>Linearity of a point; larger values indicate more linear regions.</td>
</tr>
<tr>
<td>Longitude-Centroid</td>
<td>double</td>
<td>Longitude Centroid</td>
</tr>
<tr>
<td>Longitude-High</td>
<td>double</td>
<td>Longitude High</td>
</tr>
<tr>
<td>Longitude-Low</td>
<td>double</td>
<td>Longitude Low</td>
</tr>
<tr>
<td>LvisLfid</td>
<td>uint64</td>
<td>LVIS_LFID</td>
</tr>
<tr>
<td>Mark</td>
<td>uint8</td>
<td>Mark</td>
</tr>
<tr>
<td>NNDistance</td>
<td>double</td>
<td>Distance metric related to a point’s nearest neighbors.</td>
</tr>
<tr>
<td>NormalX</td>
<td>double</td>
<td>X component of a vector normal to surface at this point</td>
</tr>
<tr>
<td>NormalY</td>
<td>double</td>
<td>Y component of a vector normal to surface at this point</td>
</tr>
<tr>
<td>NormalZ</td>
<td>double</td>
<td>Z component of a vector normal to surface at this point</td>
</tr>
<tr>
<td>NumberOfReturns</td>
<td>uint8</td>
<td>Total number of returns for a given pulse.</td>
</tr>
<tr>
<td>OffsetTime</td>
<td>uint32</td>
<td>Milliseconds from first acquired point</td>
</tr>
<tr>
<td>Omit</td>
<td>uint8_t</td>
<td>Used to shallowly mark a point as being omitted without removing it</td>
</tr>
<tr>
<td>Omnivariance</td>
<td>double</td>
<td>Omnivariance of a point; cube root of the product of all eigenvalues.</td>
</tr>
<tr>
<td>OptimalKNN</td>
<td>uint64</td>
<td>Optimal number of k nearest neighbors, such that eigenentropy is minimized.</td>
</tr>
<tr>
<td>OptimalRadius</td>
<td>double</td>
<td>Radius corresponding to optimal k nearest neighbors, such that eigenentropy is minimized.</td>
</tr>
<tr>
<td>OriginId</td>
<td>uint32</td>
<td>A file source ID from which the point originated. This ID is global to a derivative dataset which may be aggregated from multiple files.</td>
</tr>
<tr>
<td>PassiveSignal</td>
<td>int32</td>
<td>Relative passive signal</td>
</tr>
<tr>
<td>PassiveX</td>
<td>double</td>
<td>Passive X footprint</td>
</tr>
<tr>
<td>PassiveY</td>
<td>double</td>
<td>Passive Y footprint</td>
</tr>
<tr>
<td>PassiveZ</td>
<td>double</td>
<td>Passive Z footprint</td>
</tr>
<tr>
<td>Pdop</td>
<td>float</td>
<td>GPS PDOP (dilution of precision)</td>
</tr>
<tr>
<td>Pitch</td>
<td>float</td>
<td>Pitch in degrees</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planarity</td>
<td>double</td>
<td>Planarity of a point; larger values indicate more planar regions.</td>
</tr>
<tr>
<td>PointId</td>
<td>uint32</td>
<td>An explicit representation of point ordering within a file, which allows this usually-implicit information to be preserved when re-ordering points.</td>
</tr>
<tr>
<td>PointSourceId</td>
<td>uint16</td>
<td>File source ID from which the point originated. Zero indicates that the point originated in the current file.</td>
</tr>
<tr>
<td>PulseWidth</td>
<td>float</td>
<td>Laser received pulse width (digitizer samples)</td>
</tr>
<tr>
<td>Red</td>
<td>uint16</td>
<td>Red image channel value</td>
</tr>
<tr>
<td>Reflectance</td>
<td>float</td>
<td>Ratio of the received power to the power that would be received from a white diffuse target at the same distance expressed in dB. The reflectance represents a range independent property of the target. The surface normal of this target is assumed to be in parallel to the laser beam direction.</td>
</tr>
<tr>
<td>Reflected-Pulse</td>
<td>int32</td>
<td>Relative reflected pulse signal strength</td>
</tr>
<tr>
<td>ReturnNumber</td>
<td>uint8</td>
<td>Pulse return number for a given output pulse. A given output laser pulse can have many returns, and they must be marked in order, starting with 1</td>
</tr>
<tr>
<td>Roll</td>
<td>float</td>
<td>Roll in degrees</td>
</tr>
<tr>
<td>ScanAngleRank</td>
<td>float</td>
<td>Angle degree at which the laser point was output from the system, including the roll of the aircraft. The scan angle is based on being nadir, and -90 the left side of the aircraft in the direction of flight</td>
</tr>
<tr>
<td>ScanChannel</td>
<td>uint8</td>
<td>Scan Channel</td>
</tr>
<tr>
<td>ScanDirectionFlag</td>
<td>uint8</td>
<td>Direction at which the scanner mirror was traveling at the time of the output pulse. A value of 1 is a positive scan direction, and a bit value of 0 is a negative scan direction, where positive scan direction is a scan moving from the left side of the in-track direction to the right side and negative the opposite</td>
</tr>
<tr>
<td>Scattering</td>
<td>double</td>
<td>Scattering of a point; larger values indicate complex (scattered) 3D regions.</td>
</tr>
<tr>
<td>ShotNumber</td>
<td>uint64</td>
<td>Shot Number</td>
</tr>
<tr>
<td>StartPulse</td>
<td>int32</td>
<td>Relative pulse signal strength</td>
</tr>
<tr>
<td>SurfaceVariation</td>
<td>double</td>
<td>Surface variation of a point; larger values indicate higher surface variation.</td>
</tr>
<tr>
<td>TextureU</td>
<td>double</td>
<td>U component of a texture location at this point</td>
</tr>
<tr>
<td>TextureV</td>
<td>double</td>
<td>V component of a texture location at this point</td>
</tr>
<tr>
<td>TextureW</td>
<td>double</td>
<td>W component of a texture location at this point</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserData</td>
<td>uint8</td>
<td>Unspecified user data</td>
</tr>
<tr>
<td>Verticality</td>
<td>double</td>
<td>Verticality of a point; larger values indicate vertical structure.</td>
</tr>
<tr>
<td>W</td>
<td>double</td>
<td>W coordinate</td>
</tr>
<tr>
<td>WanderAngle</td>
<td>double</td>
<td>Wander Angle</td>
</tr>
<tr>
<td>X</td>
<td>double</td>
<td>X coordinate</td>
</tr>
<tr>
<td>XBodyAccel</td>
<td>double</td>
<td>X Body Acceleration</td>
</tr>
<tr>
<td>XBodyAngRate</td>
<td>double</td>
<td>X Body Angle Rate</td>
</tr>
<tr>
<td>XVelocity</td>
<td>double</td>
<td>X Velocity</td>
</tr>
<tr>
<td>Y</td>
<td>double</td>
<td>Y coordinate</td>
</tr>
<tr>
<td>YBodyAccel</td>
<td>double</td>
<td>Y Body Acceleration</td>
</tr>
<tr>
<td>YBodyAngRate</td>
<td>double</td>
<td>Y Body Angle Rate</td>
</tr>
<tr>
<td>YVelocity</td>
<td>double</td>
<td>Y Velocity</td>
</tr>
<tr>
<td>Z</td>
<td>double</td>
<td>Z coordinate</td>
</tr>
<tr>
<td>ZBodyAccel</td>
<td>double</td>
<td>Z Body Acceleration</td>
</tr>
<tr>
<td>ZBodyAngRate</td>
<td>double</td>
<td>Z Body Angle Rate</td>
</tr>
<tr>
<td>ZVelocity</td>
<td>double</td>
<td>Z Velocity</td>
</tr>
</tbody>
</table>
### 9.1 Types

PDAL supports the standard integral and floating point types for *dimensions* (page 287). This table lists the types and associated strings that can be used to describe the types in options.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size in Bits</th>
<th>Text Representations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed Integer</td>
<td>8</td>
<td><code>int8, int8_t, char</code></td>
</tr>
<tr>
<td>Signed Integer</td>
<td>16</td>
<td><code>int16, int16_t, short</code></td>
</tr>
<tr>
<td>Signed Integer</td>
<td>32</td>
<td><code>int32, int32_t, int</code></td>
</tr>
<tr>
<td>Signed Integer</td>
<td>64</td>
<td><code>int64, int64_t, long</code></td>
</tr>
<tr>
<td>Unsigned Integer</td>
<td>8</td>
<td><code>uint8, uint8_t, uchar</code></td>
</tr>
<tr>
<td>Unsigned Integer</td>
<td>16</td>
<td><code>uint16, uint16_t, ushort</code></td>
</tr>
<tr>
<td>Unsigned Integer</td>
<td>32</td>
<td><code>uint32, uint32_t, uint</code></td>
</tr>
<tr>
<td>Unsigned Integer</td>
<td>64</td>
<td><code>uint64, uint64_t, ulong</code></td>
</tr>
<tr>
<td>Floating Point</td>
<td>32</td>
<td><code>float, float32</code></td>
</tr>
<tr>
<td>Floating Point</td>
<td>64</td>
<td><code>double, float64</code></td>
</tr>
</tbody>
</table>
10.1 Python

PDAL provides Python support in two significant ways. First it embeds Python to allow you to write Python programs that interact with data using filters.python (page 276) filter. Second, it extends Python by providing an extension that Python programmers can use to leverage PDAL capabilities in their own applications.

Note: PDAL’s Python story always revolves around Numpy (http://www.numpy.org/) support. PDAL’s data is provided to both the filters and the extension as Numpy arrays.

10.1.1 Versions

PDAL supports both Python 3.5+. Continuous Integration (page 518) tests Python Linux, OSX, and Windows.

10.1.2 Embed

PDAL allows users to embed Python functions inline with other Pipeline (page 45) processing operations. The purpose of this capability is to allow users to write small programs that implement interesting actions without requiring a full C++ development activity of building a PDAL stage to implement it. A Python filter is an opportunity to interactively and iteratively prototype a data operation without strong considerations of performance or generality. If something works well enough, maybe one takes on the effort to formalize it, but that isn’t necessary. PDAL’s embed of Python allows you to be as grimy as you need to get the job done.
10.1.3 Extend

PDAL provides a Python extension (https://pypi.python.org/pypi/PDAL) that gives users access to executing pipeline (page 45) instantiations and capturing the results as Numpy (http://www.numpy.org/) arrays. This mode of operation is useful if you are looking to have PDAL simply act as your data format and processing handler.

Python extension users are expected to construct their own PDAL pipeline (page 45) using Python’s json library, or whatever other libraries they wish to manipulate JSON. They then feed it into the extension and get back the results as Numpy (http://www.numpy.org/) arrays:

```python
json = ""
[
    "1.2-with-color.las",
    {"type": "filters.sort",
     "dimension": "X"
    }
]
"

import pdal
pipeline = pdal.Pipeline(json)
count = pipeline.execute()
arrays = pipeline.arrays
metadata = pipeline.metadata
log = pipeline.log
```

Fig. 1: Embedding a Python function to take Z values read from a `readers.las` (page 71) and then output them to a `writers.bpf` (page 114).
Installation

The PDAL Python extension requires a working PDAL installation (page 13). Unless you choose the Conda installation method, make sure that you a current, working version of PDAL before installing the extension.

Note: Previous to PDAL 2.1, Python support was spread across the embedded stages (readers.numpy (page 80) and filters.python (page 276)) which were installed by PDAL itself and the PDAL extension that was installed from PyPI. As of PDAL 2.1 and PDAL/python 2.3, both the embedded stages and the extension are installed from PyPI.

Installation Using pip

As administrator, you can install PDAL using pip:

`pip install PDAL`

Note: To install pip please read here (https://pip.pypa.io/en/stable/installing/)

Installation from Source

PDAL Python support is hosted in a separate repository than PDAL itself at GitHub (https://github.com/PDAL/python). If you have a working PDAL installation and a working Python installation, you can install the extension using the following procedure on Unix. The procedure on Windows is similar

```
$ git clone https://github.com/PDAL/python pdalextension
$ cd pdalextension
$ pip install .
```

Install using Conda

The PDAL Python support can also be installed using the Conda (https://conda.io/docs/) package manager. An advantage of using Conda to install the extension is that Conda will install PDAL. We recommend installing PDAL and the PDAL Python extension in an environment other than the base environment. To install in an existing environment, use the following
conda install -n <environment name> -c conda-forge python-pdal

Use the following command to install PDAL and the PDAL Python extension into a new environment and activate that environment

conda create -n <environment name> -c conda-forge python-pdal
conda activate <environment name>

Note: The official pdal and python-pdal packages reside in the conda-forge channel, which can be added via conda config or manually specified with the -c option, as shown in the examples above.
11.1 Java

PDAL provides Java bindings to use PDAL on JVM (https://github.com/PDAL/java). It is released independently from PDAL itself as of PDAL 1.7. Native binaries are prebuilt for Linux and MacOS and delivered in a jar, so there is no need in building PDAL with a special flag or building JNI binaries manually.

The project consists of the following modules:

- **pdal-native** - with packed OS specific libraries to link PDAL to JNI proxy classes. Dependency contains bindings for x86_64-darwin and x86_64-linux, other versions are not supported yet.
- **pdal** - with the core bindings functionality.
- **pdal-scala** - a Scala API package that simplifies PDAL Pipeline construction.

11.1.1 Versions

PDAL JNI major version usually follows PDAL versioning i.e. pdal-java 1.8.x was built and tested against PDAL 1.8.x and pdal-java 2.1.x against PDAL 2.x.x.

11.1.2 Using PDAL Java bindings

PDAL provides JNI bindings (https://docs.oracle.com/javase/8/docs/technotes/guides/jni/index.html) that gives users access to executing pipeline (page 45) instantiations and capturing the results in Java interfaces. This mode of operation is useful if you are looking to have PDAL simply act as your data format and processing handler.

Users are expected to construct their own PDAL pipeline (page 45), execute it, and retrieve points into Java memory:
import io.pdal._

val json = ""
  |
  | "pipeline": [  
  |   | "filename": "1.2-with-color.las",  
  |   | "spatialreference": "EPSG:2993"  
  |   | },  
  |   | {  
  |   |   | "type": "filters.reprojection",  
  |   |   | "out_srs": "EPSG:3857"  
  |   | },  
  |   | {  
  |   |   | "type": "filters.delaunay"  
  |   | }  
  |
  |}  
"").stripMargin

val pipeline = Pipeline(json)
pipeline.validate() // check if our JSON and options were good
pipeline.setLogLevel(8) // make it really noisy
pipeline.execute() // execute the pipeline
val metadata: String = pipeline.getMetadata() // retrieve metadata
val pvs: PointViewIterator = pipeline<PointView> // iterator over PointViews
val pv: PointView = pvs.next() // let's take the first PointView

// load all points into JVM memory
// PDAL provides operations on PDAL points that are loaded in this case into JVM memory as a single Array[Byte]
val pointCloud: PointCloud = pv.getPointCloud()
val x: Double = pointCloud.getDouble(0, DimType.X) // get a point with PointId = 0 and only a single dimension

// in some cases it is not necessary to load everything into JVM memory
// so it is possible to get only required points directly from the PointView
val y: Double = pv.getDouble(0, DimType.Y)

// it is also possible to get access to the triangular mesh generated via PDAL
(continues on next page)
val mesh: TriangularMesh = pv.getTriangularMesh()
// the output is an Array of Triangles
// Each Triangle contains PointIds from the PDAL point table
val triangles: Array[Triangle] = mesh.asArray

pv.close()
pipeline.close()

11.1.3 Using PDAL Scala

PDAL Scala project introduces a DSL to simplify PDAL Pipeline construction (this is the same pipeline from the section above):

```scala
import io.pdal._
import io.pdal.pipeline._

val expression =
  ReadLas("1.2-with-color.las", spatialreference = Some("EPSG:2993")) ~
  FilterReprojection("EPSG:3857") ~
  FilterDelaunay()

val pipeline = expression.toPipeline
pipeline.validate() // check if our JSON and options were good
pipeline.setLogLevel(8) // make it really noisy
pipeline.execute() // execute the pipeline
val metadata: String = pipeline.getMetadata() // retrieve metadata
val pvs: PointViewIterator = pipeline.getPointViews() // iterator over PointViews
val pv: PointView = pvs.next() // let's take the first PointView

// load all points into JVM memory
// PointCloud provides operations on PDAL points that are loaded in this case into JVM memory as a single Array[Byte]
val pointCloud: PointCloud = pv.getPointCloud()
val x: Double = pointCloud.getDouble(0, DimType.X) // get a point with PointId = 0 and only a single dimensions

// in some cases it is not neccessary to load everything into JVM memory
// so it is possible to get only required points directly from the PointView
```

(continues on next page)
It covers PDAL 2.0.x, but to use any custom DSL that is not covered by the current Scala API you can use `RawExpr` type to build a `Pipeline Expression`:

```scala
import io.pdal._
import io.pdal.pipeline._
import io.circe.syntax._

val pipelineWithRawExpr =
  ReadLas("1.2-with-color.las") ~
  RawExpr(Map("type" -> "filters.crop").asJson) ~
  WriteLas("1.2-with-color-out.las")
```

**Installation**

PDAL Java artifacts are cross published for Scala 2.13, 2.12 and 2.11. However, if it is not required, a separate artifact that has no Scala specific artifact postfix is published as well.

```scala
// pdal is published to maven central, but you can use following repos in addition
resolvers ++= Seq(
  Resolver.sonatypeRepo("releases"),
  Resolver.sonatypeRepo("snapshots") // for snapshots
)

libraryDependencies ++= Seq(
  "io.pdal" %% "pdal" % "x.x.x", // core library
  "io.pdal" % "pdal-native" % "x.x.x", // jni binaries
  "io.pdal" %% "pdal-scala" % "x.x.x" // if scala core library (if
  required)
)
```

The latest version is: [maven central 2.2.0](https://search.maven.org/search?q=g:io.pdal)
There is also an example SBT PDAL Demo project (https://github.com/PDAL/java/tree/master/examples/pdal-jni) in the bindings repository, that can be used for a quick start.

## Compilation

### Development purposes (including binaries) compilation:

1. Install PDAL (using brew / package managers (unix) / build from sources / etc)
2. Build native libs .sbt native/nativeCompile (optionally, binaries would be built during tests run)
3. Run .sbt core/test to run PDAL tests

### Only Java development purposes compilation:

1. Provide $LD_LIBRARY_PATH or $DYLD_LIBRARY_PATH
2. If you don’t want to provide global variable you can pass 
   -Djava.library.path=<path> into sbt:
   
   ./sbt -Djava.library.path=<path>
3. Set PDAL_DEPEND_ON_NATIVE=false (to disable native project build)
4. Run PDAL_DEPEND_ON_NATIVE=false ./sbt

If you would like to use your own bindings binary, it is necessary to set java.library.path:

```scala
// Mac OS X example with manual JNI installation
// cp -f native/target/resource_managed/main/native/x86_64-darwin/
//    libpdaljni.2.1.dylib /usr/local/lib/libpdaljni.2.1.dylib
// place built binary into /usr/local/lib, and pass java.library.
//    path to your JVM
javaOptions += "-Djava.library.path=/usr/local/lib"
```

You can use pdal-native dep in case you don’t have installed JNI bindings and to avoid steps described above. Dependency contains bindings for x86_64-darwin and x86_64-linux, other versions are not supported yet.
12.1 Tutorials

This section provides a collection of tutorials on how to use the PDAL Applications (page 25) and Pipelines (page 45) to process data.

Note: Users looking for documentation on how to contribute to PDAL should look here (page 445) and users looking to use the PDAL API in their own applications should look here (page 519).

12.1.1 Reading with PDAL

Author  Bradley Chambers
Contact brad.chambers@gmail.com
Date  01/21/2015

Contents

• Reading with PDAL (page 305)
  – A basic inquiry example (page 306)
  – A conversion example (page 307)
    * Metadata (page 307)
  – A Pipeline Example (page 308)
    * Simple conversion (page 308)
    * Loop a directory and filter it through a pipeline (page 308)
This tutorial is an introduction to using PDAL to read data using pdal from the command line.

**A basic inquiry example**

Our first example to demonstrate PDAL’s utility will be to simply query an LAS file to determine the data that are in it in the very first point.

---

**Note:** The interesting.las file in these examples can be found on github.

---

*pdal info* outputs JavaScript JSON (http://www.json.org/).

```bash
$ pdal info interesting.las -p 0
```

```
{
  "filename": "interesting.las",
  "pdal_version": "1.0.1 (git-version: 80644d)",
  "points":
  {
    "point":
    {
      "Blue": 88,
      "Classification": 1,
      "EdgeOfFlightLine": 0,
      "GpsTime": 245381,
      "Green": 77,
      "Intensity": 143,
      "NumberOfReturns": 1,
      "PointId": 0,
      "PointSourceId": 7326,
      "Red": 68,
      "ReturnNumber": 1,
      "ScanAngleRank": -9,
      "ScanDirectionFlag": 1,
      "UserData": 132,
      "X": 637012,
      "Y": 849028,
      "Z": 431.66
    }
  }
}
```
A conversion example

Conversion of data from one format to another may be lossy, in that some data in the source format may not be representable in the same format or at all in the destination format. For example, some formats don’t support spatial references for point data, some have no metadata support and others have limited dimension support. Even when data types are supported in both source and destination formats, there may be limitations with regard to data type, precision or, scaling. PDAL attempts to convert data as accurately as possible, but you should make sure that you’re aware of the capabilities of the data formats you’re using.

```bash
$ pdal translate interesting.las output.txt
```

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Intensity</th>
<th>ReturnNumber</th>
<th>NumberOfReturns</th>
<th>ScanDirectionFlag</th>
<th>EdgeOfFlightLine</th>
<th>Classification</th>
<th>ScanAngleRank</th>
<th>UserData</th>
<th>PointSourceId</th>
<th>Time</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>637012.24</td>
<td>849028.31</td>
<td>431.66</td>
<td>143</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-9</td>
<td>132</td>
<td>7326</td>
<td>245381</td>
<td>68</td>
<td>77</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>636896.33</td>
<td>849087.70</td>
<td>446.39</td>
<td>18</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-11</td>
<td>128</td>
<td>7326</td>
<td>245381</td>
<td>54</td>
<td>66</td>
<td>68</td>
</tr>
<tr>
<td>636784.74</td>
<td>849106.66</td>
<td>426.71</td>
<td>118</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-10</td>
<td>122</td>
<td>7326</td>
<td>245382</td>
<td>112</td>
<td>97</td>
<td>114</td>
</tr>
<tr>
<td>636699.38</td>
<td>848991.01</td>
<td>425.39</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-6</td>
<td>124</td>
<td>7326</td>
<td>245383</td>
<td>178</td>
<td>138</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>636601.87</td>
<td>849018.60</td>
<td>425.10</td>
<td>124</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-4</td>
<td>126</td>
<td>7326</td>
<td>245383</td>
<td>134</td>
<td>104</td>
<td>134</td>
</tr>
<tr>
<td>636451.97</td>
<td>849250.59</td>
<td>435.17</td>
<td>48</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-9</td>
<td>122</td>
<td>7326</td>
<td>245384</td>
<td>99</td>
<td>85</td>
<td>95</td>
</tr>
</tbody>
</table>

The text format supports all point attributes, but provides no support for metadata such as the input spatial reference system or the LAS header fields, such as UUID. You may need to preserve some more information as part of your conversion to make it useful down the road.

**Metadata**

PDAL carries metadata (page 467) for each stage through the PDAL processing pipeline (page 45). The metadata can be written in JSON form using the pdal info command.

```bash
$ pdal info --metadata interesting.las
```

This produces metadata that looks like this. You can use your JSON manipulation tools to extract this information. For formats that do not have the ability to preserve this metadata internally, you can keep a .json file alongside the .txt file as auxiliary information.
A Pipeline Example

The full power of PDAL comes in the form of pipeline (page 32) invocations. Pipelines allow you to take advantage of PDAL’s ability to manipulate data as they are converted. This section will provide a basic example and demonstration of pipeline usage. See the pipeline specification (page 45), for more detailed exposition of the topic.

The pipeline (page 32) describes a series of processing stages to be performed in JSON format. Each stage can be provided a set of options that control the details of processing. PDAL is single-threaded and stages are executed in a linear order. Some stages support what is known as “stream mode”. If all stages in a pipeline support stream mode the command is run using using stream mode to reduce the memory processing footprint. Even when run in stream mode, execution is single-threaded and can be thought of as linear.

Simple conversion

The following JSON (http://www.json.org/) document defines a pipeline that takes the file.las LAS (http://www.asprs.org/a/society/committees/standards/lidar_exchange_format.html) file and converts it to a new file called output.las.

```
[  
    "file.las",
    "output.las"
]
```

Loop a directory and filter it through a pipeline

This bash script loops through a directory and pushes the las files through a pipeline, substituting the input and output as it goes.

```
ls *.las | cut -d. -f1 | xargs -P 20 -I{} pdal pipeline -i /path/to/\n   proj.json --readers.las.filename={}.las --writers.las.
   --filename=output/{}.laz
```

Here is an example doing something similar with Windows PowerShell

```
$indir="Documents\inlas"
outdir="Documents\outlas"
get-childitem $indir |
foreach-object {
    if ($_.extension -ne ".las") {
        continue
    }
} (continues on next page)"
12.1.2 Reading data from EPT

Introduction

This tutorial describes how to use Conda (https://conda.io), Entwine (https://entwine.io), PDAL (https://pdal.io), and GDAL (https://gdal.org) to read data from the USGS 3DEP AWS Public Dataset (https://www.usgs.gov/news/usgs-3dep-lidar-point-cloud-now-available-amazon-public-dataset). We will be using PDAL’s readers.ept (https://pdal.io/stages/readers.ept.html) to fetch data, we will filter it for noise using filters.outlier (https://pdal.io/stages/filters.outlier.html), we will classify the data as ground/not-ground using filters.smrf (https://pdal.io/stages/filters.smrf.html), and we will write out a digital terrain model with writers.gdal (page 119). Once our elevation model is constructed, we will use GDAL gdaldem (https://www.gdal.org/gdaldem.html) operations to create hillshade, slope, and color relief.

Install Conda

We first need to install PDAL, and the most convenient way to do that is by installing Miniconda (https://docs.conda.io/en/latest/miniconda.html). Select the 64-bit installer for your platform and install it as directed.

Install PDAL

Once Miniconda is installed, we can install PDAL into a new Conda Environment (https://docs.conda.io/projects/conda/en/latest/user-guide/concepts.html) that we created for this tutorial. Open your Anaconda Shell and start issuing the following commands:

1. Create the environment

   conda create -n iowa -y

2. Activate the environment

   conda activate iowa

3. Install PDAL

   conda install pdal
4. Ensure PDAL works by listing the available drivers

```bash
pdal --drivers
```

Once you confirmed you see output similar to that in your shell, your PDAL installation should be good to go.

**Write the Pipeline**

PDAL uses the concept of pipelines (https://pdal.io/pipeline.html) to describe the reading, filtering, and writing of point cloud data. We will construct a pipeline that will do a number of things in succession.

![Pipeline diagram](image)

Fig. 1: Pipeline diagram. The data are read from the Entwine Point Tile (https://entwine.io/entwine-point-tile.html) resource at https://usgs.entwine.io for Iowa using `readers.ept` (page 56) and filtered through a number of steps until processing is complete. The data are then written to an `iowa.laz` and `iowa.tif` file.
Pipeline

1. Create a file called `iowa.json` with the following content:

```json
{
    "pipeline": [
        {
            "bounds": "([-10425171.940, -10423171.940], [5164494.710, 5166494.710])",
            "type": "readers.ept",
            "tag": "readdata"
        },
        {
            "limits": "Classification[7:7]",
            "type": "filters.range",
            "tag": "nonoise"
        },
        {
            "assignment": "Classification[]=0",
            "tag": "wipeclasses",
            "type": "filters.assign"
        },
        {
            "out_srs": "EPSG:26915",
            "tag": "reprojectUTM",
            "type": "filters.reprojection"
        },
        {
            "tag": "groundify",
            "type": "filters.smrf"
        },
        {
            "limits": "Classification[2:2]",
            "type": "filters.range",
            "tag": "classify"
        },
        {
            "filename": "iowa.laz",
            "tag": "iowa.laz"
        }
    ]
}
```

Note: The file is also available from https://gist.github.com/hobu/ee22084e24ed7e3c0d10600798a94c31 for convenient copy/paste.

(continues on next page)
"inputs": [ "classify" ],
"tag": "writerslas",
"type": "writers.las"
},
{
"filename": "iowa.tif",
"gdalopts": "tiled=yes, compress=deflate",
"inputs": [ "writerslas" ],
"nodata": -9999,
"output_type": "idw",
"resolution": 1,
"type": "writers.gdal",
"window_size": 6
}
]

### Stages

**readers.ept**

`readers.ept` (page 56) reads the point cloud data from the EPT resource on AWS. We give it a URL to the root of the resource in the `filename` option, and we also give it a `bounds` object to define the window in which we should select data from.

**Note:** The full URL to the EPT root file (`ept.json`) must be given to the `filename` parameter for PDAL 2.2+. This was a change in behavior of the `readers.ept` (page 56) driver.

The `bounds` object is in the form `([minx, maxx], [miny, maxy])`.

**Warning:** If you do not define a `bounds` option, PDAL will try to read the data for the entire state of Iowa, which is about 160 billion points. Maybe you have enough memory for this...
**filters.range**

The data we are selecting may have noise properly classified, and we can use *filters.range* (page 248) to keep all data that does not have a *Classification Dimensions* (page 287) value of 7.

![filters.range](image)

Fig. 3: The *filters.range* (page 248) filter utilizes range selection to allow users to select data for processing or removal. The *filters.mongoexpression* filter can be used for even more complex logic operations.

**filters.assign**

After removing points that have noise classifications, we need to reset all of the classification values in the point data. *filters.assign* (page 204) takes the expression *Classification [:]=0* and assigns the *Classification* for each point to 0.

![filters.assign](image)

Fig. 4: *filters.assign* (page 204) can also take in an option to apply assignments based on a conditional. If you want to assign values based on a bounding geometry, use *filters.overlay* (page 206).

**filters.reprojection**

The data on the AWS 3DEP Public Dataset are stored in Web Mercator (https://en.wikipedia.org/wiki/Web_Mercator_projection) coordinate system, which is not suitable for many operations. We need to reproject them into an appropriate UTM coordinate system (EPSG:26915 (https://epsg.io/32615)).

![filters.reprojection](image)

Fig. 5: *filters.reprojection* (page 222) can also take override the incoming coordinate system using the `a_srs` option.
**filters.smrf**

The Simple Morphological Filter (*filters.smrf* (page 155)) classifies points as ground or not-ground.

![filters.smrf](image)

Fig. 6: *filters.smrf* (page 155) provides a number of tuning options, but the defaults tend to work quite well for mixed urban environments on flat ground (ie, Iowa).

**filters.range**

After we have executed the SMRF filter, we only want to keep points that are actually classified as ground in our point stream. Selecting for points with `Classification[2:2]` does that for us.

![filters.range](image)

Fig. 7: Remove any point that is not ground classification for our DTM generation.

**writers.gdal**

Having filtered our point data, we’re now ready to write a raster digital terrain model with *writers.gdal* (page 119). Interesting options we choose here are to set the `nodata` value, specify only outputting the inverse distance weighted raster, and assigning a resolution of 1 (m). See *writers.gdal* (page 119) for more options.

![writers.gdal](image)

Fig. 8: Output a DTM at 1m resolution.
writers.las

We can also write a LAZ file containing the same points that were used to make the elevation model in the section above. See writers.las (page 125) for more options.

Fig. 9: Also output the LAZ file as part of our processing pipeline.

Execute the Pipeline

1. Save the PDAL pipeline in Pipeline (page 311) to a file called iowa.json
2. Invoke the PDAL pipeline (https://pdal.io/pipeline.html) command

   pdal pipeline iowa.json

   Add the --debug option if you would like information about how PDAL is fetching and processing the data.

   pdal pipeline iowa.json --debug

3. Save a color scheme to dem-colors.txt

   # Color ramp for Iowa State Campus
   270.187,250,250,250,255,270.2
   272.059,230,230,230,255,272.1
   272.835,209,209,209,255,272.8
   273.985,189,189,189,255,274
   276.204,168,168,168,255,276.2
   277.835,148,148,148,255,277.8
   279.199,128,128,128,255,279.2
   280.964,107,107,107,255,281
   282.809,87,87,87,255,282.8
   283.745,66,66,66,255,283.7
   284.547,46,46,46,255,284.5
   286.526,159,223,250,255,286.5
   296.901,94,139,156,255,296.9

4. Invoke gdaldem to colorize a PNG file for your TIFF

   gdaldem color-relief iowa.tif dem-colors.txt iowa-color.png
5. View your raster

12.1.3 LAS Reading and Writing with PDAL

Author  Howard Butler
Contact  howard@hobu.co
Date  3/27/2017

Table of Contents

- LAS Reading and Writing with PDAL (page 316)
  - Introduction (page 316)
  - LAS Versions (page 317)
  - Spatial Reference System (page 317)
  - Point Formats (page 320)
  - Extra Dimensions (page 322)
  - Required Header Fields (page 322)
  - Coordinate Scaling (page 324)
  - Compression (page 325)
  - PDAL Metadata (page 327)

This tutorial will describe reading and writing ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) data with PDAL, discuss the capabilities that PDAL readers.las (page 71) and writers.las (page 125) can provide for this format.

Introduction

ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) is probably the most commonly used LiDAR (https://en.wikipedia.org/wiki/Lidar) format, and PDAL’s support of LAS is important for many users of the library. This tutorial describes and demonstrates some of the capabilities the drivers provide, points out items to be aware of when using the drivers, and hopefully provides some examples you can use to get what you need out of the LAS drivers.
LAS Versions

There are five LAS versions – 1.0 to 1.4. Each iteration added some complexity to the format in terms of capabilities it supports, possible data types it stores, and metadata. Users of LAS must balance the features they need with the use of the data by downstream applications. While LAS support in some form is quite widespread throughout the industry, most applications do not support every feature of each version. PDAL works to provide many of these features, but it is also incomplete. Specifically, PDAL doesn’t support point formats that store waveform data.

Version Example

We can use the `minor_version` option of `writers.las` (page 125) to set the version PDAL should output. The following example will write a 1.1 version LAS file. Depending on the features you need, this may or may not be what you want.

```json
[
    {
        "type": "readers.las",
        "filename": "input.las"
    },
    {
        "type": "writers.las",
        "minor_version": 1,
        "filename": "output.las"
    }
]
```

Note: PDAL defaults to writing a LAS 1.2 version if no `minor_version` is specified or the `forward` option of `writers.las` (page 125) is not used to carry along a version from a previously read file.

Spatial Reference System

LAS 1.0 to 1.3 use GeoTIFF (https://trac.osgeo.org/geotiff/) keys for storing coordinate system information, while LAS 1.4 uses Well Known Text (https://en.wikipedia.org/wiki/Well-known_text#Coordinate_reference_systems). GeoTIFF is well-supported by most software that read LAS, but it is not possible to express some coordinate system specifics with GeoTIFF. WKT is supports more coordinate systems than GeoTIFF, but vendor-specific and later versions (WKT 2) may not be handled well.
Assignment Example

The PDAL writers.las (page 125) allows you to override or assign the coordinate system to an explicit value if you need. Often the coordinate system defined by a file might be incorrect or non-existent, and you can set this with PDAL.

The following example sets the a_srs option of the writers.las (page 125) to EPSG:4326.

```
[  
  { 
    "type" : "readers.las",
    "filename" : "input.las"
  },
  
  { 
    "type" : "writers.las",
    "a_srs" : "EPSG:4326",
    "filename" : "output.las"
  }
]
```

**Note:** Remember to set offset_x, offset_y, scale_x, and scale_y values to something appropriate if you are storing decimal degree data in LAS files. The special value auto can be used for the offset values, but you should set an explicit value for the scale values to prevent overdriving the precision of the data and disrupting Compression (page 325) with LASzip (http://laszip.org).

Vertical Datum Example

Vertical coordinate control is important in LiDAR (https://en.wikipedia.org/wiki/Lidar) and PDAL supports assignment and reprojection/transform of vertical coordinates using Proj.4 (http://proj4.org) and GDAL (http://gdal.org/). The coordinate system description magic happens in GDAL, and you assign a compound coordinate system (both vertical and horizontal definitions) using the following syntax:

**EPSG:4326+3855**

This assignment states typical 4326 horizontal coordinate system plus a vertical one that represents EGM08 (http://earth-info.nga.mil/GandG/wgs84/gravmod/egm2008/egm08_wgs84.html). In Well Known Text (https://en.wikipedia.org/wiki/Well-known_text#Coordinate_reference_systems), this coordinate system is described by:
As in Assignment Example (page 318), it is common to need to reassign the coordinate system. The following example defines both the horizontal and vertical coordinate system for a file to UTM Zone 15N NAD83 (http://epsg.io/26915) for horizontal and NAVD88 (http://epsg.io/5703) for the vertical.

```json
{
    "type": "readers.las",
    "filename": "input.las"
},
{
    "type": "writers.las",
    "a_srs": "EPSG:26915+5703",
    "filename": "output.las"
}
```

**Note:** Any coordinate system description format supported by GDAL’s SetFromUserInput (http://www.gdal.org/ogr__srs__api_8h.html#a927749db01cec3af8aa5e577d032956bk) method can be used to assign or set the coordinate system in PDAL. This includes WKT, Proj.4 (http://proj4.org) definitions, or OGC URNs. It is your responsibility, however, to escape or massage any input data to make it be valid JSON.
Reprojection Example

A common desire is to transform the coordinates of an ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) file from one coordinate system to another. The mechanism to do that with PDAL is `filters.reprojection` (page 222).

```json
[
  
  
  "type" : "readers.las",
  "filename" : "input.las"
  
  ],
  
  "type" : "filters.reprojection",
  "out_srs" : "EPSG:26915"
  
  ],
  
  "type" : "writers.las",
  "filename" : "output.las"

]```

Note: If the input data doesn’t specify a projection, you must specify the `in_srs` option of `filters.reprojection` (page 222). `in_srs` can also be used to override an existing spatial reference attached to the input point set.

Point Formats

As each revision of LAS was released, more point formats were added. A point format is the fixed set of dimensions (page 287) that a LAS file stores for each point in the file. For any point format, the size and composition of dimensions is consistent across versions, but users should be aware of some minor interpretation changes based on LAS file version. For example, a classification value of 11 in version 1.4 indicates “Road Surface”, while that value is reserved in version 1.1.
Point Format Example

Point format or dataformat_id is an integer that defines the set of fixed dimensions (page 287) stored for each point in a LAS file. All point formats specify the following dimensions as part of a point record:

Table 1: Base LAS Dimensions

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intensity</td>
<td>ReturnNumber</td>
<td>NumberOfReturns</td>
</tr>
<tr>
<td>ScanDirectionFlag</td>
<td>EdgeOfFlightLine</td>
<td>Classification</td>
</tr>
<tr>
<td>ScanAngleRank</td>
<td>UserData</td>
<td>PointSourceId</td>
</tr>
</tbody>
</table>

Because LAS files have no built-in compression, it’s important to use a point format that stores the fewest fields possible that store the desired data. For example, point format 10 uses 45 more bytes per point than point format zero.

If one wanted remove the Red/Green/Blue fields from a LAS file (one using point format 2), one could simply set the dataformat_id option to 0. The forward option can also be set to carry forward all possible header values from the source file to the new, smaller file.

```json
[
  {
    "type": "readers.las",
    "filename": "input.las"
  },
  {
    "type": "writers.las",
    "forward": "all",
    "dataformat_id": 0,
    "filename": "output.las"
  }
]
```

Note: The LASzip (http://laszip.org) storage of GPSTime and Red/Green/Blue fields with no data is perfectly efficient.
Extra Dimensions

A LAS Point Format ID defines the fixed set of dimensions (page 287) a file must store, but programs are allowed to store extra data beyond that fixed set. This feature of the format was regularized in LAS 1.4 as something called “extra bytes” or “extra dims”, but previous versions can also store these extra per-point attributes.

Extra Dimension Example

LAS 1.4 provides for the storage of dimensions not part of the chosen point format by appending them to each point record. PDAL supports this feature when writing files with the “extra_dims” option. The following example will store all source dimensions in the output file and place a description of the dimensions that aren’t part of the point format in an “extra bytes” VLR:

```json
[
    "some_non_las_file",
    {
        "type" : "writers.las",
        "extra_dims": "all",
        "minor_version" : "4",
        "filename" : "output.las"
    }
]
```

Required Header Fields

Readers of the ASPRS LAS Specification will see there are many fields that softwares are required to write, with their content mandated by various options and configurations in the format. PDAL does not assume responsibility for writing these fields and coercing meaning from the content to fit the specification. It is the PDAL users’ responsibility to do so. Fields where this might matter include:

- `project_id`
- `global_encoding`
- `system_id`
- `software_id`
- `filesource_id`
Header Fields Example

The “forward” option of *writers.las* (page 125) is the easiest way to get most of what you might want in terms of header settings copied from an input to an output file upon processing. Imagine the scenario of zero’ing out the classification values for an LAS file in preparation for using *filters.pmf* (page 152) to reassign them. During this scenario, we’d like to keep all of the other LAS header information, such as *Variable Length Records* (page 325), extent information, and format settings.

```json
[
  {
    "type": "readers.las",
    "filename": "input.las"
  },
  {
    "type": "filters.assign",
    "assignment": "Classification[0:32]=0"
  },
  {
    "type": "filters.pmf",
    "cell_size": 2.5,
    "approximate": false,
    "max_distance": 25
  },
  {
    "type": "writers.las",
    "forward": "all",
    "filename": "output.las"
  }
]
```

**Note:** If multiple input LAS files are being written to an output file, the forward option can only preserve values when they are the same in all input files. If the values differ, a default will be used (as it would if the forward option weren’t supplied). You can specify specific option values for output that will also override any forwarded data.
Coordinate Scaling

LAS stores coordinates as 32 bit integers. It is the user’s responsibility to ensure that the coordinate domain required by the data in the file fits within the 32 bit integer domain. Most coordinate values have digits to the right of the decimal point that must be preserved for sufficient accuracy. Using the scale factor allows for integers to be interpreted as floating point values when read by software.

When writing data to LAS, choosing an appropriate scale factor should take into account not just the maximum precision that can be accommodated by the format, but the actual precision of the data. Using a precision greater than the resolution of the data collection can mislead users as to the actual measurement precision of the data. In addition, it can lead to larger files when writing compressed data with LASzip (http://laszip.org).

Auto Offset Example

Users can allow PDAL select scale and offset values for data with the auto option. This can have some detrimental effects on downstream processing. auto for scale values will use the entire 32-bit integer domain. This maximizes the precision available to store the data, but this will have a detrimental effect on LASzip (http://laszip.org) storage efficiency. auto for offset calculation is just fine, however. When given the option, choose to store ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) data with an explicit scale for the X, Y, and Z dimensions that represents actual expected data precision, not artificial storage precision or maximal storage precision.

```
{
    "type": "readers.las",
    "filename": "input.las"
},
{
    "type": "writers.las",
    "scale_x": "0.0000001",
    "scale_y": "0.0000001",
    "scale_z": "0.01",
    "offset_x": "auto",
    "offset_y": "auto",
    "offset_z": "auto",
    "filename": "output.las"
}
```
Compression

LASzip (http://laszip.org) is an open source, lossless compression technique for ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) data. It is supported by two different software libraries, and it can be used in both the C/C++ and the JavaScript execution environments. LAZ support is provided by both readers.las (page 71) and writers.las (page 125). It can be enabled by setting the compression option to laszip.

Compression Example

Providing a filename with a .laz extension will write compressed data. Compression can be turned on explicitly as well:

```json
[
  {
    "type" : "readers.las",
    "filename" : "input.las"
  },
  {
    "type" : "writers.las",
    "compression" : "laszip",
    "filename" : "output.laz"
  }
]
```

Variable Length Records

Variable Length Records, or VLRs, are binary data that the LAS format supports to allow applications to store their own data. Coordinate system information is one type of data stored in VLRs, and many different LAS-using applications store data and metadata with this format capability. PDAL allows users to access VLR information, forward it along to newly written files, and create VLRs that store processing history information.

Common VLR data include:

- Coordinate system
- Metadata
- Processing history
- Indexing

Note: There are VLRs that are defined by the specification, and they have the VLR user_id
of LASF_Spec or LASF_Projection. LASF_Spec VLRs provide a description of the data beyond that available in the header. LASF_Projection VLRs store the spatial coordinate system of the data.

For LAS 1.0-1.3, the VLR length could be no larger than 65535 bytes. Version 1.4 introduced extended VLRs, stored at the end of the file, which could be up to 4gb in size.

**VLR Example**

You can add your own VLRs to files to store processing information or whatever you want by providing a JSON block via writers.las (page 125) vlrs option that defines the user_id and data items for the VLR. The data option must be base64 (https://en.wikipedia.org/wiki/Base64)-encoded string output. The data will be converted to binary information and stored in the VLR when the file is written.

```
[
   "input.las",
   {
      "type": "writers.las",
      "filename": "output.las",
      "vlrs": [
         {
            "description": "A description under 32 bytes",
            "record_id": 42,
            "user_id": "hobu",
            "data": "dGhpcyBpcyBzb21lIHRleHQ="
         },
         {
            "description": "A description under 32 bytes",
            "record_id": 43,
            "user_id": "hobu",
            "data": "dGhpcyBpcyBzb21lIG1vcmUgdGV4dA=="
         }
      ]
   }
]
```
PDAL Metadata

The *writers.las* (page 125) driver supports an option, `pdal_metadata`, that writes two PDAL VLRs to LAS files. The first is the equivalent of *info* (page 29)’s `--metadata` output. The second is a copy of the output of the `--pipeline` serialization option that describes all stages and options of the pipeline that created the file. These two VLRs may be useful in tracking down processing history of data, allow you to determine which versions of PDAL may have written a file and what filter options were set when it was written, and give you the ability to store metadata and other information via pipeline `user_data` from your own applications.

Metadata Example

The pipeline used to construct the file and all of its *Metadata* (page 467) can be written into VLRs in ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) files under the *PDAL VLR key* (http://www.asprs.org/misc/las-key-list.html).

```json
[,
  {
    "type" : "readers.las",
    "filename" : "input.las"
  },
  {
    "type" : "writers.las",
    "pdal_metadata" : "true",
    "filename" : "output.laz"
  }
]
```

**Warning:** LAS versions prior to 1.4 only support VLRs of at most 64K of information. It is possible, though improbable, that the metadata or pipeline stored in the VLRs will not fit in that space.

12.1.4 Clipping with Geometries

**Author**  Howard Butler

**Contact**  howard@hobu.co

**Date**  11/09/2015
Introduction

This tutorial describes how to construct a pipeline that takes in geometries and clips out data with given geometry attributes. It is common to desire to cut or clip point cloud data with 2D geometries, often from auxiliary data sources such as OGR (http://www.gdal.org)-readable Shapefiles (https://en.wikipedia.org/wiki/Shapefile). This tutorial describes how to construct a pipeline that takes in geometries and clips out point cloud data inside geometries with matching attributes.

Contents

- Clipping with Geometries (page 327)
  - Introduction (page 328)
  - Example Data (page 328)
  - Stage Operations (page 328)
  - Data Preparation (page 329)
  - Pipeline (page 330)
  - Processing (page 331)
  - Conclusion (page 332)

Example Data

This tutorial utilizes the Autzen dataset. In addition to typical PDAL software (fetch it from Download (page 13)), you will need to download the following two files:

- https://github.com/PDAL/data/autzen/autzen.laz

Stage Operations

This operation depends on two stages PDAL provides. The first is the filters.overlay (page 206) stage, which allows you to assign point values based on polygons read from OGR (http://www.gdal.org). The second is filters.range (page 248), which allows you to keep or reject points from the set that match given criteria.

See also:

filters.python (page 276) allow you to construct sophisticated logic for keeping or rejecting points in a more expressive environment.
Data Preparation

Fig. 10: Autzen Stadium, a 100 million+ point cloud file.

The data are mixed in two different coordinate systems. The LAZ (page 71) file is in Oregon State Plane Ft. (http://www.oregon.gov/DAS/CIO/GEO/pages/coordination/projections/projections.aspx) and the GeoJSON (http://geojson.org) defining the polygons is in EPSG:4326 (http://epsg.io/4326). We have two options – project the point cloud into the coordinate system of the attribute polygons, or project the attribute polygons into the coordinate system of the points. The latter is preferable in this case because it will be less math and therefore less computation. To make it convenient, we can utilize OGR (http://www.gdal.org)’s VRT (http://www.gdal.org/drv_vrt.html) capability to reproject the data for us on-the-fly:

```
<OGRVRTDataSource>
  <OGRVRTWarpedLayer>
    <OGRVRTLayer name="OGRGeoJSON">
      <SrcDataSource>attributes.json</SrcDataSource>
      <LayerSRS>EPSG:4326</LayerSRS>
    </OGRVRTLayer>
    <TargetSRS>+proj=lcc +lat_1=43 +lat_2=45.5 +lat_0=41.75 +lon_0=-120.5 +x_0=399999.9999999999 +y_0=0 +ellps=GRS80 +units=ft +no_defs</TargetSRS>
  </OGRVRTWarpedLayer>
</OGRVRTDataSource>
```

Note: The GeoJSON file does not have an externally-defined coordinate system, so we are explicitly setting one with the LayerSRS parameter. If your data does have coordinate system information, you don’t need to do that.
Save this VRT definition to a file, called `attributes.vrt` in the same location where you stored the `autzen.laz` and `attributes.json` files.

The attribute GeoJSON file has a couple of features with different attributes. For our scenario, we want to clip out the yellow-green polygon, marked number “5”, in the upper right hand corner.

![Image of a polygon in the upper right hand corner.](image)

**Fig. 11:** We want to clip out the polygon in the upper right hand corner. We can view the GeoJSON (http://geojson.org) geometry using something like QGIS (http://qgis.org)

### Pipeline

A PDAL *pipeline* (page 45) is how you define a set of actions to apply to data as they are read, filtered, and written.

```json
[
    "autzen.laz",
    {
        "type" : "filters.overlay",
        "dimension" : "Classification",
        "datasource" : "attributes.vrt",
        "layer" : "OGRGeoJSON",
        "column" : "CLS"
    },
    {
        "type" : "filters.range",
        "limits" : "Classification[5:5]"
    },
    "output.las"
]```
• **readers.las** (page 71): Define a reader that can read **ASPRS LAS** (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) or **LASzip** (http://laszip.org) data.

• **filters.overlay** (page 206): Using the VRT we defined in **Data Preparation** (page 329), read attribute polygons out of the data source and assign the values from the **CLS** column to the **Classification** field.

• **filters.range** (page 248): Given that we have set the **Classification** values for the points that have coincident polygons to 2, 5, and 6, only keep **Classification** values in the range of 5:5. This functionally means we’re only keeping those points with a classification value of 5.


---

**Note:** You don’t have to use only **Classification** to set the attributes with **filters.overlay** (page 206). Any valid dimension name could work, but most LiDAR softwares will display categorical coloring for the **Classification** field, and we can leverage that behavior in this scenario.

---

**Processing**

1) Save the pipeline to a file called **shape-clip.json** in the same directory as your **attributes.json** and **autzen.laz** files.

2) Run **pdal pipeline** on the json file.

```bash
$ pdal pipeline shape-clip.json
```

3) Visualize **output.las** in an environment capable of viewing it. [http://plas.io](http://plas.io) or [CloudCompare](http://www.danielgm.net/cc/) should do the trick.
Conclusion

PDAL allows the composition of point cloud operations. This tutorial demonstrated how to use the filters.overlay (page 206) and filters.range (page 248) stages to clip points with shapefiles.

12.1.5 Ground Filter Tutorial

**Author**  Bradley Chambers  
**Contact**  brad.chambers@gmail.com  
**Date**  04/17/2017

**Background**

In previous tutorials we introduced our implementation of the **Progressive Morphological Filter (PMF)** (page 152), a **ground kernel** (page 27) to simplify command-line access to PMF, and a filter for **removing outliers** (page 159).

This tutorial will highlight some recent enhancements to the PDAL library, in the context of a ground segmentation workflow. Specifically, we will discuss:

- Constructing and executing a “filters-only” pipeline
- Resetting existing classifications prior to processing
- Using Extended Local Minimum (ELM) to identify low outliers
- Using Simple Morphological Filter (SMRF) as an alternative to PMF
• Ignoring outliers during ground segmentation
• Considering only last returns during ground segmentation
• Extracting ground returns as a post-processing step

Note: The pipeline discussed in this tutorial requires PDAL v1.5 (https://github.com/PDAL/PDAL/releases/tag/1.5.0).

The Pipeline

Begin by creating a new file called pipeline.json with the following contents.

```json
{
    "pipeline": [
        {
            "type": "filters.reprojection",
            "out_srs": "EPSG:32632"
        },
        {
            "type": "filters.assign",
            "assignment": "Classification[:]=0"
        },
        {
            "type": "filters.elm"
        },
        {
            "type": "filters.outlier"
        },
        {
            "type": "filters.smrf",
            "last": true,
            "ignore": "Classification[7:7]",
            "slope": 0.2,
            "window": 16,
            "threshold": 0.45,
            "scalar": 1.2
        },
        {
            "type": "filters.range",
            "limits": "Classification[2:2]"
        }
    ]
}
```
Note: For users familiar with PDAL pipelines, this example may seem to be missing a couple of very important stages, namely the reader and writer! A new feature of PDAL is the ability to provide a PDAL pipeline with no reader or writer stages to the translate (page 39) command. The input and output filenames can be specified on the command line and will be automatically inserted into the pipeline by the application.

The Explanation

We continue by explaining the various stages of the pipeline in order.

Reprojecting Data

Many of PDAL’s default parameters are specified in meters, and individual filter stages typically assume that units are at least uniform in X, Y, and Z. Because data will not always be provided in this way, PDAL pipelines should account for any data reprojections and parameter scaling that are required from one dataset to the next.

```json
{
    "type": "filters.reprojection",
    "out_srs": "EPSG:32632"
}
```

In this example, we show data being reprojected to EPSG:32632 with X, Y, and Z in meters.

Assigning Classification Values

Let’s assume that you have been given an LAS file that contains per point classifications, but you’d like to start with a clean slate and derive your own classifications with your PDAL pipeline.

PDAL’s assign filter (page 204) has been added to assign values to a given dimension. In our example, a single option has been provided that specifies the dimension, range, and value to assign. In this case, we are stating that we would like to apply a value of 0 to the Classification dimension for every point.

```json
{
    "type": "filters.assign",
    "assignment": "Classification[:]=0"
}
```
Extended Local Minimum

The Extended Local Minimum (ELM) method (page 157) helps to identify low noise points that can adversely affect ground segmentation algorithms. ELM was first published in [Chen2012] as part of the upward-fusion method of DTM generation. Noise points are classified with a value of 7 in keeping with the LAS specification.

```json
  "type":"filters.elm"
},
```

Outliers

PDAL’s outlier filter (page 159) provides two methods of outlier detection at the moment: radius and statistical. Both aim to identify points that are isolated and likely arise from noise sources. Noise points are classified with a value of 7 in keeping with the LAS specification.

```json
  "type":"filters.outlier"
},
```

Ground Segmentation

The Simple Morphological Filter (SMRF) (page 155) [Pingel2013] is a newer addition to PDAL that has quietly existed in an alpha state since v1.3. With the release of PDAL v1.5, our SMRF implementation is much more complete, although it only implements nearest neighbor void filling and not the authors’ preferred “Springs” algorithm.

The changes to SMRF between PDAL v1.3 and v1.5 are substantial. The original version had actually drifted quite far from the authors’ published approach, namely in the area of filling voids. We have reverted the code to match the published work, but for now are only using the nearest neighbors approach to filling voids. The morphological operations are also accelerated by moving to an iterative approach and using a diamond structuring element.
In addition to specifying some of the SMRF-specific arguments, our example also demonstrates the use of two optional pre-filtering capabilities: ignore and last.

The ignore option accepts a range (page 249), here indicating that we have points marked as noise (i.e., Classification of 7) that should be excluded from ground segmentation, but are kept as part of the output dataset.

The last option, when set to true indicates that we would like to only consider last returns for ground segmentation when return information is available. Again, returns that are not “last returns” are still retained in the output dataset - they are simply ignored for the purposes of ground segmentation.

Note: Many lidar systems provide return information. This includes the number of returns per pulse and the order of a particular return within the pulse. Where the return number and number of returns are equal, we call this a last return.

Last returns are not by definition ground returns. In fact, the first and only return from surfaces such as rooftops will also be last returns, and last returns within dense foliage may not ever make it all the way to ground. Still, whenever there are multiple returns within a pulse, it stands to reason that anything before the last return would not be from the ground.

Some bare earth algorithms explicitly operate on last returns only. In this case, this logic will presumably be implemented within the filter stage itself. That being said, it stands to reason that any ground segmentation approach could be improved by excluding all returns but the so-called last returns. Neither PMF nor SMRF make this assertion, but our implementations still consider only last returns by default. This behavior can be changed by setting last=false.

For an example of how to filter on last returns outside the context of SMRF and PMF, see this (https://github.com/PDAL/PDAL/blob/master/test/data/pipeline/predicate-keep-last-return.json.in) within PDAL’s source tree.

Note: SMRF is not intended to be a replacement for the Progressive Morphological Filter (PMF) (page 152) [Zhang2003]. Rather, it is offered as an alternative. PMF has been a part of
PDAL since v1.0, first as part of the PCL plugin and now as filters.pmf. Since PDAL v1.4, we have fixed a number of bugs, and have accelerated the approximate mode by implementing iterative morphological operations and using a diamond structuring element.

### Extracting Ground Returns

Any time we have points classified as ground, we may wish to extract just these points, e.g., to create a digital terrain model (DTM). In this case, we use a range filter (page 248) as shown.

```json
{
    "type": "filters.range",
    "limits": "Classification[2:2]"
}
```

The range filter (page 248) accepts a limits option that identifies the dimension(s) on which to filter and the range (page 249) of values to passthrough. In this case, we are indicating that the filter should only pass points whose Classification value is equal to 2.

**Note:** The default behavior of both PMF (page 152) and SMRF (page 155) is to classify points, which has not changed from previous versions of PDAL. The extract and classify options have been removed in PDAL v1.5. These filters now only classify points, such that ground points can be identified and filtered downstream, as we have shown with the range filter above.

### Running the Pipeline

Now let’s run our pipeline.json example, using it to translate (page 39) input.las to output.las.

```
$ pdal translate input.las output.las --json pipeline.json
```

### 12.1.6 Applying a grid shift to point clouds

#### Introduction

This tutorial first appeared on Land Information New Zealand’s On-Location Blog (https://medium.com/on-location).

It describes how to use Conda (https://conda.io), PDAL (https://pdal.io), and GDAL (https://gdal.org) to apply a grid shift to point cloud files. It uses PDAL’s readers.las
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

(https://pdal.io/stages/readers.las.html) to fetch the data, filters.reprojection
(https://pdal.io/stages/filters.reprojection.html) to apply the grid shift, and writers.las
(https://pdal.io/stages/writers.las.html) to write the reprojected point cloud.

The data used in this tutorial is available for free under a CC-BY 4.0 license on Land Information New Zealand’s LINZ Data Service (https://data.linz.govt.nz).

The tutorial will be reprojecting point cloud files from:

- A New Zealand local vertical datum to the New Zealand Vertical Datum 2016 (NZVD2016).
- Finally, NZVD2016 to NZGD2000 or to a local vertical datum.

**Background**

Historically in New Zealand, heights were defined in terms of 13 local vertical datums (LVD) referenced to an estimate of the local mean sea level (MSL).

In 2016, New Zealand Vertical Datum 2016 (NZVD2016), which is defined by the NZGeoid2016 geoid, became the official national vertical datum for New Zealand. The general relationship between the different datums is shown in the diagram below.


The NZ Quasigeoid 2016 (https://data.linz.govt.nz/layer/53447-nz-quasigeoid-2016-raster/), also a relationship grid, models the difference between the NZGD2000 ellipsoid and NZVD2016 (N in the above diagram).

The equations to transform heights using the published values in the relationship grids are:

**Before we begin**

We will be using multiple tools to perform the reprojection. To retrieve these tools and have them all accessible in a nice self-contained environment we will be using a system called Conda. Conda is an open source package and environment management system that runs on Windows, macOS, and Linux. Essentially we will create an environment within Conda which will contain the packages we need: PDAL, GDAL and Python.
LVD to NZVD2016

\[ H_{NZVD2016} = H_A - O \]

For the 13 local vertical datums in New Zealand the published value (O) will always be a negative number. Which means the NZVD2016 height will always be less than the LVD height.

NZVD2016 to NZGD2000

\[ h_{NZGD2000} = H + N \]
Install Conda

Download Miniconda (https://docs.conda.io/en/latest/miniconda.html), selecting the 64-bit installer for your platform and install it as directed.

Create a Conda Environment

1. After installing, open the Anaconda Prompt from your start menu.

2. When you begin using conda, you already have a default environment named base. We don’t want to put programs into our base environment so we’ll create a separate environment just for doing this reprojection. To do this, type:

   ```bash
   conda create --name vd-reproject
   ```

3. It will check for the additional packages/dependencies that are needed, and will ask if you want to proceed. Say yes.

   ```bash
   Proceed ([y]/n)? y
   ```

4. To start to use the new environment and install our required packages within it, we need to activate the environment first:

   ```bash
   conda activate vd-reproject
   ```

   **Note:** After the environment is activated the name of the environment appears as (vd-reproject) at the beginning of the command line. This indicates that you’re now inside the environment.

5. Finally, we need to install the tools/packages we will be using.

   ```bash
   conda install -c conda-forge pdal gdal
   ````

   When these packages are installed, they will also install the packages they’re dependent on to run. Python is one of these dependent packages, so we won’t need to install it ourselves as conda would’ve already done it for us.

   Now that the packages are installed, we are ready to begin.
Step 1: Create a Datum Transformation Grid (GTX)

PDAL allows for the use of PROJ.4 strings to define the spatial reference system of the inputted or outputted data. This is great, because it gives us the ability to use `+geoidgrid` which is an option to add a grid shift file in the format of NOAA Vdatum’s GTX file format. But where to we get a GTX file from? We have two options:

Option 1 — LINZ supplied GTX file

LINZ has created GTX files for each of the relationship grids mentioned earlier. They can be downloaded from [https://www.geodesy.linz.govt.nz/download/proj-datumgrid-nz](https://www.geodesy.linz.govt.nz/download/proj-datumgrid-nz)

Here is a list of which GTX file belongs to which Local Vertical Datum:

- **Auckland 1946**: auckht1946-nzvd2016.gtx
- **Bluff 1955**: blufht1955-nzvd2016.gtx
- **Dunedin 1958**: duneht1958-nzvd2016.gtx
- **Dunedin-Bluff 1960**: dublht1960-nzvd2016.gtx
- **Gisborne 1926**: gisbht1926-nzvd2016.gtx
- **Lyttelton 1937**: lyttht1937-nzvd2016.gtx
- **Moturiki 1953**: motuht1953-nzvd2016.gtx
- **Napier 1962**: napiht1962-nzvd2016.gtx
- **Nelson 1955**: nelsht1955-nzvd2016.gtx
- **One Tree Point 1964**: ontpht1964-nzvd2016.gtx
- **Stewart Island 1977**: stisht1977-nzvd2016.gtx
- **Taranaki 1970**: taraht1970-nzvd2016.gtx
- **Wellington 1953**: wellht1953-nzvd2016.gtx

There is also a GTX file for the Quasigeoid which would be used if converting between NZVD2016 and the NZGD2000 ellipsoid.

- **New Zealand Quasigeoid 2016**: nzgeoid2016.gtx
Option two — Create a GTX file

You can create your own GTX file using the relationship grids available on the LDS. For example, if you intend to convert from Moturiki 1953 to NZVD2016, you have to do the following:


2. Open the Anaconda Prompt from the start menu and activate the environment we created earlier:

   conda activate vd-reproject

3. Navigate to the location of the downloaded TIFF file and execute gdal_translate to convert the TIFF file to a GTX file:

   cd path/to/TIFF/file
   gdal_translate -ot Float32 "moturiki-1953-to-nzvd2016-conversion-raster.tif" "moturiki-1953-to-nzvd2016-conversion-raster.gtx"

   **Note:** -ot Float32 indicates the data type of the output image’s bands. GTX files
only support Float32.

Step 2: Prepare a JSON Pipeline file

We will be using a PDAL pipeline (https://pdal.io/pipeline.html) to transmit a chain of processing elements into PDAL. These elements will be represented in a JSON file.

Using a text editor, create a JSON file named pipeline.json containing the contents as below.

```
{
    "pipeline":
    [
        {
            "type": "readers.las",
            "filename": "#"
        },
        {
            "type": "filters.reprojection",
            "in_srs": "EPSG:2193",
            "out_srs": "EPSG:2193"
        },
        {
            "type": "writers.las",
            "filename": "#",
            "a_srs": "EPSG:2193",
            "forward": "all"
        }
    ]
}
```

Update the srs details for `in_srs`, `out_srs` and `a_srs` to the EPSG code of the horizontal map projection your source LAS files are in. In the example above we are using New Zealand Transverse Mercator 2000 (EPSG:2193).

**Warning:** Be aware "forward": "all" under the writers.las section represents the header fields whose values should be preserved from the source LAS file. `all` will transfer all header fields, including scale and offset values, as well as VLRs. If you desire to transfer only specific header fields, refer to https://pdal.io/stages/writers.las.html for more information about this option.
Step 3: Use PDAL to reproject

Reprojecting one file from LVD to NZVD2016

Using the Anaconda Prompt, activate the vd-reproject environment:

```bash
conda activate vd-reproject
```

Then issue the following command to reproject one file (of course, replace the files and paths to suit your needs).

```bash
pdal pipeline "path/to/your/pipeline.json" -- readers.las.filename="path/to/source_las_file.las" -- writers.las.filename="path/to/reprojected_las_file.las" -- filters.reprojection.out_srs="+init=EPSG:2193 +geoidgrids=path/to/your/gtx_file.gtx"
```

Reprojecting multiple files from LVD to NZVD2016

Below is a python script which executes multiple LAS files. Save to your computer as lvd_to_nzvd2016.py, then open in a text editor and update src_directory, gtxfile, jsonfile, horizontal_srs with the necessary information.

Note: The file is also available from https://gist.github.com/rclarkelinz/d48de5c0432f5c00d02a452e6d1d3bc3

To execute the script, open the Anaconda Prompt, activate the vd-reproject environment and then navigate to where you have saved the script and issue this command:

```bash
python lvd_to_nzvd2016.py
```

This script creates a new directory called ‘reprojected’ in the same location as the LAS files. On completion the reprojected LAS files will be located in this directory, ready for your GIS needs.

You can spot check the accuracy of the conversion by using the LINZ Online converter: www.geodesy.linz.govt.nz/concord
Reprojecting from NZGD2000 to NZVD2016

The steps to do this reprojection are the same as above except for one change:

In **Step 1**, for option one, the GTX file required will be `nzgeoid2016.gtx`. Or, if you are following option two, the relationship grid on the LDS is the [NZ Quasigeoid 2016](https://data.linz.govt.nz/layer/53447-nz-quasigeoid-2016-raster/).

**NZVD2016 to NZGD2000 or LVD**

Previously, the grid values are being *subtracted* from the point cloud value in **Step 3**. To reproject to NZGD2000 or an LVD, the grid values need to be *added* to the NZVD2016 value.

To accommodate this change in PDAL, you need to alter the following text in the PDAL command from `filters.reprojection.out_srs` to `filters.reprojection.in_srs`. 
13.1 Point Cloud Processing and Analysis with PDAL

Author  Howard Butler
Author  Pete Gadomski
Author  Dr. Craig Glennie
Author  Michael Smith
Author  Dr. Adam Steer
Contact  howard@hobu.co
Date  08/26/2019

13.1.1 Introduction

1. *Introduction to LiDAR* (page 351)
2. *Introduction to PDAL* (page 5)
3. *Software Installation* (page 356)
4. *Basic Information* (page 357)
5. *Translation* (page 363)
6. *Analysis* (page 370)
7. *Georeferencing* (page 430)
Materials

Slides

- Slides (https://pdal.s3.amazonaws.com/workshop/slides.zip)

Workshop Materials

These materials are available as a PDF and an HTML website.


USB Example Data Drive

A companion USB drive containing workshop example data is required to follow along with these examples.

Note: A drive image is available for download at
13.1.2 Introduction to LiDAR

LiDAR is a remote sensing technique that uses visible or near-infrared laser energy to measure the distance between a sensor and an object. LiDAR sensors are versatile and (often) mobile; they help autonomous cars avoid obstacles and make detailed topographic measurements from space. Before diving into LiDAR data processing, we will spend a bit of time reviewing some LiDAR fundamentals and discussing some terms of art.

Types of LiDAR

LiDAR systems, generally speaking, come in one of three types:

- **Pulse-based**, or **linear-mode**, systems emit a pulse of laser energy and measure the time it takes for that energy to travel to a target, bounce off the target, and be returned to the sensor. These systems are called linear-mode because they (generally) only have a single aperture, and so can only measure distance along a single vector at any point in time. Pulse-based systems are very common, and are usually what you would think of when you think of LiDAR.

- **Phase-based** LiDAR systems measure distance via *interferometry*, that is, by using the phase of a modulated laser beam to calculate a distance as a fraction of the modulated signal’s wavelength. Phase-based systems can be very precise, on the order of a few millimeters, but since they require comparatively more energy than the other two types they are usually used for short-range (e.g. indoor) scanning.

- **Geiger-mode**, or **photon-counting**, systems use extremely sensitive detectors that can be triggered by a single photon. Since only a single photon is required to trigger a measurement, these systems can operate at much much higher altitudes than linear mode systems. However, Geiger-mode systems are relatively new and suffer from very high amounts of noise and other operational restrictions, making them significantly less common than linear-mode systems.

**Note:** Unless otherwise noted, if we talk about a LiDAR scanner in this program, we will be referring to a pulse-based (linear) system.

Modes of LiDAR Collection

LiDAR collects are generally categorized into four subjective types:

- **Terrestrial LiDAR Scanning (TLS)**: scanning with a stationary LiDAR sensor, usually mounted on a tripod.

- **Airborne LiDAR scanning (ALS)**: also called airborne laser swath mapping (ALSM), scanning with a LiDAR scanner mounted to a fixed-wing or rotor aircraft.
Mobile LiDAR scanning (MLS): scanning from a ground-based vehicle, such as a car.

Unmanned LiDAR scanning (ULS): scanning with drones or other unmanned vehicles.

With the exception of stationary TLS, LiDAR scanning generally requires the use of an integrated GNSS/IMU (Global Navigation Satellite System/Inertial Motion Unit), which provides information about the position, rotation, and motion of the scanning platform.

Note: As stated in the class description, we will focus on mobile and airborne laser scanning (MLS/ALS), though we will also use some TLS data.

Georeferencing

LiDAR scanners collect information in the Scanner’s Own Coordinate System (SOCS); this is a coordinate system centered at the scanner. The process of rotating, translating, and (possibly) transforming a point cloud into a real-world spatial reference system is known as georeferencing.

In the case of TLS, georeferencing is simply a matter of discovering the position and orientation of the static scanner. This is usually done with GNSS control points, which are used to solve for the scanner’s position via least-squares.

For mobile or airborne LiDAR scanning, it is necessary to merge the scanner’s points with the GNSS/IMU data. This can be done on-the-fly or as a part of a post-processing workflow. Since this is a common operation for mobile and airborne LiDAR collects, we will spend a moment discussing the methods and complications for georeferencing mobile LiDAR and GNSS/IMU data.

Integrating LiDAR and GNSS/IMU data

The LiDAR georeferencing equation is well-established; we present a version here from \([\text{Gle07]}\):

\[
p_G^l = p_{\text{GPS}}^l + R_6^l (R_s^b r_s^b - l^b) \tag{13.1}
\]

where:

- \(p_G^l\) are the coordinates of the target point in the global reference frame
- \(p_{\text{GPS}}^l\) are the coordinates of the GNSS sensor in the global reference frame
- \(R_6^l\) is the rotation matrix from the navigation frame to the global reference frame
- \(R_s^b\) is the rotation matrix from the scanner’s frame to the navigation frame (boresight matrix)
• $r^s$ is the coordinates of the laser point in the scanner’s frame
• $l^b$ is the lever-arm offset between the scanner’s original and the navigation’s origin

This equation contains fourteen unknowns, and in order to georeference a single LiDAR return we must determine all fourteen variables at the time of the pulse.

As a rule of thumb, the position, attitude, and motion of the scanning platform (aircraft, vehicle, etc) are sampled at a much lower rate than the pulse rate of the laser — rates of ~1Hz are common for GNSS/IMU sampling. In order to match the GNSS/IMU sampling rate with the sampling rate of the laser, GNSS/IMU measurements are interpolated to line up with the LiDAR measurements. Then, these positions and attitudes are combined via Equation (13.1) to create a final, georeferenced point cloud.

**Note:** While lever-arm offsets are usually taken from the schematic drawings of the LiDAR mounting system, the boresight matrix cannot be reliably determined from drawings alone. The boresight matrix must therefore be determined either via manual or automated boresight calibration using actual LiDAR data of planar surfaces, such as the roof and sides of buildings. The process for determining a boresight calibration from LiDAR data is beyond the scope of this class.

**Discrete-Return vs. Full-Waveform**

Pulse-based LiDAR systems use the round-trip travel time of a pulse of laser energy to measure distances. The outgoing pulse of a LiDAR system is roughly (but not exactly) a Gaussian:

This pulse can interact with multiple objects in a scene before it is returned to the sensor. Here is an example of a LiDAR return:

As you can see, this return pulse can be very complicated. While there is more information contained in the “full waveform” picture displayed above, many LiDAR consumers are only interested in detecting the presence or absence of an object — simplistically, the peaks in that waveform.

Full waveform data is used only in specialized circumstances. If you have or receive LiDAR data, it will usually be discrete return (point clouds). Processing full waveform data is beyond the scope of this class.

**Note:** PDAL is a discrete-return point cloud processing library. It does not have any functionality to analyze or process full waveform data.
Fig. 1: A real-world outgoing LiDAR pulse.
Fig. 2: A real-world incoming LiDAR return. Potential discrete-return peaks are marked in red.
13.1.3 Software Installation

Conda

What is Conda

Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer. It was created for Python programs, but it can package and distribute software for any language.

How will we use Conda?

PDAL stands on the shoulders of giants. It uses GDAL, GEOS, and many other dependencies (page 456). Because of this, it is very challenging to build it yourself. We could easily burn an entire workshop learning the esoteric build mysteries of PDAL and all of its dependencies. Fortunately, Conda provides us a fully-featured known configuration to run our examples and exercises without having to suffer so much, and provides it for Windows, Linux, and macOS.

Note: Not everyone uses Conda. Another alternative to get a known configuration is to go through the workshop using docker as your platform. A previous edition of the workshop was provided as Docker, but it was found to be a bit too difficult to follow.

Installing Conda

1. Copy the entire contents of your workshop USB key to a PDAL directory in your home directory (something like C:\Users\hobu\PDAL) or the equivalent for your OS. We will refer to this location for the rest of the workshop materials.

2. Download the Conda installer for your OS setup.
   https://docs.conda.io/en/latest/miniconda.html

3. After installing Conda, create an environment for PDAL with:
   ```bash
   conda create --name pdalworkshop
   ```

4. Then activate the new environment:
   ```bash
   conda activate pdalworkshop
   ```

5. Install PDAL, Entwine, and GDAL, and install it from conda-forge:
13.1.4 Exercises

Basic Information

Printing the first point

Exercise

This exercise uses PDAL to print information from the first point. Issue the following command in your Conda Shell.

```
pdal info ./exercises/info/interesting.las -p 0
```

Here’s a summary of what’s going on with that command invocation

1. `pdal`: The `pdal` application :)  
2. `info`: We want to run `info` (page 29) on the data. All commands are run by the `pdal` application.  
3. `./exercises/info/interesting.las`: The file we are running the command on. PDAL will be able to identify this file is an ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) file from the extension, `.las`, but not every file type is easily identified. You can use a `pipeline` (page 32) to override which `reader` (page 54) PDAL will use to open the file.  
4. `-p 0`: `-p` corresponds to “print a point”, and `0` means to print the first one (computer people count from 0).
Notes


2. You can use the writers.text (page 145) writer to output point attributes to CSV (https://en.wikipedia.org/wiki/Comma-separated_values) format for other processing.

3. Output help information on the command line by issuing the --help option

4. A common query with pdal info is --all, which will print all header, metadata, and statistics about a file.

Printing file metadata

Exercise

This exercise uses PDAL to print metadata information. Issue the following command in your Conda Shell.

```
pdal info ./exercises/info/interesting.las --metadata
```
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

Note: PDAL metadata (page 467) is returned in a tree structure corresponding to processing pipeline that produced it.

See also:

Use the JSON (https://en.wikipedia.org/wiki/JSON) processing capabilities of your favorite processing software to selectively access and manipulate values.

- Python JSON library (https://docs.python.org/2/library/json.html)
- jsawk (https://github.com/micha/jsawk) (like awk but for JSON data)
- jq (https://stedolan.github.io/jq/) (command line processor for JSON)
- Ruby JSON library (http://ruby-doc.org/stdlib-2.0.0/libdoc/json/rdoc/JSON.html)

Structured Metadata Output

Many command-line utilities output their data in a human-readable custom format. The downsides to this approach are significant. PDAL was designed to be used in the context of other software tools driving it. For example, it is quite common for PDAL to be used in data validation scenarios. Other programs might need to inspect information in PDAL’s output and then act based on the values. A human-readable format would mean that downstream program would need to write a parser to consume PDAL’s special format.

JSON (https://en.wikipedia.org/wiki/JSON) provides a nice balance between human- and machine-readable, but even then it can be quite hard to find what you’re looking for, especially
if the output is long, pdal command output used in conjunction with a JSON parsing tool like jq provide a powerful inspection combination.

For example, we might only care about the system_id and compressed flag for this particular file. Our simple pdal info --metadata command gives us that, but it also gives us a bunch of other stuff we don’t need at the moment either. Let’s focus on extracting what we want using the jq command.

```bash
pdal info ./exercises/info/interesting.las --metadata | jq ".metadata.compressed, .metadata.system_id"
```

**Note:** PDAL’s JSON output is very powerfully combined with the processing capabilities of other programming languages such as JavaScript or Python. Both of these languages have excellent built-in tools for consuming JSON, along with plenty of other features to allow you to do something with the data inside the data structures. As we will see later in the workshop, this PDAL feature is one that makes construction of custom data processing workflows with PDAL very convenient.

**Notes**

1. PDAL uses JSON (https://en.wikipedia.org/wiki/JSON) as the exchange format when printing information from info (page 29). JSON provides human and machine-readable text data.

2. The PDAL metadata document (page 467) contains background and information about specific metadata entries and what they mean.

3. Metadata available for a given file depends on the stage that produces the data. Readers (page 54) produce same-named values where possible, but it is common that variables are different. Filters (page 150) and even writers (page 113) can also produce metadata entries.

4. Spatial reference system or coordinate system information is a kind of special metadata. Spatial references are come directly from source data or are provided via options in PDAL.
Searching near a point

Exercise

This exercise uses PDAL to find points near a given search location. Our scenario is a simple one – we want to find the two points nearest the midpoint of the bounding cube of our `interesting.las` data file.

First we need to find the midpoint of the bounding cube. To do that, we need to print the `--all` info for the file and look for the `bbox` output:

```
pdal info ./exercises/info/interesting.las --all | jq .stats.bbox.
```

Find the average the X, Y, and Z values:

```
x = 635619.85 + (638982.55 - 635619.85)/2 = 637301.20
y = 848899.70 + (853535.43 - 848899.70)/2 = 851217.57
z = 406.59 + (586.38 - 406.59)/2 = 496.49
```

With our “center point”, issue the `--query` option to `pdal info` and return the three nearest points to it:

```
pdal info ./exercises/info/interesting.las --query "637301.20,851217.57,496.49/3"
```

Note: The `/3` portion of our query string tells the `query` command to give us the 3 nearest points. Adjust this value to return data in closest-distance ordering.
Notes


2. The `--query` option of `info` (page 29) constructs a KD-tree (https://en.wikipedia.org/wiki/K-d_tree) of the entire set of points in memory. If you have really large data sets, this isn’t going to work so well, and you will need to come up with a different solution.
Translation

Compression

Exercise

This exercise uses PDAL to compress ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) data into LASzip (http://laszip.org).

1. Issue the following command in your Conda Shell.

```
pdal translate ./exercises/translation/interesting.laz ./exercises/translation/interesting.las
```

```
pdal translate ./exercises/translation/interesting.laz ./exercises/translation/interesting.las
```

LAS is a very fluffy binary format. Because of the way the data are stored, there is ample redundant information, and LASzip (http://laszip.org) is an open source solution for compressing this information. Note that we are actually inflating the data here. Its laz from the workshop and we are converting it to las.

2. Verify that the laz data is compressed over the las:

```
ls -alh ./exercises/translation/interesting.laz
```

```
ls -alh ./exercises/translation/interesting.las
```

```
dir ./exercises/translation/interesting.laz
```

```
dir ./exercises/translation/interesting.las
```
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

See also:

*LAS Reading and Writing with PDAL* (page 316) contains many pointers about settings for ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) data and how to achieve specific data behaviors with PDAL.

Notes

1. Typical LASzip (http://laszip.org) compression is 5:1 to 8:1, depending on the type of LiDAR (https://en.wikipedia.org/wiki/Lidar). It is a compression format specifically for the ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) model, however, and will not be as efficient for other types of point cloud data.

2. You can open and view LAZ data in web browsers using http://plas.io

Reprojection

Exercise

This exercise uses PDAL to reproject ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) data

Issue the following command in your *Conda Shell*:

```bash
pdal translate ./exercises/analysis/ground/CSite1_orig-utm.laz ./exercises/translation/csite-dd.laz reprojection --filters.reprojection.out_srs="EPSG:4326"
```

Unfortunately this doesn’t produce the intended results for us. Issue the following *pdal info* command to see why:
pdal info ./exercises/translation/csite-dd.laz --all \ 
| jq .stats.bbox.native.bbox

pdal info ./exercises/translation/csite-dd.laz --all ^ 
| jq .stats.bbox.native.bbox

```
{ "maxx": 0.18,  
  "maxy": 48.70,  
  "maxz": 426.91,  
  "minx": 9.16,  
  "miny": 48.78,  
  "minz": 99.43 
}
```

--all dumps all info (page 29) information about the file, and we can then use the jq (https://stedolan.github.io/jq/) command to extract out the “native” (same coordinate system as the file itself) bounding box. As we can see, the problem is we only have two decimal places of precision on the bounding box. For geographic coordinate systems, this isn’t enough precision.

Printing the first point confirms this problem:

```
{ "filename": "C:\\Users\\hobu\\PDAL\\exercises\\translation\\csite-dd.laz",  
  "pdal_version": "1.9.1 (git-version: Release)",  
  "points":  
  {  
    "point":  
    {  
      "Blue": 0,  
      "Classification": 0,  
      "EdgeOfFlightline": 0,  
      "GpsTime": 0,  
      "Green": 0,  
      "Intensity": 100,  
      "NumberOfReturns": 2,  
      "PointId": 0,  
      "PointSourceId": 0,  
      "Red": 0,  
      "ReturnNumber": 1,  
      "ScanAngleRank": 0,  
      "ScanDirectionFlag": 0,  
      "UserData": 0,  
      "X": 9.17,  
      "Y": 48.78,  
      "Z": 316.88  
    }  
  }  
}
```

Some formats, like writers.las (page 125) do not automatically set scaling information. PDAL cannot really do this for you because there are a number of ways to trip up. For latitude/longitude data, you will need to set the scale to smaller values like 0.0000001. Additionally, LAS uses an offset value to move the origin of the value. Use PDAL to set that to auto so you don’t have to compute it.
Run the `pdal info` command again to verify the X, Y, and Z dimensions:

```
(pdalworkshop) $pdal info ./exercises/translation/csite-dd.laz --all
> | jq .stats.bbox.native.bbox
{
  "maxx": 9.179302939,
  "maxy": 48.78976523,
  "maxz": 426.91,
  "minx": 9.164037839,
  "miny": 48.78345443,
  "minz": 99.43
}
```

```
(pdalworkshop) $pdal info ^
> ./exercises/analysis/ground/CSite1_orig-utm.laz ^
> ./exercises/translation/csite-dd.laz reprojection ^
> --filters.reprojection.out_srs="EPSG:4326" ^
> --writers.las.scale_x=0.0000001 ^
> --writers.las.scale_y=0.0000001 ^
> --writers.las.offset_x="auto" ^
> --writers.las.offset_y="auto"
```

```
(pdalworkshop) $pdal translate ^
> ./exercises/analysis/ground/CSite1_orig-utm.laz ^
> ./exercises/translation/csite-dd.laz reprojection ^
> --filters.reprojection.out_srs="EPSG:4326" ^
> --writers.las.scale_x=0.0000001 ^
> --writers.las.scale_y=0.0000001 ^
> --writers.las.offset_x="auto" ^
> --writers.las.offset_y="auto"
```
Notes

1. *filters.reprojection* (page 222) will use whatever coordinate system is defined by the point cloud file, but you can override it using the `in_srs` option. This is useful in situations where the coordinate system is not correct, not completely specified, or your system doesn’t have all of the required supporting coordinate system dictionaries.

2. PDAL uses Proj.4 (http://proj4.org) library for reprojection. This library includes the capability to do both vertical and horizontal datum transformations.

Entwine

Exercise

This exercise uses PDAL to fetch data from an Entwine index stored in an Amazon Web Services object store (bucket). Entwine is a point cloud indexing strategy, which rearranges points into a lossless octree structure known as EPT, for Entwine Point Tiles. The structure is described here: https://entwine.io/entwine-point-tile.html.

EPT indexes can be used for visualization as well as analysis and data manipulation at any scale.

Examples of Entwine usage can be found from very fine photogrammetric surveys to continental scale lidar management.

US Geological Survey (USGS) example data is here: https://usgs.entwine.io/

We will use a sample data set from Dublin, Ireland


1. View the `entwine.json` file in your editor. If the file does not exist, create it and paste the following JSON into it:

```json
{
  "pipeline": [
    {
      "type": "readers.ept",
      "filename": "https://na-c.entwine.io/dublin/",
      "resolution": 5
    },
    {
      "type": "writers.las",
      "compression": "true",
      "minor_version": "2",
      "dataformat_id": "0",
      "filename": "dublin.laz"
    }
  ]
}
```

(continues on next page)
Note: If you use the Developer Console (https://developers.google.com/web/tools/chrome-devtools/console/) when visiting http://speck.ly or http://potree.entwine.io, you can see the browser making requests against the EPT resource at http://na-c.entwine.io/dublin/ept.json

2. Issue the following command in your Conda Shell.

```
pdal pipeline ./excercises/translation/entwine.json -v 7
```

3. Verify that the data look ok:

```
pdal info dublin.laz | jq .stats.bbox.native.bbox
```
```
pdal info dublin.laz -p 0
```
4. Visualize the data in http://plas.io
Notes

1. *readers.ept* (page 56) contains more detailed documentation about how to use PDAL's EPT reader.

Analysis

Finding the boundary

This exercise uses PDAL to find a tight-fitting boundary of an aerial scan. Printing the coordinates of the boundary for the file is quite simple using a single `pdal info` call, but visualizing the boundary is more complicated. To complete this exercise, we are going to use qgis to view the boundary, which means we must first install it on our system.

Exercise

Note: We are going to run using the Uncompahgre data in the `./density` directory.

```
pdal info ./exercises/analysis/density/uncompahgre.laz --boundary
```

... a giant blizzard of coordinate output scrolls across our terminal. Not very useful.

Instead, let’s generate some kind of vector output we can visualize with qgis. The `pdal tindex` is the “tile index” command, and it outputs a vector geometry file for each point.
cloud file it reads. It generates this boundary using the same mechanism we invoked above – `filters.hexbin` (page 263). We can leverage this capability to output a contiguous boundary of the `uncompahgre.laz` file.

```
pdal tindex create --tindex ./exercises/analysis/boundary/boundary.
  --filespec ./exercises/analysis/density/uncompahgre.laz \
  -f SQLite
```

Once we’ve run the `tindex` (page 37), we can now visualize our output:

Open qgis and select `Add Vector Layer`:  

13.1. Point Cloud Processing and Analysis with PDAL
Navigate to the exercises/analysis/boundary directory and then open the boundary.sqlite file:
Notes

1. The PDAL boundary computation is an approximation based on a hexagon tessellation. It uses the software at http://github.com/hobu/hexer to do this task.

2. filters.hexbin (page 263) can also be used by the density (page 27) to generate a tessellated surface. See the Visualizing acquisition density (page 387) example for steps to achieve this.

3. The tindex (page 37) can be used to generate boundaries for large collections of data. A boundary-based indexing scheme is commonly used in LiDAR processing, and PDAL supports it through the tindex application. You can also use this command to merge data together (query across boundaries, for example).
Clipping data with polygons

This exercise uses PDAL to apply to clip data with polygon geometries.

Note: This exercise is an adaption of the PDAL tutorial (page 327).

Exercise

The autzen.laz file is a staple in PDAL and libLAS examples. We will use this file to demonstrate clipping points with a geometry. We’re going to clip out the stadium into a new LAS file.

Data preparation

The data are mixed in two different coordinate systems. The LAZ (page 71) file is in Oregon State Plane Ft. (http://www.oregon.gov/DAS/CIO/GEO/pages/coordination/projections/projections.aspx) and the GeoJSON (http://geojson.org) defining the polygons, attributes.json, is in EPSG:4326 (http://epsg.io/4326). We have two options – project the point cloud into the coordinate system of the attribute polygons, or project the attribute polygons into the
coordinate system of the points. The latter is preferable in this case because it will be less math and therefore less computation. To make it convenient, we can utilize OGR (http://www.gdal.org)’s VRT (http://www.gdal.org/drv_vrt.html) capability to reproject the data for us on-the-fly:

```xml
<OGRVRTDataSource>
  <OGRVRTWarpedLayer>
    <OGRVRTLayer name="OGRGeoJSON">
      <SrcDataSource>./exercises/analysis/clipping/attributes.json</SrcDataSource>
      <SrcLayer>attributes</SrcLayer>
      <LayerSRS>EPSG:4326</LayerSRS>
    </OGRVRTLayer>
    <TargetSRS>+proj=lcc +lat_1=43 +lat_2=45.5 +lat_0=41.75 +lon_0=-120.5 +x_0=399999.9999999999 +y_0=0 +ellps=GRS80 +units=ft +no_defs</TargetSRS>
  </OGRVRTWarpedLayer>
</OGRVRTDataSource>
```

Note: This VRT file is available in your workshop materials in the ./exercises/analysis/clipping/attributes.vrt file. You will need to open this file, go to line 4 and replace ./ with the correct path for your machine.

A GDAL or OGR VRT is a kind of “virtual” data source definition type that combines a definition of data and a processing operation into a single, readable data stream.
Note: The GeoJSON file does not have an externally-defined coordinate system, so we are explicitly setting one with the LayerSRS parameter. If your data does have coordinate system information, you don’t need to do that. See the OGR VRT documentation (http://www.gdal.org/drv_vrt.html) for more details.

Pipeline breakdown

```json
{
    "pipeline": [
        "./exercises/analysis/clipping/autzen.laz",
        ">
            "column": "CLS",
            "datasource": "./exercises/analysis/clipping/attributes.vrt",
            "dimension": "Classification",
        
(continues on next page)```
"layer": "OGRGeoJSON",
"type": "filters.overlay"
},
{
"limits": "Classification[6:6]",
"type": "filters.range"
},
"./exercises/analysis/clipping/stadium.las"
}
}

**Note:** This pipeline is available in your workshop materials in the 
./exercises/analysis/clipping/clipping.json file. Remember to replace each of the three occurrences of ./ in this file with the correct location for your machine.

1. **Reader**

autzen.laz is the LASzip (http://laszip.org) file we will clip.

2. **filters.overlay**

The filters.overlay (page 206) filter allows you to assign values for coincident polygons. Using the VRT we defined in Data preparation (page 374), filters.overlay (page 206) will assign the values from the CLS column to the Classification field.
3. filters.range

The attributes in the attributes.json file include polygons with values 2, 5, and 6. We will use filters.range (page 248) to keep points with Classification values in the range of 6:6.

4. Writer

We will write our content back out using a writers.las (page 125).

Execution

Invoke the following command, substituting accordingly, in your Conda Shell:

The –nostream option disables stream mode. The point-in-polygon check (see notes) performs poorly in stream mode currently.

```
pdal pipeline .exeercises/analysis/clipping/clipping.json --nostream
```

Visualization

Use one of the point cloud visualization tools you installed to take a look at your ./exercises/analysis/clipping/stadium.las output. In the example below, we opened the file to view it using the http://plas.io website.
1. `filters.overlay` (page 206) does point-in-polygon checks against every point that is read.
2. Points that are on the boundary are included.
Colorizing points with imagery

This exercise uses PDAL to apply color information from a raster onto point data. Point cloud data, especially LiDAR (https://en.wikipedia.org/wiki/Lidar), do not often have coincident color information. It is possible to project color information onto the points from an imagery source. This makes it convenient to see data in a larger context.

Exercise

PDAL provides a filter (page 150) to apply color information from raster files onto point cloud data. Think of this operation as a top-down projection of RGB color values on the points.

Because this operation is somewhat complex, we are going to use a pipeline to define it.

```json
{
   "pipeline": [
      "./exercises/analysis/colorization/uncompahgre.laz",
      {
         "type": "filters.colorization",
         "raster": "./exercises/analysis/colorization/casi-2015-04-29-weekly-mosaic.tif"
      },
      {
         "type": "filters.range",
         "limits": "Red[1:]"
      },
      {
         "type": "writers.las",
         "compression": "true",
         "minor_version": "2",
         "dataformat_id": "3",
         "filename": "./exercises/analysis/colorization/uncompahgre-colored.laz"
      }
   ]
}
```

Note: This JSON file is available in your workshop materials in the ./exercises/analysis/colorization/colorize.json file. Remember to open this file and replace each occurrence of ./ with the correct path for your machine.
Pipeline breakdown

1. Reader

After our pipeline errata, the first item we define in the pipeline is the point cloud file we’re going to read.

"./exercises/analysis/colorization/uncompahgre.laz",

2. filters.colorization

The filters.colorization (page 178) PDAL filter does most of the work for this operation. We’re going to use the default data scaling options. This filter will create PDAL dimensions Red, Green, and Blue.

```
{
    "type": "filters.colorization",
    "raster": "./exercises/analysis/colorization/casi-2015-04-29-weekly-mosaic.tif"
}
```

3. filters.range

A small challenge is the raster will colorize many points with NODATA values. We are going to use the filters.range (page 248) to filter keep any points that have Red >= 1.

```
{
    "type": "filters.range",
    "limits": "Red[1:]
}
```

4. writers.las

We could just define the uncompahgre-colored.laz filename, but we want to add a few options to have finer control over what is written. These include:

```
{
    "type": "writers.las",
    "compression": "true",
    "minor_version": "2",
    "dataformat_id": "3",
}
```

(continues on next page)
1. compression: LASzip (http://laszip.org) data is ~6x smaller than ASPRS LAS.

2. minor_version: We want to make sure to output LAS 1.2, which will provide the widest compatibility with other softwares that can consume LAS.

3. dataformat_id: Format 3 supports both time and color information

Note: writers.las (page 125) provides a number of possible options to control how your LAS files are written.

**Execution**

Invoke the following command, substituting accordingly, in your Conda Shell:

```
pdal pipeline ./exercises/analysis/colorization/colorize.json
```

**Visualization**

Use one of the point cloud visualization tools you installed to take a look at your uncompahgre-colored.laz output. In the example below, we simply opened the file using the http://plas.io website.
Notes

1. Applying color information that is not time-coincident with the point cloud data will mean you will see discontinuities.

2. GDAL is used to read the image source. Any GDAL-readable data format can be used.

3. There are performance considerations to be aware of depending on the raster format and type being used. See `filters.colorization` (page 178) for more information.

Removing noise

This exercise uses PDAL to remove unwanted noise in an airborne LiDAR collection.

**Exercise**

PDAL provides the *outlier filter* (page 159) to apply a statistical filter to data. Because this operation is somewhat complex, we are going to use a pipeline to define it.

```json
{
   "pipeline": [
      "/exercises/analysis/denoising/18TWK820985.laz",
      {
         "type": "filters.outlier",
         "method": "statistical",
         "multiplier": 3,
         "mean_k": 8
      },
      {
         "type": "filters.range",
         "limits": "Classification!7:7",Z[-100:3000]"
      },
      {
         "type": "writers.las",
         "compression": "true",
         "minor_version": "2",
         "dataformat_id": "0",
         "filename": "/exercises/analysis/denoising/clean.laz"
      }
   ]
}
```

**Note:** This pipeline is available in your workshop materials in the 
./exercises/analysis/denoising/denoise.json file.
Pipeline breakdown

1. Reader

After our pipeline errata, the first item we define in the pipeline is the point cloud file we’re going to read.

```
./exercises/analysis/denoising/18TWK820985.laz
```

2. filters.outlier

The PDAL outlier filter (page 159) does most of the work for this operation.

```
{
   "type": "filters.outlier",
   "method": "statistical",
   "multiplier": 3,
   "mean_k": 8
}
```

3. filters.range

At this point, the outliers have been classified per the LAS specification as low/noise points with a classification value of 7. The range filter (page 248) can remove these noise points by constructing a range (page 249) with the value Classification!\[7:7\], which passes every point with a Classification value not equal to 7.

Even with the filters.outlier (page 159) operation, there is still a cluster of points with extremely negative $Z$ values. These are some artifact or miscomputation of processing, and we don’t want these points. We can construct another range (page 249) to keep only points that are within the range $-100 <= Z <= 3000$.

Both ranges (page 249) are passed as a comma-separated list to the range filter (page 248) via the limits option.

```
{
   "type": "filters.range",
   "limits": "Classification!\[7:7\],Z[-100:3000]"
}
```
4. writers.las

We could just define the `clean.laz` filename, but we want to add a few options to have finer control over what is written. These include:

```json
{
    "type": "writers.las",
    "compression": "true",
    "minor_version": "2",
    "dataformat_id": "0",
    "filename": "./exercises/analysis/denoising/clean.laz"
}
```

1. **compression**: LASzip (http://laszip.org) data is ~6x smaller than ASPRS LAS.
2. **minor_version**: We want to make sure to output LAS 1.2, which will provide the widest compatibility with other softwares that can consume LAS.
3. **dataformat_id**: Format 3 supports both time and color information

**Note**: `writers.las` (page 125) provides a number of possible options to control how your LAS files are written.

## Execution

Invoke the following command, substituting accordingly, in your `Shell`:

```
pdal pipeline ./exercises/analysis/denoising/denoise.json
```

```
(pdalworkshop) $ pdal density ./exercises/analysis/density/uncompahgre.laz \
> -o ./exercises/analysis/density/density.sqlite \
> -f SQLite
(pdalworkshop) $
```
Visualization

Use one of the point cloud visualization tools you installed to take a look at your clean.laz output. In the example below, we simply opened the file using the Fugro Viewer (http://www.fugroviewer.com/)

Notes

1. Control the aggressiveness of the algorithm with the mean_k parameter.

2. filters.outlier (page 159) requires the entire set in memory to process. If you have really large files, you are going to need to split (page 256) them in some way.

Visualizing acquisition density

This exercise uses PDAL to generate a density surface. You can use this surface to summarize acquisition quality.
**Exercise**

PDAL provides an application (page 27) to compute a vector field of hexagons computed with `filters.hexbin` (page 263). It is a kind of simple interpolation, which we will use for visualization in QGIS (http://qgis.org).

**Command**

Invoke the following command, substituting accordingly, in your `Shell`:

```
1  pdal density ./exercises/analysis/density/uncompahgre.laz \\
2  -o ./exercises/analysis/density/density.sqlite \\
3  -f SQLite
```

**Visualization**

The command uses GDAL to output a SQLite (http://sqlite.org) file containing hexagon polygons. We will now use QGIS (http://qgis.org) to visualize them.

1. Add a vector layer
2. Navigate to the output directory

3. Add the `density.sqlite` file to the view
4. Right click on the `density.sqlite` layer in the Layers panel and then choose Properties.

5. Pick the Graduated drop down
6. Choose the **Count** column to visualize

![Image of count column visualization](image1)

7. Choose the **Classify** button to add intervals

![Image of classify button](image2)

8. Adjust the visualization as desired
Notes

1. You can control how the density hexagon surface is created by using the options in `filters.hexbin` (page 263).

   The following settings will use a hexagon edge size of 24 units.
   
   ```
   --filters.hexbin.edge_size=24
   ```

2. You can generate a contiguous boundary using PDAL (https://pdal.io/)’s `tindex` (page 37).

Thinning

This exercise uses PDAL to subsample or thin point cloud data. This might be done to accelerate processing (less data), normalize point density, or ease visualization.
Exercise

As we showed in the Visualizing acquisition density (page 387) exercise, the points in the uncompahgre.laz file are not evenly distributed across the entire collection. While we will not get into reasons why that particular property is good or bad, we note there are three different sampling strategies we could choose. We can attempt to preserve shape, we can try to randomly sample, and we can attempt to normalize posting density. PDAL provides capability for all three:

- Poisson using the filters.sample (page 233)
- Random using a combination of filters.decimation (page 231) and filters.randomize (page 212)
- Voxel using filters.voxelgrid

In this exercise, we are going to thin with the Poisson method, but the concept should operate similarly for the filters.voxelgrid approach too.

Command

Invoke the following command, substituting accordingly, in your Conda Shell:

```
pdal translate ./exercises/analysis/density/uncompahgre.laz \
./exercises/analysis/thinning/uncompahgre-thin.laz \
sample --filters.sample.radius=20
```

Visualization

http://plas.io has the ability to switch on/off multiple data sets, and we are going to use that ability to view both the uncompahgre.laz and the uncompahgre-thin.laz file.
Fig. 3: Thinning strategies available in PDAL
Fig. 4: Selecting multiple data sets in http://plas.io
Fig. 5: Full resolution Uncompahgre data set
Fig. 6: Uncompahgre thinned at a radius of 20m
Notes

1. Poisson sampling is non-destructive. Points that are filtered with `filters.sample` (page 233) will retain all attribute information.

Identifying ground

This exercise uses PDAL to classify ground returns using the Simple Morphological Filter (SMRF) technique.

Note: This exercise is an adaptation of the pcl_ground tutorial on the PDAL website by Brad Chambers. You can find more detail and example invocations there.

Exercise

The primary input for Digital Terrain Model (https://en.wikipedia.org/wiki/Digital_elevation_model) generation is a point cloud with ground vs. not-ground classifications. In this example, we will use an algorithm provided by PDAL, the Simple Morphological Filter technique to generate a ground surface.

See also:

You can read more about the specifics of the SMRF algorithm from [Pingle2013].

Command

Invoke the following command, substituting accordingly, in your Conda Shell:

```
pdal translate ./exercises/analysis/ground/CSite1_orig-utm.laz -o ./exercises/analysis/ground/ground.laz smrf -v 4
```
As we can see, the algorithm does a great job of discriminating the points, but there’s a few issues.
There’s noise underneath the main surface that will cause us trouble when we generate a terrain surface.
Filtering

We do not yet have a satisfactory surface for generating a DTM. When we visualize the output of this ground operation, we notice there’s still some noise. We can stack the call to SMRF with a call to a the `filters.outlier` technique we learned about in denoising.

1. Let us start by removing the non-ground data to just view the ground data:

```bash
pdal translate \\
./exercises/analysis/ground/CSite1_orig-utm.laz \\
-o ./exercises/analysis/ground/ground.laz \\
smrf range \\
```

(continues on next page)
2. Now we will instead use the `translate` (page 39) command to stack the `filters.outlier` (page 159) and `filters.smrf` (page 155) stages:
In this invocation, we have more control over the process. First the outlier filter merely classifies outliers with a \texttt{Classification} value of 7. These outliers are then ignored during SMRF processing with the \texttt{ignore} option. Finally, we add a range filter to extract only the ground returns (i.e., \texttt{Classification} value of 2).

The result is a more accurate representation of the ground returns.
Generating a DTM

This exercise uses PDAL to generate an elevation model surface using the output from the *Identifying ground* (page 398) exercise, PDAL’s *writers.gdal* (page 119) operation, and GDAL (http://gdal.org/) to generate an elevation and hillshade surface from point cloud data.
Exercise

Note: The primary input for Digital Terrain Model (https://en.wikipedia.org/wiki/Digital_elevation_model) generation is a point cloud with ground classifications. We created this file, called denoised-ground-only.laz, in the Identifying ground (page 398) exercise. Please produce that file by following that exercise before starting this one.

Command

Invoke the following command, substituting accordingly, in your Conda Shell:

PDAL capability to generate rasterized output is provided by the writers.gdal (page 119) stage. There is no application (page 25) to drive this stage, and we must use a pipeline.

Pipeline breakdown

```
{
    "pipeline": [
        "./exercises/analysis/ground/denoised-ground-only.laz",
        {
            "filename": "./exercises/analysis/dtm/dtm.tif",
            "gdaldriver": "GTiff",
            "output_type": "all",
            "resolution": "2.0",
            "type": "writers.gdal"
        }
    ]
}
```

Note: This pipeline is available in your workshop materials in the ./exercises/analysis/dtm/dtm.json file. Make sure to edit the filenames to match your paths.
1. Reader

denoised-ground-only is the LASzip (http://laszip.org) file we will clip. You should have created this output as part of the Identifying ground (page 398) exercise.

2. writers.gdal

The writers.gdal (page 119) writer that bins the point cloud data into an elevation surface.

**Execution**

```
pdal pipeline ./exercises/analysis/dtm/gdal.json

(pdalworkshop) $ pdal pipeline ./exercises/analysis/dtm/gdal.json
(pdalworkshop) $]
```

**Visualization**

Something happened, and some files were written, but we cannot really see what was produced. Let us use qgis to visualize the output.

1. Open qgis and *Add Raster Layer*:
2. Add the *dtm.tif* file from your */exercises/analysis/dtm* directory.
3. Classify the DTM by right-clicking on the \textit{dtm.tif} and choosing \textit{Properties}. Pick the pseudocolor rendering type, and then choose a color ramp and click \textit{Classify}. 
4. qgis provides access to GDAL (http://gdal.org/) processing tools, and we are going to use that to create a hillshade of our surface. Choose **Raster→Analysis→Dem:**
5. Click the window for the Output file and select a location to save the hillshade.tif file.
6. Click **OK** and the hillshade of your DTM is now available
Notes

1. **gdaldem** ([http://www.gdal.org/gdaldem.html](http://www.gdal.org/gdaldem.html)), which powers the qgis DEM tools, is a very powerful command line utility you can use for processing data.

2. **writers.gdal** (page 119) can be used for large data, but it does not interpolate a typical TIN ([https://en.wikipedia.org/wiki/Triangulated_irregular_network](https://en.wikipedia.org/wiki/Triangulated_irregular_network)) surface model.

Creating surface meshes

This exercise uses PDAL to create surface meshes. PDAL is able to use a number of meshing filters: [https://pdal.io/stages/filters.html#mesh](https://pdal.io/stages/filters.html#mesh). Three of these are ‘in the box’, without needing plugins compiled. These are 2D Delaunay triangulation, Greedy projection meshing and Poisson surface meshing.

In this exercise we’ll create a Poisson surface mesh - a watertight isosurface - from our input point cloud.

Exercise

We will create mesh models of a building and its surrounds using an entwine data input source. After running each command, the output .ply file can be viewed in Meshlab or CloudCompare.

See also:

PDAL implements Mischa Kazhdan’s Poisson surface reconstruction algorithm. For details see [Kazhdan2006]_

---

**Note:** *writers.ply* will write out mesh vertices by default. In this exercise we set the attribute `faces="true"`. Try using the ply writer without it. Also, if you’re using a machine with a lot of processing power, try increasing the `depth` parameter for a more detailed mesh.

---

Command

Invoke the following command, substituting accordingly, in your *Conda Shell*:

```bash
  -o ./exercises/analysis/meshing/first-mesh.ply \
  poisson --filters.poisson.depth=16 \
  --readers.ept.bounds="([692738, 692967], [6092255,6092562])" \
  --verbose 4
```
You can view the mesh in Cloud Compare, you should see something similar to
Filtering

If we want to just mesh a building, or just terrain, or both we can apply a range filter based on point classification. These data have ground labeled as class 2, and buildings as 6.

In this exercise we will create a poisson mesh surface of a building and the ground surrounding it, using the same data subset as above and adding a filters.range (page 248) stage to limit the set of points used in mesh creation.

Command

Invoke the following command, substituting accordingly, in your Conda Shell:

```
    -o ./exercises/analysis/meshing/building-exercise.ply \
    range poisson \
    --filters.range.limits="Classification[2:2],Classification[6:6]" \
    --filters.poisson.depth=16 \
    --readers.ept.bounds="([692738, 692967], [6092255,6092562])" \
    --verbose 4
```

```
    -o ./exercises/analysis/meshing/building-exercise.ply ^
    range poisson ^
    --filters.range.limits="Classification[2:2],Classification[6:6]" ^
    --filters.poisson.depth=16 ^
    --readers.ept.bounds="([692738, 692967], [6092255,6092562])" ^
    --verbose 4
```

    > -o ./exercises/analysis/meshing/building-exercise.ply \
    > range poisson \
    > --filters.range.limits="Classification[2:2],Classification[6:6]" \
    > --filters.poisson.depth=16 \
    > --readers.ept.bounds="([692738, 692967], [6092255,6092562])" \
    > --verbose 2

# Read input into tree:
#  Got kernel density:
#  Got normal field:
#  Finalized tree:
#  Set FEM constraints:
#Set point constraints:
Got average:
(pdalworkshop) $
Rasterizing Attributes

This exercise uses PDAL to generate a raster surface using a fully classified point cloud with PDAL’s `writers.gdal` (page 119).

Exercise

Note: The exercise fetches its data from a Entwine (https://entwine.io) service that organizes the point cloud collection for the entire country of Denmark. You can view the data online at http://potree.entwine.io/data/denmark.html

Command

PDAL capability to generate rasterized output is provided by the `writers.gdal` (page 119) stage. There is no `application` (page 25) to drive this stage, and we must use a pipeline.

Pipeline breakdown

```json
{
  "pipeline": [
    {
      "type": "readers.ept",
      "filename": "http://na-c.entwine.io/dk",
      "bounds": "([1401016, 1410670], [7476527, 7484590])",
      "resolution": 5
    },
    {
      "type": "writers.gdal",
      "filename": "denmark-classification.tif",
      "dimension": "Classification",
      "data_type": "uint16_t",
      "output_type": "mean",
      "resolution": 5
    }
  ]
}
```

Note: This pipeline is available in your workshop materials in the `./exercises/analysis/dtm/dtm.json` file. Make sure to edit the filenames to match
The data is read from a EPT resource that contains the Denmark data. We’re going to
download a small patch of data by the Copenhagen airport area that is the limited to a spatial
resolution of 5m.

2. writers.gdal

The writers.gdal (page 119) writer that bins the point cloud data with classification values.

Execution

Issue the pipeline (page 45) operation to execute the interpolation:

```
pdal pipeline ./exercises/analysis/rasterize/classification.json -v 3
```

(continues on next page)
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

(continued from previous page)

},
{
    "type":"writers.gdal",
    "filename":"denmark-classification.tif",
    "dimension":"Classification",
    "data_type":"uint16_t",
    "output_type":"mean",
    "resolution": 5
}
}

(pdalworkshop) $ pdal pipeline ./exercises/analysis/rasterize/classification.json -v 3
(PDAL Debug) Debugging...


Get EPT info

SRS: PROJCS["WGS 84 / Pseudo-Mercator", GEOCSGCS["WGS 84", DATUMGCS["WGS 84", SPHEROIDGCS["WGS 84", 6378137.0, 298.257223563, AUTHORITY["EPSG","7030"]], AUTHORITY["EPSG","6326"]], PROJECTION["Pseudo-Mercator"], AUTHORITY["EPSG","9800"]], UNIT["degree", 0.0174532925199433, AUTHORITY["EPSG","8922"]], AUTHORITY["EPSG","4326"]], PROJECTION["Mercator_1SP"], AUTHORITY["EPSG","9001"]], PARAMETER["central_meridian", 0], PARAMETER["scale_factor", 1], PARAMETER["false_easting", 0], PARAMETER["false_northing", 0], UNIT["metre", 1], AUTHORITY["EPSG","9001"]], AXIS["x", EAST], AXIS["y", NORTH], EXTENSION["PROJ3", "+proj=merc +a=6378137 +b=6378137 +lat_ts=1.0 +lon_0=0.0 +x_0=0.0 +y_0=0.0 +k=1.0 +units=m +nadgrids=@null +text_wgs_84_def", AUTHORITY["EPSG","3857"]]

Root resolution: 3308.53

Query resolution: 10
Actual resolution: 6.07135

Depth end: 10

Query bounds: ((1104622, 1107468), (7762273, 7777891), [-1.797693134862316e+308, 1.797693134862316e+308])

Threads: 8

(pdal pipeline readers.gdal Debug) Registering dim X: double
(pdal pipeline readers.gdal Debug) Registering dim Y: double
(pdal pipeline readers.gdal Debug) Registering dim Z: double
(pdal pipeline readers.gdal Debug) Registering dim INTENSITY: uint16_t
(pdal pipeline readers.gdal Debug) Registering dim ReturnNumber: uint8_t
(pdal pipeline readers.gdal Debug) Registering dim ScanDirectionFlag: uint8_t
(pdal pipeline readers.gdal Debug) Registering dim EdgeOffLightline: uint8_t
(pdal pipeline readers.gdal Debug) Registering dim Classification: uint8_t
(pdal pipeline readers.gdal Debug) Registering dim ScanAngleRank: float
(pdal pipeline readers.gdal Debug) Registering dim UserData: uint8_t
(pdal pipeline readers.gdal Debug) Registering dim PointSourceID: uint16_t
(pdal pipeline readers.gdal Debug) Registering dim GpsTime: double
(pdal pipeline readers.gdal Debug) Registering dim Red: uint16_t
(pdal pipeline readers.gdal Debug) Registering dim Green: uint32_t
(pdal pipeline readers.gdal Debug) Registering dim Blue: uint16_t
(pdal pipeline Debug) Executing pipeline in standard mode.
(pdal pipeline readers.gdal Debug) Overlap points: 7242657
(pdal pipeline readers.gdal Debug) Data 1/79: 0-0-0-0
(pdal pipeline readers.gdal Debug) Data 2/79: 1-0-1-1
(pdal pipeline readers.gdal Debug) Data 4/79: 3-2-3-4
(pdal pipeline readers.gdal Debug) Data 5/79: 4-3-11-8
(pdal pipeline readers.gdal Debug) Data 6/79: 5-8-22-16
(pdal pipeline readers.gdal Debug) Data 7/79: 5-8-23-16
(pdal pipeline readers.gdal Debug) Data 8/79: 6-16-45-32
(pdal pipeline readers.gdal Debug) Data 9/79: 6-16-46-32

420 Chapter 13. Workshop
Visualization

Basic interpolation of data with \texttt{writers.gdal} (page 119) will output raw classification values into the resulting raster file. We will need to add a color ramp to the data for a satisfactory preview.

Unfortunately, this does not give us a very satisfactory image to view. The reason is there is no color ramp associated with the file, and we’re looking at pixel values with values from 0-31 according to the ASPRS LAS specification.

We want colors that correspond to the classification values a bit more directly. We can use a color ramp to assign explicit values. \texttt{qgis} allows us to create a text file color ramp that \texttt{gdaldem} can consume to apply colors to the data.

\begin{verbatim}
# QGIS Generated Color Map Export File
2 139 51 38 255 Ground
3 143 201 157 255 Low Veg
4 5 159 43 255 Med Veg
5 47 250 11 255 High Veg
6 209 151 25 255 Building
7 232 41 7 255 Low Point
8 197 0 204 255 reserved
\end{verbatim}

(continues on next page)
With this ramp, you can load the color values into QGIS as a color ramp if you change the layer to Paletted/Unique Values, and then load the color ramp file:

With the ramp, we can also use gdaldem (http://www.gdal.org/gdaldem.html) to apply it to a new image:
Intensity

With PDAL’s ability to override pipeline via commands, we can generate a relative intensity image:

```bash
pdal pipeline ./exercises/analysis/rasterize/classification.json \
  --writers.gdal.dimension="Intensity" \
  --writers.gdal.data_type="float" \
  --writers.gdal.filename="intensity.tif" \
  -v 3

gdal_translate intensity.tif intensity.png -of PNG
```

(continues on next page)
The same pipeline can be used to generate a preview image of the Intensity channel of the data by overriding pipeline arguments at the command line.

```
-v 3
gdal_translate intensity.tif intensity.png -of PNG
```

Notes

1. *writers.gdal* (page 119) can output any dimension PDAL can provide, but it is up to the user to interpolate the values. For categorical data, neighborhood smoothing might produce undesirable results, for example.

2. *Pipeline* (page 45) contains more information about overrides and organizing complex pipelines.
Python

Plotting a histogram

Exercise

PDAL doesn’t provide every possible analysis option, but it strives to make it convenient to link PDAL to other places with substantial functionality. One of those is the Python/Numpy universe, which is accessed through PDAL’s Python (page 295) bindings and the filters.python (page 276) filter. These tools allow you to manipulate point cloud data with convenient Python tools rather than constructing substantial C/C++ software to achieve simple tasks, compute simple statistics, or investigate data quality issues.

This exercise uses PDAL to create a histogram plot of all of the dimensions of a file. matplotlib (https://matplotlib.org/) is a Python package for plotting graphs and figures, and we can use it in combination with the Python (page 295) bindings for PDAL to create a nice histogram. These histograms can be useful diagnostics in an analysis pipeline. We will combine a Python script to make a histogram plot with a pipeline (page 32).

Note: Python allows you to enhance and build functionality that you can use in the context of other Pipeline (page 45) operations.

PDAL Pipeline

We’re going to create a PDAL Pipeline (page 45) to tell PDAL to run our Python script in a filters.python (page 276) stage.

```json
{
    "pipeline": [
        {
            "filename": "./exercises/python/athletic-fields.laz"
        },
        {
            "type": "filters.python",
            "function": "make_plot",
            "module": "anything",
            "pdalargs": "{"filename": "/exercises/python/histogram.png"}",
            "script": "/exercises/python/histogram.py"
        },
        {
            "type": "writers.null"
        }
    ]
}
```

(continues on next page)
Note: This pipeline is available in your workshop materials in the ./exercises/python/histogram.json file.

Python script

The following Python script will do the actual work of creating the histogram plot with matplotlib (https://matplotlib.org/). Store it as histogram.py next to the histogram.json Pipeline (page 45) file above. The script is mostly regular Python except for the ins and outs arguments to the function – those are special arguments that PDAL expects to be a dictionary of Numpy dictionaries.

Note: This Python file is available in your workshop materials in the ./exercises/python/histogram.py file.

```python
# import numpy
import numpy as np

# import matplotlib stuff and make sure to use the # AGG renderer.
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

# This only works for Python 3. Use # StringIO for Python 2.
from io import BytesIO

# The make_plot function will do all of our work. The # filters.programmable filter expects a function name in the # module that has at least two arguments -- "ins" which # are numpy arrays for each dimension, and the "outs" which # the script can alter/set/adjust to have them updated for # further processing.
def make_plot(ins, outs):
    (continues on next page)
```
23 # figure position and row will increment
24 figure_position = 1
25 row = 1
26
27 fig = plt.figure(figure_position, figsize=(6, 8.5), dpi=300)
28
29 for key in ins:
30     dimension = ins[key]
31     ax = fig.add_subplot(len(ins.keys()), 1, row)
32
33     # histogram the current dimension with 30 bins
34     n, bins, patches = ax.hist(dimension, 30,
35                                 normed=0,
36                                 facecolor='grey',
37                                 alpha=0.75,
38                                 align='mid',
39                                 histtype='stepfilled',
40                                 linewidth=None)
41
42     # Set plot particulars
43     ax.set_ylabel(key, size=10, rotation='horizontal')
44     ax.get_xaxis().set_visible(False)
45     ax.set_yticklabels('')
46     ax.set_yticks((),)
47     ax.set_xlim(min(dimension), max(dimension))
48     ax.set_ylim(min(n), max(n))
49
50     # increment plot position
51     row = row + 1
52     figure_position = figure_position + 1
53
54     # We will save the PNG bytes to a BytesIO instance
55     # and the nwrite that to a file.
56     output = BytesIO()
57     plt.savefig(output, format="PNG")
58
59     # a module global variable, called 'pdalargs' is available
60     # to filters.programmable and filters.predicate modules that
61     # contains
62     # a dictionary of arguments that can be explicitly passed into
63     # the module by the user. We passed in a filename arg in our
64     # `pdal pipeline` call
65     if 'filename' in pdalargs:
66         filename = pdalargs['filename']
67     else:
filename = 'histogram.png'

# open up the filename and write out the
# bytes of the PNG stored in the BytesIO instance
o = open(filename, 'wb')
o.write(output.getvalue())
o.close()

# filters.programmable scripts need to
# return True to tell the filter it was successful.
return True

Run pdal pipeline

pdal pipeline ./exercises/python/histogram.json

(pdalworkshop) $ pdal pipeline ./exercises/python/histogram.json
anything:46: MatplotlibDeprecationWarning: The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1. Use 'density' instead.
anything:47: UserWarning: Attempting to set identical left == right == 0 results in singular transformations; automatically expanding.
(pdalworkshop) $
13.1. Point Cloud Processing and Analysis with PDAL
Notes

1. *writers.null* (page 132) simply swallows the output of the pipeline. We don’t need to write any data.

2. The **pdalargs** JSON needs to be escaped because a valid Python dictionary entry isn’t always valid JSON.

Georeferencing

Georeferencing

As discussed *in the introduction* (page 352), laser returns from a mobile LiDAR (https://en.wikipedia.org/wiki/Lidar) system must be georeferenced, i.e. placed into a local or global coordinate system by combining data from the laser and from a GNSS/IMU. As of this writing, PDAL does *not* include generic georeferencing tools — this is considered future work. However, the **Optech** (http://www.teledyneoptech.com/) csd file format includes both laser return and GNSS/IMU data in the same file, and the PDAL csd reader includes built in georeferencing support.

In this section, we will demonstrate how to georeference an **Optech** (http://www.teledyneoptech.com/) csd file and reproject that file into a UTM projection.

---

**Note:** **Optech**’s (http://www.teledyneoptech.com/) csd format is just one of several vendor-specific data formats PDAL supports; we also support data files directly from **RIEGL** (http://riegl.com/) sensors and from several project-specific government platforms.

Exercise

The file **S1C1_csd_004.csd** contains airborne data from an **Optech** (http://www.teledyneoptech.com/) sensor. Without georeferencing these points, they would be impossible to interpret — once they are georeferenced, we will be able to inspect and analyze these points like any other point cloud.

In addition to georeferencing, we are going to make two other tweaks to our point cloud:

- The point cloud is, by default, in **WGS84** (https://en.wikipedia.org/wiki/Geodetic_datum), but we will reproject these points to a **UTM** (https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system) coordinate system for visualization purposes.
• Because these are raw data coming from the sensor, these data are noisy. In particular, there are a few points very close to the sensor which were probably caused by air returns or laser light reflecting off of part of the airplane or sensor. These points have very high intensity values, which will screw up our visualization. We will use the `filters.range` (page 248) PDAL filter to drop all points with very high intensity values.

**Note:** These data were provided by Dr. Craig Glennie and were collected by NCALM (http://ncalm.cive.uh.edu/), the National Center for Airborne Laser Mapping. The collect area is southwest of Austin, TX.

### Command

Invoke the following command, substituting accordingly, into your `Conda Shell`:

```bash
pdal translate \
./exercises/georeferencing/S1C1_csd_004.csd \
./exercises/georeferencing/S1C1_csd_004.laz \
reprojection range \
--filters.reprojection.out_srs="EPSG:32614" \
--filters.range.limits="Intensity[0:500]"
```

```bash
pdal translate ^
./exercises/georeferencing/S1C1_csd_004.csd ^
./exercises/georeferencing/S1C1_csd_004.laz ^
reprojection range ^
--filters.reprojection.out_srs="EPSG:32614" ^
--filters.range.limits="Intensity[0:500]"
```
Visualization

View your georeferenced point cloud in http://plas.io.

Fig. 7: Our airborne laser point cloud after georeferencing, reprojection, and intensity filtering.

Batch Processing

PDAL doesn’t handle matching multiple file inputs except for glob handling for merge operations, but does allow for command line substitution parameters to make batch processing simpler, substitutions. Substitutions work with both Pipeline (page 45) operations as well as with other applications such as translate (page 39).
Operating system variations

How substitutions are passed generally depends on the operating system and tools available. In the unix/linux environments, this is primarily using the `find` and `ls` programs to get lists of files (either with directories or just filenames) and the `xargs` or `parallel` program to pass those files to the `pdal` application (although `-exec` with `find` can also be used). These tools are available in the `docker` environment if you are running PDAL under docker. They are also available under Windows one installs `Cygwin` or `MinGW`. They are also available if Git for Windows is installed. They are also available as win32 command line programs installed from the GNU Findutils (https://www.gnu.org/software/findutils/findutils.html). They are available for MacOS and Linux.

Windows native tools

Substitutions can be handled directly in windows using `PowerShell` syntax.

While there are a number of ways to generate lists of files, the `Get-ChildItem` is used here, along with the `foreach` option to pass each separate filepath to the `pdal` application.

Example - Batch compression of LAS files to LAZ - PowerShell:

To compress a series of LAS files in one directory into compressed LAZ files in another directory, the `PowerShell` syntax would be:

```
Get-ChildItem .\DIR1\*.las | foreach { pdal translate -i .\DIR1\$_ BaseName).las ^ -o .\DIR2\$_.laz }
```

Note the use of the `$_ BaseName)` syntax for the files passed. This option on the `$_` shortcut for the full filename, removes the directory and the extension on the file and allows the user to set the path and extension manually.

Example - Parallel Batch compression of LAS files to LAZ - PowerShell:

This use of the `PowerShell` syntax doesn’t allow a user to execute more than one process at a time. There is a free download of the `xargs` program that provides parallel execution available at http://www.pirosa.co.uk/demo/wxargs/ppx2.exe. For this tool, the file names are passed with using the `{}` syntax.

```
Get-ChildItem .\dir1\ | Select-Object -ExpandProperty BaseName ^ | .\ppx2.exe -P 3 pdal translate -i ".\dir1\{}.las" -o ".\dir2\{}.laz ")
```
Example - Batch compression of LAS files to LAZ - Bash:

To compress a series of LAS files in one directory into compressed LAZ files in another directory, the Bash syntax would be:

```
ls ./dir1/*.las | parallel -I{} \npdal translate -i ./dir1/{/.}.las -o ./dir2/{/.}.laz
```

In Parallel, then {/.} syntax means strip the directory and the extension and just use the basename of the file. This allows you to easily change the output format and the location.

Example - Parallel Batch compression of LAS files to LAZ - Bash:

Parallel, as its name implies, allows parallel operations. Adding the -j syntax indicates the number simultaneous jobs to run

```
ls ./dir1/*.las | parallel -I{} -j 4 \npdal translate -i ./dir1/{/.}.las -o ./dir2/{/.}.laz
```

Exercise - Pipeline Substitutions:

For the most flexibility, pipelines are used to apply a series of operations to a file (or group of files). In this exercise, we build on the Generating a DTM (page 404) pipeline example, but run this pipeline over 4 files and reproject, calculate a bare earth using the filters.smrf (page 155) filter, remove those points that aren’t bare earth with filters.range (page 248) and then write the output using the writers.gdal (page 119).

The pipeline we are using is:

```
{
   "pipeline": [
      {
         "type": "readers.las"
      },
      {
         "type": "filters.reprojection"
      },
      {
         "type": "filters.smrf"
      },
      {
         "type": "filters.range",
         "limits": "Classification[2:2]"
      }
   ]
```

(continues on next page)
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

You might have spotted that this pipeline doesn’t have any input or output file references, or a value for the output spatial reference. We will be adding those at the command line, not within the actual pipeline and using the substitutions syntax to do this.

```
PS ./exercises/batch> Get-ChildItem ./exercises/batch/ | foreach {pdal pipeline ./exercises/batch/batch_srs_gdal.json --readers.las.filename=./source/$($_.BaseName).laz --writers.gdal.filename=./dtm/$($_.BaseName).tif --filters.reprojection.in_srs=epsg:3794 --filters.reprojection.out_srs=epsg:32733}
```

```
ls ./exercises/batch_processing/source/*.laz | \parallel -I{} pdal pipeline ./exercises/batch_processing/batch_srs_gdal.json \--readers.las.filename={} \--writers.gdal.filename=./exercises/batch_processing/dtm/./.tif \--filters.reprojection.in_srs=epsg:3794 \--filters.reprojection.out_srs=epsg:32733
```

Once you have your dtms created with pdal, combine them to a single file with:

```
gdalbuildvrt ./exercises/batch_processing/dtm.vrt ./exercises/batch_processing/dtm*.tif
```

You can then visualize the vrt with qgis. Add the vrt twice, and set the properties of the lower layer to hillshade. Set the upper layer to Singleband PseudoColor and choose a pleasing color ramp. Then set the transparency of the upper layer to 50% and you’ll get a nice display of the terrain.
13.1.5 Final Project

The final project brings together a number of PDAL processing workflow operations into a single effort. It builds upon the exercises to enable you to use the capabilities of PDAL in a coherent processing strategy, and it will give you ideas about how to orchestrate PDAL in the context of larger data processing scenarios.

Given the following pipeline for fetching the data, complete the rest of the tasks:

```json
{
    "pipeline": [
        // Rest of the tasks
    ]
}
(continues on next page)
• Read data from an EPT resource using *readers.ept* (page 56) (See *Entwine* (page 367))

• Thin it by 1.0 meter spacing using *filters.sample* (page 233) (See *Thinning* (page 392))

• Filter out noise using *filters.outlier* (page 159) (See *Removing noise* (page 384))

• Classify ground points using *filters.smrf* (page 155) (See *Identifying ground* (page 398))

• Compute height above ground using *filters.hag_nn* (page 171)

• Generate a digital terrain model (DTM) using *writers.gdal* (page 119) (See *Generating a DTM* (page 404))

• Generate an average vegetative height model using *writers.gdal* (page 119)

**Note:** You should review specific *Exercises* (page 357) for specifics how to achieve each task.

### 13.1.6 Notes

**Notes**
Notes
14.1 Development

Developer documentation, such as how to update the docs, where the test frameworks are, who develops the software, and conventions to use when developing new code can be found in this section.

**Note:** Users looking for documentation on how to use PDAL’s command line applications should look [here](#) (page 25) and users looking to use the PDAL API in their own applications should look [here](#) (page 519).

14.1.1 PDAL Architecture Overview

**Author** Andrew Bell  
**Contact** andrew@hobu.co  
**Date** 5/15/2016

PDAL is a set of applications and library to facilitate translation of point cloud data between various formats. In addition, it provides some facilities for transformation of data between various geometric projections and can calculate some statistical, boundary and density data. PDAL also provides point classification algorithms. PDAL provides an API that can be used by programmers for integration into their own projects or to allow extension of existing capabilities.
The PDAL model

PDAL reads data from a set of input sources using format-specific readers. Point data can be passed through various filters that transform data or create metadata. If desired, points can be written to an output stream using a format-specific writer. PDAL can merge data from various input sources into a single output source, preserving attribute data where supported by the input and output formats.

The above diagram shows a possible arrangement of PDAL readers, filters and writers, all of which are known as stages. Any merge operation or filter may be placed after any reader. Output filters are distinct from other filters only in that they may create more than one set of points to be further filtered or written. The arrangement of readers, filters and writers is called a PDAL pipeline. Pipelines can be specified using JSON as detailed later.

Extending PDAL

PDAL is simple to extend by implementing subclasses of existing stages. All processing in PDAL is completely synchronous. No parallel processing occurs, eliminating locking or other concurrency issues. Understanding of several auxiliary classes is necessary to effectively create a new stage.
Dimension

Point cloud formats support various data elements. In order to be useful, all formats must provide some notion of location for points (X, Y and perhaps Z), but beyond that, the data collected in formats may or may not have common data fields. Some formats redefine the elements that make up a point. Other formats provide this information in a header or preamble. PDAL calls each of the elements that make up a point a dimension. PDAL predefines the dimensions that are in common use by the formats that it currently supports. Readers may register their use of a predefined dimension or may have PDAL create a dimension with a name and type as requested. Dimensions are described in a JSON file, Dimension.json.

PDAL has a default type (Double, Float, Signed32, etc.) for each of its predefined dimensions which is believed to be sufficient to accurately hold the necessary data. Only when the default data type is deemed insufficient should a request be made to “upgrade” a storage datatype. There is no simple facility to “downsize” a dimension type to save memory, though it can be done by creating a custom PointLayout object. Dimension.json can be examined to determine the default storage type of each predefined dimension. In most cases knowledge of the storage data type for a dimension isn’t required. PDAL properly converts data to and from the internal storage type transparently. Invalid conversions raise an exception.

When a storage type is explicitly requested for a dimension, PDAL examines the existing storage type and requested type and chooses the storage type so that it can hold both types. In some cases this results in a storage type different from either the existing or requested storage type. For instance, if the current storage type is a 16 bit signed integer (Signed16) and the requested type is a 16 bit unsigned integer (Unsigned16), PDAL will use a 32 bit signed integer as the storage type for the dimension so that both 16 bit storage types can be successfully accommodated.

Point Layout

PDAL stores the dimension information in a point layout structure (PointLayout object). It stores information about the physical layout of data of each point in memory and also stores the type and name of each dimension.

Point Table

PDAL stores points in what is called a point table (PointTable object). Each point table has an associated point layout describing its format. All points in a single point table have the same dimensions and all operations on a PDAL pipeline make use of a single point table. In addition to storing points, a point table also stores pipeline metadata that may be created as pipeline stages are executed. Most functions receive a PointTableRef object, which refers to the active point table. A PointTableRef can be stored or copied cheaply.
A subclass of PointTable called StreamingPointTable exists to allow a pipeline to run without loading all points in memory. A StreamingPointTable holds a fixed number of points. Some filters can’t operate in streaming mode and an attempt to run a pipeline with a stage that doesn’t support streaming will raise an exception.

**Point View**

A point view (PointView object) stores references to points. Storage and retrieval of points is done through a point view rather than directly through a point table. Point data is accessed from a point view through a point ID (type PointId), which is an integer value. The first point reference in a point view has a point ID of 0, the second has a point ID of 1, the third has a point ID of 2 and so on. There are no null point references in a point view. The size of a point view is the number of point references contained in the view. A point view acts like a self-expanding array or vector of point references, but it is always full. For example, one can’t set the field value of a point with a PointId of 9 unless there already exist at least 8 point references in the point view.

Point references can be copied from one point view to another by appending an existing reference to a destination point view. The point ID of the appended point in the destination view may be different than the point ID of the same point in the source view. The point ID of an appended point reference is the same as the size of the point view after the operation. Note that appending a point reference does not create a new point. Rather, it creates another reference to an existing point. There are currently no built-in facilities for creating copies of points.

**Point Reference**

Some functions take a reference to a single point (PointRef object). In streaming mode, stages implement the processOne() function which operates on a point reference instead of a point view.

**Making a Stage (Reader, Filter or Writer):**

All stages (Stage object) share a common interface, though readers, filters and writers each have a simplified interface if the generic stage interface is more complex than necessary. One should create a new stage by creating a subclass of reader (Reader object), filter (Filter object) or writer (Writer object). When a pipeline is made, each stage is created using its default constructor.

When a pipeline is started, each of its stages is processed in two distinct steps. First, all stages are prepared.
Stage Preparation

Preparation of a stage is done by calling the prepare() function of the stage at the end of the pipeline. prepare() executes the following private virtual functions calls, none of which need to be implemented in a stage unless desired. Each stage is guaranteed to be prepared after all stages that precede it in the pipeline.

1) void addArgs(ProgramArgs& args)

   Stages can accept various options to control processing. These options can be declared and bound to variables in this function. When arguments are added, the stage also provides a description and optionally a default value for the argument.

2) void initialize() OR void initialize(PointTableRef)

   Some stages, particularly readers, may need to do things such as open files to extract header information before the next step in processing. Other general processing that needs to take place before any stage is executed should occur at this time. If the initialization requires knowledge of the point table, implement the function that accepts one, otherwise implement the no-argument version. Whether to place initialization code at this step or in prepared() or ready() (see below) is a judgment call, but detection of errors earlier in the process allows faster termination of a command. Files opened in this step should also be closed before returning.

3) void addDimensions(PointLayoutPtr layout)

   This method allows stages to inform a point table’s layout of the dimensions that it would like as part of the record of each point. Usually, only readers add dimensions to a point table, but there is no prohibition on filters or writers from adding dimensions if necessary. Dimensions should not be added to the layout outside of this method.

4) void prepared(PointTableRef)

   Called after dimensions are added. It can be used to verify state and raise exceptions before stage execution.

Stage Execution

After all stages are prepared, processing continues with the execution of each stage by calling execute(). Each stage will be executed only after all stages preceding it in a pipeline have been executed. A stage is executed by invoking the following private virtual methods. It is important to note that ready() and done() are called only once for each stage while run() is called once for each point view to be processed by the stage.

1) void ready(PointTablePtr table)
This function allows preprocessing to be performed prior to actual processing of the points in a point view. For example, filters may initialize internal data structures or libraries, readers may connect to databases and writers may write a file header. If there is a choice between performing operations in the preparation stage (in the initialize() method) or the execution stage (in ready()), prefer to defer the operation until this point.

2) PointViewSet run(PointViewPtr buf)

This is the method in which processing of individual points occurs. One might read points into the view, transform point values in some way, or distribute the point references in the input view into numerous output views. This method is called once for each point view passed to the stage.

3) void done(PointTablePtr table)

This function allows a stage to clean up resources not released by a stage’s destructor. It also allows other execution of termination functions, such as closing of databases, writing file footers, rewriting headers or closing or renaming files.

Streaming Stage Execution

PDAL normally processes all points through each stage before passing the points to the next stage. This means that all point data is held in memory during processing. There are some situations that may make this undesirable. As an alternative, PDAL allows execution of data with a point table that contains a fixed number of points (StreamPointTable). When a StreamPointTable is passed to the execute() function, the private run() function detailed above isn’t called, and instead processOne() is called for each point. If a StreamPointTable is passed to execute() but a pipeline stage doesn’t implement processOne(), an exception is thrown.

bool processOne(PointRef& ref)

This method allows processing of a single point. A reader will typically read a point from an input source. When a reader returns ‘false’ from this function, it indicates that there are no more points to be read. When a filter returns ‘false’ from this function, it indicates that the point just processed should be filtered out and not passed to subsequent stages for processing.
Implementing a Reader

A reader is a stage that takes input from a point cloud format supported by PDAL and loads points into a point table through a point view.

A reader needs to register or assign those dimensions that it will reference when adding point data to the point table. Dimensions that are predefined in PDAL can be registered by using the point table’s registerDim() method. Dimensions that are not predefined can be added using assignDim(). If dimensions are determined as named entities from a point cloud source, it may not be known whether the dimensions are predefined or not. In this case the function registerOrAssignDim() can be used. When a dimension is assigned, rather than registered, the reader needs to inform PDAL of the type of the variable using the enumeration Dimension::Type.

In this example, the reader informs the point table’s layout that it will reference the dimensions X, Y and Z.

```cpp
void Reader::addDimensions(PointLayoutPtr layout)
{
    layout->registerDim(Dimension::Id::X);
    layout->registerDim(Dimension::Id::Y);
    layout->registerDim(Dimension::Id::Z);
}
```

Here a reader determines dimensions from an input source and registers or assigns them. All of the input dimension values are in this case double precision floating point.

```cpp
void Reader::addDimensions(PointLayoutPtr layout)
{
    FileHeader header;

    for (auto di = header.names.begin(), di != header.names.end(); ++di)
    {
        std::string dimName = *di;
        Dimension::Id id = layout->registerOrAssignDim(dimName,
            Dimension::Type::Double);
    }
}
```

If a reader implements initialize() and opens a source file during the function, the file should be closed again before exiting the function to ensure that file handles aren’t exhausted when processing a large number of files.

Readers should use the ready() function to reset the input data to a state where the first point can be read from the source. The done() function should be used to free resources or reset the state initialized in ready().

14.1. Development 451
Readers should implement a function, `read()`, that will place the data from the input source into the provided point view:

```cpp
template<>
point_count_t read(PointViewPtr view, point_count_t count)
{
    // Determine the number of points remaining in the input.
    point_count_t remainingInput = m_totalNumPts - m_index;

    // Determine the number of points to read.
    count = std::min(count, remainingInput);

    // Determine the ID of the next point in the point view
    PointId nextId = view->size();

    // Determine the current input position.
    auto pos = m_pointSize * m_index;

    point_count_t remaining = count;
    while (remaining--)
    {
        double x, y, z;

        // Read X, Y and Z from input source.
        x = m_file.read<double>(pos);
        pos += sizeof(double);
        y = m_file.read<double>(pos);
        pos += sizeof(double);
        z = m_file.read<double>(pos);
        pos += sizeof(double);

        // Set X, Y and Z into the pointView.
        view->setField(Dimension::Id::X, nextId, x);
    }
}
```

The reader should read at most ‘count’ points from the input source and place them in the view. The reader must keep track of its current position in the input source and points should be read until no points remain or ‘count’ points have been added to the view. The current location in the input source is typically tracked with an integer variable called the index.

As each point is read from the input source, it must be placed at the end of the point view. The ID of the end of the point view can be determined by calling `size()` function of the point view. `read()` should return the number of points read by during the function call.

(continues on next page)
view->setField(Dimension::Id::Y, nextId, y);
view->setField(Dimension::Id::Z, nextId, z);

nextId++;
}
m_index += count;
return count;
}

Note that we don’t read more points than requested, we don’t read past the end of
the input stream and we keep track of our location in the input so that subsequent
calls to read() will result in all points being read.

Here’s the same function written so that streaming can be supported:

point_count_t MyFormat::read(PointViewPtr view, point_count_t count)
{

    // Determine the number of points remaining in the input.
    point_count_t remainingInput = m_totalNumPts - m_index;

    // Determine the number of points to read.
    count = std::min(count, remainingInput);

    // Determine the ID of the next point in the point view
    PointId nextId = view->size();

    // Determine the current input position.
    auto pos = m_pointSize * m_index;

    point_count_t remaining = count;
    while (remaining--)
    {
        PointRef point(view->point(nextId));

        processOne(point);
        nextId++;
    }
    m_index += count;
    return count;
}

bool MyFormat::processOne(PointRef& point)
{
    double x, y, z;

(continues on next page)
// Read X, Y and Z from input source.
x = m_file.read<double>(pos);
pos += sizeof(double);
y = m_file.read<double>(pos);
pos += sizeof(double);
z = m_file.read<double>(pos);
pos += sizeof(double);

point.setField(Dimension::Id::X, x);
point.setField(Dimension::Id::Y, y);
point.setField(Dimension::Id::Z, z);
return m_file.ok();

Implementing a Filter

A filter is a stage that allows processing of data after it has been read into a pipeline’s point table. In many filters, the only function that need be implemented is filter(), a simplified version of the stage’s run() method whose input and output is a point view provided by the previous stage:

```c
void filter(PointViewPtr view)
```

One should implement filter() instead of run() if its interface is sufficient. The expectation is that a filter will iterate through the points currently in the point view and apply some transformation or gather some data to be output as pipeline metadata.

Here as an example is the actual filter function from the reprojection filter:

```c
void Reprojection::filter(PointViewPtr view)
{
    for (PointId id = 0; id < view->size(); ++id)
    {
        double x = view->getFieldAs<double>(Dimension::Id::X, id);
        double y = view->getFieldAs<double>(Dimension::Id::Y, id);
        double z = view->getFieldAs<double>(Dimension::Id::Z, id);

        transform(x, y, z);
        view->setField(Dimension::Id::X, id, x);
    }
}
```
The filter simply loops through the points, retrieving the X, Y and Z values of each point, transforms those values using a reprojection algorithm and then stores the transformed values in the point table using the point view’s setField() function.

A filter may need to use the run() function instead of filter(), typically because it needs to create multiple output point views from a single input view. The following example puts every other input point into one of two output point views:

```cpp
PointViewSet Alternator::run(PointViewPtr view)
{
    PointViewSet viewSet;
    PointViewPtr even = view();
    PointViewPtr odd = view();
    viewSet.insert(even);
    viewSet.insert(odd);
    for (PointId idx = 0; idx < view->size(); ++idx)
    {
        PointViewPtr out = idx % 2 ? even : odd;
        out->appendPoint(*view.get(), idx);
    }
    return viewSet;
}
```

**Implementing a Writer:**

Analogous to the filter() method in a filter is the write() method of a writer. This function is usually the appropriate one to override when implementing a writer – it would be unusual to need to implement run(). A typical writer will open its output file when ready() is called, write individual points in write() and close the file in done().

Like a filter, a writer may receive multiple point views during processing of a pipeline. This will result in the write() function being called once for each of the input point views. Writers may produce a separate output file for each input point view or may produce a single output file. The documentation should clearly state this behavior. Placing a merge filter in front of a writer in the pipeline will make sure that a single point view is passed to the writer.

As new writers are created, developers should try to make sure that they behave reasonably if passed multiple point views – they correctly handle write() being called multiple times after a single call to ready().
void write(const PointViewPtr view)
{
    ostream& out = *m_out;

    for (PointId id = 0; id < view->size(); ++id)
    {
        out << setw(10) << view->getFieldAs<double>(Dimension::Id::X, id);
        out << setw(10) << view->getFieldAs<double>(Dimension::Id::Y, id);
        out << setw(10) << view->getFieldAs<double>(Dimension::Id::Z, id);
    }
}

bool processOne(PointRef& point)
{
    out << setw(10) << point.getFieldAs<double>(Dimension::Id::X);
    out << setw(10) << point.getFieldAs<double>(Dimension::Id::Y);
    out << setw(10) << point.getFieldAs<double>(Dimension::Id::Z);
}

14.1.2 Compilation

This section describes how to build and install PDAL under Windows, Linux, and Mac.

See also:

*Download* (page 13) contains links to installable binaries for Windows, OSX, and RHEL Linux systems.

Contents:

Unix Compilation

PDAL comes with support for building with CMake (https://cmake.org). PDAL requires at least version 3.5 of CMake. CMake is a cross-platform meta-build system that provides a unified system for building applications on multiple platforms with various build tools. CMake has generators (https://cmake.org/cmake/help/v3.5/manual/cmake-generators.7.html) for many build tools, though PDAL has been tested only with Ninja (https://ninja-build.org/) and GNU Makefiles (https://www.gnu.org/software/make/manual/make.html) on Unix/OSX. Ninja builds PDAL faster, so the following instructions use that build tool, though building with GNU Makefiles works similarly (simply replace “ninja” with “make” when running the build tool).
Dependencies

Building PDAL successfully depends on having other libraries configured and installed. These dependencies (page 464) can be built from source or can be installed via a packaging system (apt (https://help.ubuntu.com/lts/serverguide/apt.html) works well on Ubuntu and Debian-based Linux systems. Conda (https://conda.io/en/latest/) works well on most systems. Some have had success with brew (https://brew.sh/) on OSX systems.) Often, the only package that needs to be installed prior to building PDAL is GDAL. Installing a GDAL package will normally install other PDAL dependencies automatically.

```bash
$ apt install libgdal-dev
OR
$ conda install gdal
OR
$ brew install gdal
```

Using Ninja on Linux or OSX

Get the source code

PDAL can be cloned from GitHub (page 14) or you can download a release bundle (page 13)

Prepare a build directory

CMake allows you to generate different builders for a project. Here we’re using Mac OSX, but the procedure and output are nearly identical on Linux distributions.

```bash
$ cd PDAL
$ mkdir build
$ cd build
```
Run CMake

Running CMake uses the specified generator to create an environment suitable for building PDAL with the requested tool. (Ninja in this case).

```
$ cmake -G Ninja ..
-- Numpy output: /usr/lib/python2.7/dist-packages/numpy/core/include
1.13.3

-- Could NOT find LIBEXECINFO (missing: LIBEXECINFO_LIBRARY)
-- Could NOT find LIBUNWIND (missing: LIBUNWIND_LIBRARY LIBUNWIND_IN
CLUDE_DIR)
-- The following features have been enabled:

* PostgreSQL PointCloud plugin, read/write PostgreSQL PointCloud ob
jects
* Python plugin, add features that depend on python
* Unit tests, PDAL unit tests

-- The following OPTIONAL packages have been found:

* PkgConfig
* LibXml2
* Curl

-- The following REQUIRED packages have been found:

* GDAL (required version >= 2.2.0)
  Provides general purpose raster, vector, and reference system sup
port

... The following RECOMMENDED packages have not been found:

* LASzip (required version >= 3.1)
  Provides LASzip compression

-- Configuring done
-- Generating done
-- Build files have been written to: /home/foo/pdal/build
```
**Issue the *ninja* command**

If cmake runs to completion (reports that build files have been written), you can run Ninja to build PDAL.

```
$ ninja
```

If no errors are reported, Ninja will have created the `pdal` program in the `bin` directory. A set of necessary support libraries will have been created in the `lib` directory.

```
$ ls bin/pdal
bin/pdal

$ ls lib/libpdalcpp*
lib/libpdalcpp.8.dylib
lib/libpdalcpp.dylib
lib/libpdalcpp.9.0.0.dylib
```

**Checking the build and running PDAL tests**

You can quickly check that PDAL has built properly by running the `pdal info` command.

```
$ bin/pdal info ./test/data/las/autzen_trim.las
{
  "filename": "/test/data/las/autzen_trim.las",
  "pdal_version": "1.8.0 (git-version: c39e62)",
  "stats":
  {
    "bbox":
    {
      "EPSG:4326":
      {
        "bbox":
        {
          "maxx": -123.0689038,
          "maxy": 44.0515451,
          "maxz": 158.651448,
          "minx": -123.0734481,
          "miny": 44.04990077,
          "minz": 123.828048
        },
        ...
      }
    }
  }
```

CMake will normally build a set of tests that can be used to verify that PDAL executes most functions properly. You can run these tests yourself if desired, though it’s not typically
$ ctest
Test project /Users/foo/pdal.master/build
  Start  1: pdal_filters_pcl_block_test
    1/97 Test #1: pdal_filters_pcl_block_test ............ Passed
    →0.23 sec
    Start  2: pdal_filters_icp_test
    2/97 Test #2: pdal_filters_icp_test ................. Passed
    →0.12 sec
    Start  3: pdal_filters_python_test
    3/97 Test #3: pdal_filters_python_test .............. Passed
    →3.52 sec
    Start  4: pdal_io_numpy_test
    4/97 Test #4: pdal_io_numpy_test .................... Passed
    →0.31 sec
...
93/96 Test #93: pdal_io_ilvis2_metadata_test .......... Passed
    →0.03 sec
    Start  94: pdal_io_ilvis2_reader_metadata_test
94/96 Test #94: pdal_io_ilvis2_reader_metadata_test .... Passed
    →0.05 sec
    Start  95: xml_schema_test
95/96 Test #95: xml_schema_test ....................... Passed
    →0.04 sec
    Start  96: pdal_io_ilvis2_test
96/96 Test #96: pdal_io_ilvis2_test .................... Passed
    →0.04 sec

100% tests passed, 0 tests failed out of 96
Total Test time (real) = 39.54 sec

Failed tests may not indicate problems other than a lack of support for some feature on your system. For example, tests for database drivers will fail if the database isn’t installed or configured properly.

**Install PDAL**

PDAL can be installed to the default location (usually subdirectories of `/usr/local`) using Ninja.
$ ninja install

**Building Under Windows**

**Author**  Howard Butler  
**Contact**  howard at hobu.co  
**Date**  03/20/2019

**Note:**  _Conda_ (page 16) contains a pre-built up-to-date 64 bit Windows binary. It is fully-featured, and if you do not need anything custom, it is likely the fastest way to get going.

**Introduction**

Pre-built binary packages for Windows are available via _Conda_ (page 16) (64-bit version), and all of the prerequisites required for compilation of a fully featured build are also available via that packaging system. This document assumes you will be using Conda Forge as your base, and anything more advanced is beyond the scope of the document.

**Note:**  The AppVeyor build system uses the PDAL project’s configuration on the Conda Forge system. It contains a rich resource of known working examples. See https://github.com/PDAL/PDAL/blob/master/appveyor.yml and https://github.com/PDAL/PDAL/tree/master/scripts/appveyor for inspiration.

**Required Compiler**

PDAL is known to compile on _Visual Studio 2015_ (https://www.visualstudio.com/vs/older-downloads/), and 2013 _might_ work with some source tree adjustments. PDAL makes heavy use of C++11, and a compiler with good support for those features is required.
Prerequisite Libraries

PDAL uses the AppVeyor (https://ci.appveyor.com/project/hobu/pdal/history) continuous integration platform for building and testing itself on Windows. The configuration that PDAL uses is valuable raw materials for configuring your own environment because the PDAL team must keep it up to date with both the Conda (page 16) environment and the Microsoft compiler situation.

You can see the current AppVeyor configuration at https://github.com/PDAL/PDAL/blob/master/appveyor.yml The most interesting bits are the install section, the config.cmd, and the build.cmd scripts. The AppVeyor configuration already has Miniconda installed, and the config.cmd script installs all of PDAL's prerequisites via the command line.

```bash
conda install geotiff laszip nitro curl ^
gdal pcl cmake eigen ninja libgdal ^
zstd numpy xz libxml2 laz-perf qhull ^
sqlite hdf5 tiledb conda-build ninja -y
```

**Note:** The package list here might change over time. The canonical location to learn the prerequisite list for PDAL is the scripts/appveyor/test/build.cmd file in PDAL's source tree.

Fetching the Source

Get the source code for PDAL. Presumably you have GitHub for Windows (https://desktop.github.com/) or something like it. Run a “git shell” and clone the repository into the directory of your choice.

```
c:\dev> git clone https://github.com/PDAL/PDAL.git
```

Switch to the -maintenance branch.

```
c:\dev> git checkout 1.9-maintenance
```

**Note:** PDAL's active development branch is master, and you are welcome to build it, but is not as stable as the major-versioned release branches are likely to be.
Configuration

PDAL uses CMake (http://www.cmake.org) for its build configuration. You will need to install CMake and have it available on your path to configure PDAL.

Invoke your cmake command to configure the PDAL.

```
cmake -G "NMake Makefiles" .
```

A fully-featured build will require more specification of libraries, enabled features, and their locations. There are two places in the source tree for inspiration on this topic.

1. The AppVeyor build configuration
   https://github.com/PDAL/PDAL/blob/master/scripts/appveyor/config.cmd#L26

2. Howard Butler’s example build configuration
   https://github.com/PDAL/PDAL/blob/master/scripts/conda/win64.bat

Note: Placing your command in a .bat file will make for easy reuse.

Building

If you chose NMake Makefiles as your CMake generator, you can invoke the build by calling nmake:

```
nmake /f Makefile
```

If you chose “Visual Studio 14 Win64” as your CMake generator, open PDAL.sln and chose your configuration to build.

Running

After you’ve built the tree, you can run pdal.exe by issuing it

```
c:\dev\pdal\bin\pdal.exe
```

Note: You may need to have your Conda environment active to enable access to PDAL’s dependencies.
Dependencies

PDAL depends on a number of libraries to do its work. You should make sure those dependencies are installed on your system before installing PDAL or use a packaging system that will automatically ensure that prerequisites are satisfied. Packaging system such as apt (https://help.ubuntu.com/lts/serverguide/apt.html) or Conda (https://conda.io/en/latest/) can be used to install dependencies on your system.

Required Dependencies

GDAL (2.2+)

PDAL uses GDAL for spatial reference system description manipulation, and image reading supporting for the NITF driver, and writers oci (page 133) support. In conjunction with GeoTIFF (http://trac.osgeo.org/geotiff), GDAL is used to convert GeoTIFF keys and OGC WKTI SRS description strings into formats required by specific drivers.

Source: https://github.com/OSGeo/gdal
Conda: https://anaconda.org/conda-forge/gdal

GeoTIFF (1.3+)

PDAL uses GeoTIFF in conjunction with GDAL for GeoTIFF key support in the LAS driver. GeoTIFF is typically a dependency of GDAL, so installing GDAL from a package will generally install GeoTIFF as well.

Source: https://github.com/OSGeo/libgeotiff
Conda: https://anaconda.org/conda-forge/geotiff

Note: GDAL surreptitiously embeds a copy of GeoTIFF (http://trac.osgeo.org/geotiff) in its library build but there is no way for you to know this. In addition to embedding libgeotiff, it also strips away the library symbols that PDAL needs, meaning that PDAL can’t simply link against GDAL (http://www.gdal.org). If you are building both of these libraries yourself, make sure you build GDAL using the “External libgeotiff” option, which will prevent the insanity that can ensue on some platforms. Conda Forge (https://anaconda.org/conda-forge/pdal) users, including those using that platform to link and build PDAL themselves, do not need to worry about this issue.
Optional Dependencies

LASzip (Latest package/source recommended)

LASzip (http://laszip.org) is a library with a CMake-based build system that provides periodic compression of ASPRS LAS (http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) data. It is used by the writers.las (page 125) and readers.las (page 71) to provide compressed LAS support:

Source: https://github.com/LASzip/LASzip
Conda: https://anaconda.org/conda-forge/laszip

laz-perf (Latest package/source recommended)

laz-perf provides an alternative LAS compression/decompression engine that may be slightly faster in some circumstances. laz-perf supports fewer LAS point types and versions than does LASzip. It is also used as a compression type for writers.oci (page 133) and writers.sqlite (page 144):

Source: https://github.com/verma/laz-perf/
Conda: https://anaconda.org/conda-forge/laz-perf

libxml2 (2.7+)

libxml2 (http://xmlsoft.org) is used to serialize PDAL dimension descriptions into XML for the database drivers such as writers.oci (page 133), readers.sqlite (page 103), or readers.pgpointcloud (page 89):

Source: http://www.xmlsoft.org/
Conda: https://anaconda.org/conda-forge/libxml2

Plugin Dependencies

PDAL comes with optional plugin stages that require other libraries in order to run. Many of these libraries are licensed in a way incompatible with the PDAL license or they may be commercial products that require purchase.
OCI (10g+)

Obtain the Oracle Instant Client (http://www.oracle.com/technology/tech/oci/instantclient/index.html) and install in a location on your system. Be sure to install both the “Basic” and the “SDK” modules. Set your ORACLE_HOME environment variable system- or user-wide to point to this location so the CMak configuration can find your install. OCI is used by both writers.oci (page 133) and readers.oci (page 86) for Oracle Point Cloud read/write support. In order to obtain the OCI libraries you must register with Oracle.:


Nitro (Requires specific source package)

Nitro is a library that provides NITF (http://en.wikipedia.org/wiki/National_Imagery_Transmission_Format) support for PDAL to write LAS-in-NITF files for writers.nitf (page 130). You must use the specific version of Nitro referenced below for licensing and compatibility reasons.:

Source: http://github.com/hobu/nitro

PCL (1.7.2+)

The Point Cloud Library (PCL) (http://pointclouds.org) is used by the pcl_command, writers.pcd (page 137), readers.pcd (page 88), and filters.pclblock to provide support for various PCL-related operations.:

Source: https://github.com/PointCloudLibrary/pcl
Conda: https://anaconda.org/conda-forge/pcl

TileDB (1.4.1+)

TileDB (https://www.tiledb.io) is an efficient multi-dimensional array management system which introduces a novel on-disk format that can effectively store dense and sparse array data with support for fast updates and reads. It features excellent compression, and an efficient parallel I/O system with high scalability. It is used by writers.tiledb (page 147) and readers.tiledb (page 109).:

Source: https://github.com/TileDB-Inc/TileDB
Conda: https://anaconda.org/conda-forge/tiledb
14.1.3 Errors and Error Handling

Exceptions

PDAL typically throws a `std::runtime_error` for error conditions that is catchable as `pdal::pdal_error`.

PDAL Position on (Non)conformance

PDAL proudly and unabashedly supports formal standards/specifications for file formats. We recognize, however, that in some cases files will not follow a given standard precisely, due to an unclear spec or simply out of carelessness.

When reading files that are not formatted correctly:

- PDAL may try to compensate for the error. This is typically done when as a practical matter the market needs support for well-known or pervasive, but nonetheless “broken”, upstream implementations.
- PDAL may explicitly reject such files. This is typically done where we do not wish to continue to promote or support mistakes that should be fixed upstream.

PDAL will strive to write correctly formatted files. In some cases, however, PDAL may choose to offer as an option the ability to break the standard if, as a practical matter, doing so would significantly aid the market. Such an option would never be the default behavior, however.

For files that are conformant but which lie, such as the extents in the header being wrong, we will generally offer both the ability to propagate the “wrong” information and the ability to helpfully correct it on the fly; the latter is generally our default position.

14.1.4 Metadata

In addition to point data, PDAL stores metadata during the processing of a pipeline. Metadata is stored internally as strings, though the API accepts a variety of types that are automatically converted as necessary. Each item of metadata consists of a name, a description (optional), a value and a type. In addition, each item of metadata can have a list of child metadata values.

Metadata is made available to users of PDAL through a JSON tree. Commands such as `pdal pipeline` (page 32) and `pdal translate` (page 39) provide options to allow the JSON-formatted metadata created by PDAL to be written to a file.
Metadata Nodes

Each item of metadata is stored in an object known as a MetadataNode. Metadata nodes are reference types that can be copied cheaply. Metadata nodes are annotated with the original data type to allow better interpretation of the data. For example, when binary data is stored in a base 64-encoded format, knowing that the data doesn’t ultimately represent a string can allow algorithms to convert it back to its binary representation when desired. Similarly, knowing that data is numeric allows it to be written as a JSON numeric type rather than as a string.

The name of a metadata node is immutable. If you wish to add a copy of metadata (and subchildren) to some node using a different name, you need to call the provided function “clone()”.

A metadata node is added as a child to another node using add(). Usually the type of the data assigned to the metadata node is determined through overloading, but there are instances where this is impossible and the programmer must call a specific function to set the type of the metadata node. Binary data that has been converted to a string by base 64 encoding can be tagged as a such by calling addEncoded(). Programmers can specify the type of a node explicitly by calling addWithTyped(). Currently supported types are: “boolean”, “string”, “float”, “double”, “bounds”, “nonNegativeInteger”, “integer”, “uuid” and “base64Binary”.

Metadata nodes can be presented as lists when transformed to JSON. If multiple nodes with the same name are added to a parent node, those subnodes will automatically be tagged as list nodes and will be enclosed in square brackets. Single nodes can be forced to be treated as JSON lists by calling addList() instead of add() on a parent node.

Metadata and Stages

Stages in PDAL each have a base metadata node. You can retrieve a stage’s metadata node by calling getMetadata(). When a PDAL pipeline is run, its metadata is organized as a list of stage nodes to which subnodes have been added. From within the implementation of a stage, metadata is typically added similarly to the following:

```java
MetadataNode root = getMetadata();
root.add("nodename", "Some string data");
root.add("intlist", 45);
root.add("intlist", 55);
Uuid nullUuid;
MetadataNode pnode("parent");
root.add(pnode);
pnode.add("nulluuidnode", nullUuid);
pnode.addList("num_in_list", 66);
```

If the above code was part of a stage “writers.test”, a transformation to JSON would produce the following output:
14.1.5 Writing with PDAL

**Author**  Bradley Chambers

**Contact**  brad.chambers@gmail.com

**Date**  11/02/2017

This tutorial will describe a complete example of using PDAL C++ objects to write a LAS file. The example will show fetching data from your own data source rather than interacting with a PDAL stage.

**Note:** If you implement your own *Readers* (page 54) that conforms to PDAL’s *pdal::Stage* (page 553), you can implement a simple read-filter-write pipeline using *Pipeline* (page 45) and not have to code anything explicit yourself.
Includes

First, our code.

```cpp
#include <pdal/PointView.hpp>
#include <pdal/PointTable.hpp>
#include <pdal/Dimension.hpp>
#include <pdal/Options.hpp>
#include <pdal/StageFactory.hpp>

#include <io/BufferReader.hpp>

#include <vector>

void fillView(pdal::PointViewPtr view)
{
    struct Point
    {
        double x;
        double y;
        double z;
    };

    for (int i = 0; i < 1000; ++i)
    {
        Point p;

        p.x = -93.0 + i*0.001;
        p.y = 42.0 + i*0.001;
        p.z = 106.0 + i;

        view->setField(pdal::Dimension::Id::X, i, p.x);
        view->setField(pdal::Dimension::Id::Y, i, p.y);
        view->setField(pdal::Dimension::Id::Z, i, p.z);
    }
}

int main(int argc, char* argv[])
{
    using namespace pdal;

    Options options;
    options.add("filename", "myfile.las");

    PointTable table;
    table.layout()->registerDim(Dimension::Id::X);
```

(continues on next page)
Table layout()->registerDim(Dimension::Id::Y);
Table layout()->registerDim(Dimension::Id::Z);

PointViewPtr view(new PointView(table));

fillView(view);

BufferReader reader;
reader.addView(view);

StageFactory factory;

// Set second argument to 'true' to let factory take ownership of
// stage and facilitate clean up.
Stage *writer = factory.createStage("writers.las");

writer->setInput(reader);
writer->setOptions(options);
writer->prepare(table);
writer->execute(table);

Take a closer look. We will need to include several PDAL headers.

```cpp
#include <pdal/PointView.hpp>
#include <pdal/PointTable.hpp>
#include <pdal/Dimension.hpp>
#include <pdal/Options.hpp>
#include <pdal/StageFactory.hpp>
#include <io/BufferReader.hpp>
```

`BufferReader` will not be required by all users. Here is it used to populate a bare `PointBuffer`. This will often be accomplished by a `Reader` stage.

Instead of directly including headers for individual stages, e.g., `LasWriter`, we rely on the `StageFactory` which has the ability to query available stages at runtime and return pointers to the created stages.

We proceed by providing a mechanism for generating dummy data for the x, y, and z dimensions.

```cpp
void fillView(pdal::PointViewPtr view)
{
    struct Point
    {
```
double x;
double y;
double z;
}

for (int i = 0; i < 1000; ++i)
{
    Point p;
    p.x = -93.0 + i*0.001;
    p.y = 42.0 + i*0.001;
    p.z = 106.0 + i;
    view->setField(pdal::Dimension::Id::X, i, p.x);
    view->setField(pdal::Dimension::Id::Y, i, p.y);
    view->setField(pdal::Dimension::Id::Z, i, p.z);
}

int main(int argc, char* argv[])
{
    using namespace pdal;
    Options options;
    options.add("filename", "myfile.las");
    PointTable table;

    finally, the main code which creates the dummy data, puts it into a BufferReader and sends it to a writer.

    int main(int argc, char* argv[])
    {
        using namespace pdal;
        Options options;
        options.add("filename", "myfile.las");

        PointTable table;
        table.layout()->registerDim(Dimension::Id::X);
        table.layout()->registerDim(Dimension::Id::Y);
        table.layout()->registerDim(Dimension::Id::Z);

        PointViewPtr view(new PointView(table));

        fillView(view);
BufferReader reader;
reader.addView(view);

StageFactory factory;

// Set second argument to 'true' to let factory take ownership of
// stage and facilitate clean up.
Stage *writer = factory.createStage("writers.las");

writer->setInput(reader);
writer->setOptions(options);
writer->prepare(table);
writer->execute(table);
}

Compiling and running the program

Note: Refer to Compilation (page 456) for information on how to build PDAL.

To build this example, simply copy the files tutorial.cpp and CMakeLists.txt from the examples/writing directory of the PDAL source tree.

```cmake
cmake_minimum_required(VERSION 3.6)
project(WritingTutorial)

find_package(PDAL 2.0.0 REQUIRED CONFIG)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(tutorial tutorial.cpp)

target_link_libraries(tutorial PRIVATE ${PDAL_LIBRARIES})
target_include_directories(tutorial PRIVATE
  ${PDAL_INCLUDE_DIRS}
  ${PDAL_INCLUDE_DIRS}/pdal)
```

Note: Refer to CMake (page 499) for an explanation of the basic CMakeLists.

Begin by configuring your project using CMake (shown here on Unix) and building using make.

14.1. Development
After the project is built, you can run it by typing:

```bash
$ ./tutorial
```

### 14.1.6 Writing a filter

PDAL can be extended through the development of filter functions.

**See also:**

For more on filters and their role in PDAL, and their lifecycle please refer to *PDAL Architecture Overview* (page 445).

Every filter stage in PDAL is implemented as a plugin (sometimes referred to as a “driver”). Filters native to PDAL, such as `filters.ferry` (page 208), are implemented as static filters and are statically linked into the PDAL library. Filters that require extra/optional dependencies, or are external to the core PDAL codebase altogether, such as `filters.python` (page 276), are implemented as shared filters, and are built as individual shared libraries, discoverable by PDAL at runtime.

In this tutorial, we will give a brief example of a filter, with notes on how to make it static or shared.

**The header**

First, we provide a full listing of the filter header.

```cpp
// MyFilter.hpp

#pragma once

#include <pdal/pdal_internal.hpp>
#include <pdal/Filter.hpp>

namespace pdal {

class PDAL_DLL MyFilter : public Filter
{

(continues on next page)
```
This header should be relatively straightforward, but we will point out one method that must be declared for the plugin interface to be satisfied.

```
std::string getName() const;
```

In many instances, you should be able to copy this header template verbatim, changing only the filter class name, includes, and member functions/variables as required by your implementation.

## The source

Again, we start with a full listing of the filter source.

```
// MyFilter.cpp

#include "MyFilter.hpp"

#include <pdal/pdal_internal.hpp>

namespace pdal
{

static PluginInfo const s_info
{
  "filters.name",
```

(continues on next page)
"My awesome filter",
"http://link/to/documentation"
);

CREATE_SHARED_STAGE(MyFilter, s_info)

std::string MyFilter::getName() const { return s_info.name; }

void MyFilter::addArgs(ProgramArgs& args)
{
    args.add("param", "Some parameter", m_value, 1.0);
}

void MyFilter::addDimensions(PointLayoutPtr layout)
{
    layout->registerDim(Dimension::Id::Intensity);
    m_myDimension = layout->registerOrAssignDim("MyDimension",
        Dimension::Type::Unsigned8);
}

PointViewSet MyFilter::run(PointViewPtr input)
{
    PointViewSet viewSet;
    viewSet.insert(input);
    return viewSet;
}

} // namespace pdal

For your filter to be available to PDAL at runtime, it must adhere to the PDAL plugin interface. As a convenience, we provide macros to do just this.

We begin by creating a PluginInfo struct containing three identifying elements - the filter name, description, and a link to documentation.

static PluginInfo const s_info
{
    "filters.name",
    "My awesome filter",
    "http://link/to/documentation"
};

PDAL requires that filter names always begin with filters., and end with a string that uniquely identifies the filter. The description will be displayed to users of the PDAL CLI (pdal --drivers). When making a shared plugin, the name of the shared library must correspond with the name of the filter provided here. The name of the generated shared object
must be

```
libpdal_plugin_filter_<filter name>.-<shared library extension>
```

Next, we pass the following to the `CREATE_SHARED_STAGE` macro, passing in the name of the stage and the `PluginInfo` struct.

`CREATE_SHARED_STAGE(MyFilter, s_info)`

To create a static stage, we simply change `CREATE_SHARED_STAGE` to `CREATE_STATIC_STAGE`.

Finally, we implement a method to get the plugin name, which is primarily used by the PDAL CLI when using the `--drivers` or `--options` arguments.

```
std::string MyFilter::getName() const { return s_info.name; }
```

Now that the filter has implemented the proper plugin interface, we will begin to implement some methods that actually implement the filter. The `addArgs()` method is used to register and bind any provided options to the stage. Here, we get the value of `param`, if provided, else we populate `m_value` with the default value of `1.0`. Option names, descriptions, and default values specified in `addArgs()` will be displayed via the PDAL CLI with the `--options` argument.

```
void MyFilter::addArgs(ProgramArgs& args)
{
    args.add("param", "Some parameter", m_value, 1.0);
}
```

In `addDimensions()` we make sure that the known `Intensity` dimension is registered. We can also add a custom dimension, `MyDimension`, which will be populated within `run()`.

```
void MyFilter::addDimensions(PointLayoutPtr layout)
{
    layout->registerDim(Dimension::Id::Intensity);
    m_myDimension = layout->registerOrAssignDim("MyDimension",
                                              Dimension::Type::Unsigned8);
}
```

Finally, we define `run()`, which takes as input a `PointViewPtr` and returns a `PointViewSet`. It is here that we can transform existing dimensions, add data to new dimensions, or selectively add/remove individual points.

We suggest you take a closer look at our existing filters to get an idea of the power of the `Filter` stage and inspiration for your own filters!
Compilation

Set up a CMakeLists.txt file to compile your filter against PDAL:

```cmake
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8.12)
project(FilterTutorial)

find_package(PDAL 1.9.0 REQUIRED CONFIG)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_library(pdal_plugin_filter_myfilter SHARED MyFilter.cpp)
target_link_libraries(pdal_plugin_filter_myfilter PRIVATE ${PDAL_LIBRARIES})
target_include_directories(pdal_plugin_filter_myfilter PRIVATE ${PDAL_INCLUDE_DIRS})
target_link_directories(pdal_plugin_filter_myfilter PRIVATE ${PDAL_LIBRARY_DIRS})
```

Note: CMakeLists.txt contents may vary slightly depending on your project requirements, operating system, and compiler.

14.1.7 Writing a kernel

**Author**  Bradley Chambers  
**Contact**  brad.chambers@gmail.com  
**Date**  11/02/2017

PDAL’s command-line application can be extended through the development of kernel functions. In this tutorial, we will give a brief example.

The header

First, we provide a full listing of the kernel header.

```cpp
// MyKernel.hpp

#pragma once

#include <pdal/Kernel.hpp>

(continues on next page)```
#include <string>

namespace pdal
{

class PDAL_DLL MyKernel : public Kernel
{
public:
    MyKernel();

    std::string getName() const;
    int execute(); // override

private:
    void addSwitches(ProgramArgs& args);

    std::string m_input_file;
    std::string m_output_file;
};
} // namespace pdal

As with other plugins, the MyKernel class needs to return a name.

std::string getName() const;

The source

Again, we start with a full listing of the kernel source.

// MyKernel.cpp
#include "MyKernel.hpp"

#include <pdal/Filter.hpp>
#include <pdal/Kernel.hpp>
#include <pdal/Options.hpp>
#include <pdal/PointTable.hpp>

#include <memory>
#include <string>

namespace pdal {

static PluginInfo const s_info
{
    "kernels.mykernel",
    "MyKernel",
    "http://link/to/documentation"
};

CREATE_SHARED_KERNEL(MyKernel, s_info);

std::string MyKernel::getName() const { return s_info.name; }

MyKernel::MyKernel() : Kernel()
{}

void MyKernel::addSwitches(ProgramArgs& args)
{
    args.add("input,i", "Input filename", m_input_file).
       ->setPositional();
    args.add("output,o", "Output filename", m_output_file).
       ->setPositional();
}

int MyKernel::execute()
{
    PointTable table;

    Stage& reader = makeReader(m_input_file, "readers.las");

    // Options should be added in the call to makeFilter, makeReader, 
    // or makeWriter so that the system can override them with those 
    // provided on the command line when applicable.
    Options filterOptions;
    filterOptions.add("step", 10);
    Stage& filter = makeFilter("filters.decimation", reader, 
       ->filterOptions);

    Stage& writer = makeWriter(m_output_file, filter, "writers.text 
       ->");
    writer.prepare(table);
    writer.execute(table);

    return 0;
}

} // namespace pdal
In your kernel implementation, you will use a macro defined in pdal_macros. This macro registers the plugin with the PluginManager.

```
CREATE_SHARED_KERNEL(MyKernel, s_info);
```

To build up a processing pipeline in this example, we need to create two objects: the `pdal::PointTable`.

```cpp
int MyKernel::execute()
{
    PointTable table;

    Stage& reader = makeReader(m_input_file, "readers.las");

    // Options should be added in the call to makeFilter, makeReader, 
    // or makeWriter so that the system can override them with those 
    // provided on the command line when applicable.
    Options filterOptions;
    filterOptions.add("step", 10);
    Stage& filter = makeFilter("filters.decimation", reader, filterOptions);

    Stage& writer = makeWriter(m_output_file, filter, "writers.text");
    writer.prepare(table);
    writer.execute(table);

    return 0;
}
```

To implement the actual kernel logic we implement execute(). In this case, the kernel reads a las file, decimates the data (eliminates some points) and writes the result to a text file. The base kernel class provides functions (makeReader, makeFilter, makeWriter) to create stages with options as desired. The pipeline that has been created can be run by preparing and executing the last stage in the pipeline.

When compiled, a dynamic library file will be created; in this case, `libpdal_plugin_kernel_mykernel.dylib`

Put this file in whatever directory `PDAL_DRIVER_PATH` is pointing to. Then, if you run `pdal --drivers`, you should see `mykernel` listed in the possible commands.

To run this kernel, you would use `pdal mykernel -i <input las file> -o <output text file>`.
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

Compilation

Set up a CMakeLists.txt file to compile your kernel against PDAL:

```cmake
# CMakeLists.txt

cmake_minimum_required(VERSION 2.8.12)
project(KernelTutorial)

find_package(PDAL 2.0.0 REQUIRED CONFIG)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_library(pdal_plugin_kernel_mykernel SHARED MyKernel.cpp)
target_link_libraries(pdal_plugin_kernel_mykernel PRIVATE ${PDAL_LIBRARY_DIRS})
target_include_directories(pdal_plugin_kernel_mykernel PRIVATE ${PDAL_INCLUDE_DIRS})
```

14.1.8 Writing a reader

Authors  Bradley Chambers, Scott Lewis

Contact  brad.chambers@gmail.com

Date  11/02/2017

PDAL's command-line application can be extended through the development of reader functions. In this tutorial, we will give a brief example.

The header

First, we provide a full listing of the reader header.

```cpp
// MyReader.hpp

#pragma once

#include <pdal/PointView.hpp>
#include <pdal/Reader.hpp>
#include <pdal/util/IStream.hpp>

namespace pdal {
    class MyReader : public Reader
```
public:
    MyReader() : Reader() {};
    std::string getName() const;

private:
    std::unique_ptr<ILeStream> m_stream;
    point_count_t m_index;
    double m_scale_z;

    virtual void addDimensions(PointLayoutPtr layout);
    virtual void addArgs(ProgramArgs& args);
    virtual void ready(PointTableRef table);
    virtual point_count_t read(PointViewPtr view, point_count_t count);
    virtual void done(PointTableRef table);
};

std::unique_ptr<ILeStream> m_stream;
point_count_t m_index;
double m_scale_z;

m_stream is used to process the input, while m_index is used to track the index of the records. m_scale_z is specific to MyReader, and will be described later.

virtual void addDimensions(PointLayoutPtr layout);
virtual void addArgs(ProgramArgs& args);
virtual void ready(PointTableRef table);
virtual point_count_t read(PointViewPtr view, point_count_t count);
virtual void done(PointTableRef table);

Various other override methods for the stage. There are a few others that could be overridden, which will not be discussed in this tutorial.

Note: See ./include/pdal/Reader.hpp of the source tree for more methods that a reader can override or implement.
The source

Again, we start with a full listing of the reader source.

```cpp
// MyReader.cpp

#include "MyReader.hpp"
#include <pdal/util/ProgramArgs.hpp>

namespace pdal {

static PluginInfo const s_info {
    "readers.myreader",
    "My Awesome Reader",
    "http://link/to/documentation"
};

CREATE_SHARED_STAGE(MyReader, s_info)

std::string MyReader::getName() const { return s_info.name; }

void MyReader::addArgs(ProgramArgs& args) {
    args.add("z_scale", "Z Scaling", m_scale_z, 1.0);
}

void MyReader::addDimensions(PointLayoutPtr layout) {
    layout->registerDim(Dimension::Id::X);
    layout->registerDim(Dimension::Id::Y);
    layout->registerDim(Dimension::Id::Z);
    layout->registerOrAssignDim("MyData", Dimension::Type::Unsigned64);
}

void MyReader::ready(PointTableRef) {
    m_index = 0;
    SpatialReference ref("EPSG:4385");
    setSpatialReference(ref);
}

template <typename T>
T convert(const StringList& s, const std::string& name, size_t fieldno) {

(continues on next page)
T output;
bool bConverted = Utils::fromString(s[fieldno], output);
if (!bConverted)
{
    std::stringstream oss;
    oss << "Unable to convert " << name << ", " << s[fieldno] << ", to double";
    throw pdal_error(oss.str());
}
return output;
}

point_count_t MyReader::read(PointViewPtr view, point_count_t count)
{
    PointLayoutPtr layout = view->layout();
    PointId nextId = view->size();
    PointId idx = m_index;
    point_count_t numRead = 0;

    m_stream.reset(new ILeStream(m_filename));

    size_t HEADERSIZE(1);
    size_t skip_lines((std::max)(HEADERSIZE, (size_t)m_index));
    size_t line_no(1);
    for (std::string line; std::getline(*m_stream->stream(), line); line_no++)
    {
        if (line_no <= skip_lines)
        {
            continue;
        }
        // MyReader format: X::Y::Z::Data
        StringList s = Utils::split2(line, ':');

        unsigned long u64(0);
        if (s.size() != 4)
        {
            std::stringstream oss;
            oss << "Unable to split proper number of fields. Expected 4, got "
            
(continues on next page)
std::string name("X");
view->setField(Dimension::Id::X, nextId, convert<double>(s, name, 0));

name = "Y";
view->setField(Dimension::Id::Y, nextId, convert<double>(s, name, 1));

name = "Z";
double z = convert<double>(s, name, 2) * m_scale_z;
view->setField(Dimension::Id::Z, nextId, z);

name = "MyData";
view->setField(layout->findProprietaryDim(name),
nextId,
    convert<unsigned int>(s, name, 3));

nextId++;
if (m_cb)
    m_cb(*view, nextId);

m_index = nextId;
numRead = nextId;

    return numRead;
}

void MyReader::done(PointTableRef)
{
    m_stream.reset();
}

} //namespace pdal

In your reader implementation, you will use a macro to create the plugin. This macro registers
the plugin with the PDAL PluginManager. In this case, we are declaring this as a SHARED
stage, meaning that it will be loaded at runtime instead of being linked to the main PDAL
installation. The macro is supplied with the class name of the plugin and a PluginInfo object.
The PluginInfo objection includes the name of the plugin, a description, and a link to
documentation.

When making a shared plugin, the name of the shared library must correspond with the name
of the reader provided here. The name of the generated shared object must be

```
libpdal_plugin_reader_<reader name>_<shared library extension>
```

```
static PluginInfo const s_info {
    "readers.myreader",
    "My Awesome Reader",
    "http://link/to/documentation"
};

CREATE_SHARED_STAGE(MyReader, s_info)
```

This method will process options for the reader. In this example, we are setting the z_scale value to a default of 1.0, indicating that the Z values we read should remain as-is. (In our reader, this could be changed if, for example, the Z values in the file represented mm values, and we want to represent them as m in the storage model). addArgs will bind values given for the argument to the m_scale_z variable of the stage.

```
void MyReader::addArgs(ProgramArgs& args) {
    args.add("z_scale", "Z Scaling", m_scale_z, 1.0);
}
```

This method registers the various dimensions the reader will use. In our case, we are using the X, Y, and Z built-in dimensions, as well as a custom dimension MyData.

```
void MyReader::addDimensions(PointLayoutPtr layout) {
    layout->registerDim(Dimension::Id::X);
    layout->registerDim(Dimension::Id::Y);
    layout->registerDim(Dimension::Id::Z);
    layout->registerOrAssignDim("MyData", Dimension::Type::Unsigned64);
}
```

This method is called when the Reader is ready for use. It will only be called once, regardless of the number of PointViews that are to be processed.

```
void MyReader::ready(PointTableRef) {
    m_index = 0;
    SpatialReference ref("EPSG:4385");
    setSpatialReference(ref);
}
```

This is a helper function, which will convert a string value into the type specified when it’s called. In our example, it will be used to convert strings to doubles when reading from the
This method is the main processing method for the reader. It takes a pointer to a PointView which we will build as we read from the file. We initialize some variables as well, and then reset the input stream with the filename used for the reader. Note that in other readers, the contents of this method could be very different depending on the format of the file being read, but this should serve as a good start for how to build the PointView object.

```cpp
{
    PointLayoutPtr layout = view->layout();
    PointId nextId = view->size();
    PointId idx = m_index;
    point_count_t numRead = 0;
}
```

In preparation for reading the file, we prepare to skip some header lines. In our case, the header is only a single line.

```cpp
size_t HEADERSIZE(1);
size_t skip_lines((std::max)(HEADERSIZE, (size_t)m_index));
```

Here we begin our main loop. In our example file, the first line is a header, and each line thereafter is a single point. If the file had a different format the method of looping and reading would have to change as appropriate. We make sure we are skipping the header lines here before moving on.

```cpp
size_t line_no(1);
for (std::string line; std::getline(*m_stream->stream(), line); ++line_no++)
{
    if (line_no <= skip_lines)
```
Here we take the line we read in the for block header, split it, and make sure that we have the proper number of fields.

```cpp
// MyReader format: X::Y::Z::Data
StringList s = Utils::split2(line, ':');

unsigned long u64(0);
if (s.size() != 4) {
    std::stringstream oss;
    oss << "Unable to split proper number of fields. Expected 4, got " << s.size();
    throw pdal_error(oss.str());
}
```

Here we take the values we read and put them into the PointView object. The X and Y fields are simply converted from the file and put into the respective fields. MyData is done likewise with the custom dimension we defined. The Z value is read, and multiplied by the scale_z option (defaulted to 1.0), before the converted value is put into the field.

When putting the value into the PointView object, we pass in the Dimension that we are assigning it to, the ID of the point (which is incremented in each iteration of the loop), and the dimension value.

```cpp
std::string name("X");
view->setField(Dimension::Id::X, nextId, convert<double>(s, name, 0));

name = "Y";
view->setField(Dimension::Id::Y, nextId, convert<double>(s, name, 1));

name = "Z";
double z = convert<double>(s, name, 2) * m_scale_z;
view->setField(Dimension::Id::Z, nextId, z);

name = "MyData";
view->setField(layout->findProprietaryDim(name), nextId,
```

Finally, we increment the nextId and make a call into the progress callback if we have one with our nextId. After the loop is done, we set the index and number read, and return that value as the number of points read. This could differ in cases where we read multiple streams, but that
won’t be covered here.

```cpp
nextId++;  
if (m_cb)  
    m_cb(*view, nextId);  
}  
  m_index = nextId;  
numRead = nextId;
```

When the read method is finished, the done method is called for any cleanup. In this case, we simply make sure the stream is reset.

```cpp
void MyReader::done(PointTableRef)
{
  m_stream.reset();
}
```

### Compiling and Usage

The MyReader.cpp code can be compiled. For this example, we’ll use cmake. Here is the CMakeLists.txt file we will use:

```cmake
cmake_minimum_required(VERSION 2.8.12)  
project(ReaderTutorial)  
find_package(PDAL 2.0 REQUIRED CONFIG)  
set(CMAKE_CXX_STANDARD 11)  
set(CMAKE_CXX_STANDARD_REQUIRED ON)  
add_library(pdal_plugin_reader_myreader SHARED MyReader.cpp)  
target_link_libraries(pdal_plugin_reader_myreader PRIVATE ${PDAL_LIBRARIES})  
target_include_directories(pdal_plugin_reader_myreader PRIVATE ${PDAL_INCLUDE_DIRS})  
target_link_directories(pdal_plugin_reader_myreader PRIVATE ${PDAL_LIBRARY_DIRS})
```

If this file is in the directory containing MyReader.hpp and MyReader.cpp, simply run `cmake .`, followed by `make`. This will generate a file called `libpdal_plugin_reader_myreader.dylib`.

Put this dylib file into the directory pointed to by `PDAL_DRIVER_PATH`, and then when you run `pdal --drivers`, you should see an entry for `readers.myreader`.

To test the reader, we will put it into a pipeline and output a text file.

Please download the `pipeline-myreader.json` (https://github.com/PDAL/PDAL/blob/master/examples/writing-reader/pipeline-
myreader.json?raw=true) and test-reader-input.txt
files.

In the directory with those two files, run pdal pipeline pipeline-myreader.json. You should have an output file called output.txt, which will have the same data as in the input file, except in a CSV style format, and with the Z values scaled by .001.

14.1.9 Writing a writer

Authors  Bradley Chambers, Scott Lewis
Contact  brad.chambers@gmail.com
Date  10/26/2016

PDAL’s command-line application can be extended through the development of writer functions. In this tutorial, we will give a brief example.

The header

First, we provide a full listing of the writer header.

```cpp
// MyWriter.hpp
#pragma once
#include <pdal/Writer.hpp>
#include <string>
namespace pdal{

typedef std::shared_ptr<std::ostream> FileStreamPtr;

class MyWriter : public Writer
{
public:
    MyWriter()
    {}

    std::string getName() const;

private:
    virtual void addArgs(ProgramArgs& args);
}
```

(continues on next page)
virtual void initialize();
virtual void ready(PointTableRef table);
virtual void write(const PointViewPtr view);
virtual void done(PointTableRef table);

std::string m_filename;
std::string m_newline;
std::string m_datafield;
int m_precision;
FileStreamPtr m_stream;
Dimension::Id m_dataDim;
}

// namespace pdal

In your MyWriter class, you will declare the necessary methods and variables needed to make the writer work and meet the plugin specifications.

typedef std::shared_ptr<std::ostream> FileStreamPtr;

FileStreamPtr is defined to make the declaration of the stream easier to manage later on.

std::string getName() const;

Every stage must return a unique name.

virtual void addArgs(ProgramArgs& args);
virtual void initialize();
virtual void ready(PointTableRef table);
virtual void write(const PointViewPtr view);
virtual void done(PointTableRef table);

These methods are used during various phases of the pipeline. There are also more methods, which will not be covered in this tutorial.

std::string m_filename;
std::string m_newline;
std::string m_datafield;
int m_precision;

FileStreamPtr m_stream;
Dimension::Id m_dataDim;

These are variables our Writer will use, such as the file to write to, the newline character to use, the name of the data field to use to write the MyData field, precision of the double outputs, the
output stream, and the dimension that corresponds to the data field for easier lookup.
As mentioned, there can be additional configurations done as needed.

The source

We will start with a full listing of the writer source.

```cpp
// MyWriter.cpp

#include "MyWriter.hpp"
#include <pdal/util/FileUtils.hpp>
#include <pdal/util/ProgramArgs.hpp>

namespace pdal
{
  static PluginInfo const s_info
  {
    "writers.mywriter",
    "My Awesome Writer",
    "http://path/to/documentation"
  };

  CREATE_SHARED_STAGE(MyWriter, s_info);

  std::string MyWriter::getName() const { return s_info.name; }

  struct FileStreamDeleter
  {
    template <typename T>
    void operator()(T* ptr)
    {
      if (ptr)
      {
        ptr->flush();
        FileUtils::closeFile(ptr);
      }
    }
  };

  void MyWriter::addArgs(ProgramArgs& args)
  {
    // setPositionional() Makes the argument required.
    args.add("filename", "Output filename", m_filename).
    ->setPositionational();
    args.add("newline", "Line terminator", m_newline, "\n");
  }
}
```

(continues on next page)
args.add("datafield", "Data field", m_datafield, "UserData");
    args.add("precision", "Precision", m_precision, 3);
}

void MyWriter::initialize()
{
    m_stream = FileStreamPtr(FileUtils::createFile(m_filename, true),
                            FileStreamDeleter());
    if (!m_stream)
    {
        std::stringstream out;
        out << "writers.mywriter couldn't open '" << m_filename << "' for output.";
        throw pdal_error(out.str());
    }
}

void MyWriter::ready(PointTableRef table)
{
    m_stream->precision(m_precision);
    *m_stream << std::fixed;

    Dimension::Id d = table.layout()->findDim(m_datafield);
    if (d == Dimension::Id::Unknown)
    {
        std::ostringstream oss;
        oss << "Dimension not found with name '"" << m_datafield << "'");
        throw pdal_error(oss.str());
    }

    m_dataDim = d;

    *m_stream << "#X:Y:Z:MyData" << m_newline;
}

void MyWriter::write(PointViewPtr view)
{
    for (PointId idx = 0; idx < view->size(); ++idx)
    {
        double x = view->getFieldAs<double>(Dimension::Id::X, idx);
        double y = view->getFieldAs<double>(Dimension::Id::Y, idx);
        double z = view->getFieldAs<double>(Dimension::Id::Z, idx);
        unsigned int myData = 0;
}
if (!m_datafield.empty()) {
    myData = (int)(view->getFieldAs<double>(m_dataDim, idx) + 0.5);
}

*m_stream << x << ":" << y << ":" << z << ":" << myData << m_newline;
}

void MyWriter::done(PointTableRef)
{
    m_stream.reset();
}

In the writer implementation, we will use a macro defined in pdal_macros, which is included in the include chain we are using.

static PluginInfo const s_info
{
    "writers.mywriter",
    "My Awesome Writer",
    "http://path/to/documentation"
};

CREATE_SHARED_STAGE(MyWriter, s_info);

Here we define a struct with information regarding the writer, such as the name, a description, and a path to documentation. We then use the macro to create a SHARED stage, which means it will be external to the main PDAL installation. When using the macro, we specify the name of the Stage and the PluginInfo struct we defined earlier.

When making a shared plugin, the name of the shared library must correspond with the name of the writer provided here. The name of the generated shared object must be

:: libpdal_plugin_writer_<writer name>_<shared library extension>

struct FileStreamDeleter
{
    template <typename T>
    void operator()(T* ptr)
    {
        if (ptr)
            ...
    }

(continues on next page)
This struct is used for helping with the FileStreamPtr for cleanup.

```cpp
void MyWriter::addArgs(ProgramArgs& args)
{
    // setPositional() Makes the argument required.
    args.add("filename", "Output filename", m_filename).
    setPositionional();
    args.add("newline", "Line terminator", m_newline, \\
             \"\n\");
    args.add("datafield", "Data field", m_datafield, "UserData");
    args.add("precision", "Precision", m_precision, 3);
}
```

This method defines the arguments the writer provides and binds them to private variables.

```cpp
void MyWriter::initialize()
{
    m_stream = FileStreamPtr(FileUtils::createFile(m_filename, true),
                             FileStreamDeleter());
    if (!m_stream)
    {
        std::stringstream out;
        out << "writers.mywriter couldn't open '" << m_filename << \\
             "' for output."
        throw pdal_error(out.str());
    }
}
```

This method initializes our file stream in preparation for writing.

```cpp
void MyWriter::ready(PointTableRef table)
{
    m_stream->precision(m_precision);
    *m_stream << std::fixed;
    Dimension::Id d = table.layout()->findDim(m_datafield);
    if (d == Dimension::Id::Unknown)
    {
        std::ostringstream oss;
        oss << "Dimension not found with name '" << m_datafield << \\
             "'";
        (continues on next page)
The ready method is used to prepare the writer for any number of PointViews that may be passed in. In this case, we are setting the precision for our double writes, looking up the dimension specified as the one to write into MyData, and writing the header of the output file.

This method is the main method for writing. In our case, we are writing a very simple file, with data in the format of X:Y:Z:MyData. We loop through each index in the PointView, and for each one we take the X, Y, and Z values, as well as the value for the specified MyData dimension, and write this to the output file. In particular, note the reading of MyData; in our case, MyData is an integer, but the field we are reading might be a double. Converting from double to integer is done via truncation, not rounding, so by adding .5 before making the conversion will ensure rounding is done properly.

Note that in this case, the output format is pretty simple. For more complex outputs, you may need to generate helper methods (and possibly helper classes) to help generate the proper output. The key is reading in the appropriate values from the PointView, and then writing those in whatever necessary format to the output stream.
This method is called when the writing is done. In this case, it simply cleans up the output stream by resetting it.

### Compiling and Usage

To compile this reader, we will use cmake. Here is the CMakeLists.txt file we will use for this process:

```cmake
cmake_minimum_required(VERSION 2.8.12)
project(WriterTutorial)

find_package(PDAL 2.0.0 REQUIRED CONFIG)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_library(pdal_plugin_writer_mywriter SHARED MyWriter.cpp)
target_link_libraries(pdal_plugin_writer_mywriter PRIVATE ${PDAL_LIBRARIES})
target_link_directories(pdal_plugin_writer_mywriter PRIVATE ${PDAL_LIBRARY_DIRS})
target_include_directories(pdal_plugin_writer_mywriter PRIVATE ${PDAL_INCLUDE_DIRS})
```

If this file is in the directory with the MyWriter.hpp and MyWriter.cpp files, simply run cmake . followed by make. This will generate a file called libpdal_plugin_writer_mywriter.dylib.

Put this dylib file into the directory pointed to by PDAL_DRIVER_PATH, and then when you run pdal --drivers, you will see an entry for writers.mywriter.

To test the writer, we will put it into a pipeline and read in a LAS file and covert it to our output format. For this example, use interesting.las (https://github.com/PDAL/PDAL/blob/master/test/data/interesting.las?raw=true), and run it through pipeline-mywriter.json (https://github.com/PDAL/PDAL/blob/master/examples/writing-writer/pipeline-mywriter.json?raw=true).

If those files are in the same directory, you would just run the command pdal pipeline pipeline-mywriter.json, and it will generate an output file called output.txt, which will be in the proper format. From there, if you wanted, you could run that output file through the MyReader that was created in the previous tutorial, as well.
This tutorial will explain how to use PDAL in your own projects using CMake. A more complete, working example can be found [here](page 469).

**Note:** We assume you have either *built* or *installed* (page 456) PDAL.

### Basic CMake configuration

Begin by creating a file named CMakeLists.txt that contains:

```cmake
# Basic CMake configuration
# (Page 1)

cmake_minimum_required(VERSION 2.8)
project(MY_PDAL_PROJECT)
find_package(PDAL 1.0.0 REQUIRED CONFIG)
include_directories(${PDAL_INCLUDE_DIRS})
link_directories(${PDAL_LIBRARY_DIRS})
add_definitions(${PDAL_DEFINITIONS})
set(CMAKE_CXX_FLAGS "-std=c++11")
add_executable(tutorial tutorial.cpp)
target_link_libraries(tutorial PRIVATE ${PDAL_LIBRARIES})
```

### CMakeLists explained

- **cmake_minimum_required(VERSION 2.8.12)**
  
  The *cmake_minimum_required* command specifies the minimum required version of CMake. We use some recent additions to CMake in PDAL that require version 2.8.12.

- **project(MY_PDAL_PROJECT)**
  
  The CMake *project* command names your project and sets a number of useful CMake variables.

- **find_package(PDAL 1.0.0 REQUIRED CONFIG)**
  
  We next ask CMake to locate the PDAL package, requiring version 1.0.0 or higher.
include_directories(${PDAL_INCLUDE_DIRS})
link_directories(${PDAL_LIBRARY_DIRS})
add_definitions(${PDAL_DEFINITIONS})

If PDAL is found, the following variables will be set:

- **PDAL_FOUND**: set to 1 if PDAL is found, otherwise unset
- **PDAL_INCLUDE_DIRS**: set to the paths to PDAL installed headers and the dependency headers
- **PDAL_LIBRARIES**: set to the file names of the built and installed PDAL libraries
- **PDAL_LIBRARY_DIRS**: set to the paths where PDAL libraries and 3rd party dependencies reside
- **PDAL_VERSION**: the detected version of PDAL
- **PDAL_DEFINITIONS**: list the needed preprocessor definitions and compiler flags

set(CMAKE_CXX_FLAGS "-std=c++11")

We haven’t quite implemented the setting of **PDAL_DEFINITIONS** within the **PDALConfig.cmake** file, so for now you should specify the c++11 compiler flag, as we use it extensively throughout PDAL.

add_executable(tutorial tutorial.cpp)

We use the **add_executable** command to tell CMake to create an executable named **tutorial** from the source file **tutorial.cpp**.

target_link_libraries(tutorial PRIVATE ${PDAL_LIBRARIES})

We assume that the tutorial executable makes calls to PDAL functions. To make the linker aware of the PDAL libraries, we use **target_link_libraries** to link **tutorial** against the **PDAL_LIBRARIES**.

### Compiling the project

Make a **build** directory, where compilation will occur:

$ cd /PATH/TO/MY/PDAL/PROJECT
$ mkdir build

Run cmake from within the build directory:

$ cd build
$ cmake ..
Now, build the project:

```
$ make
```

The project is now built and ready to run:

```
$ ./tutorial
```

### 14.2 Project

Project resources, such as how to update the docs, where the test frameworks are, who develops the software, and conventions to use when developing new code can be found in this section.

#### 14.2.1 Coding Conventions

To the extent possible and reasonable, we value consistency of source code formatting, class and variable naming, and so forth. Please follow existing code, rather than introducing your own (of course, better) formatting or change existing code unless you’re changing behavior.

This note lists some such conventions that we would like to follow, where it makes sense to do so.

**Source Formatting**

We use astyle ([http://astyle.sourceforge.net](http://astyle.sourceforge.net)) as a tool to reformat C++ source code files in a consistent fashion. The file astylerc, at the top of the github repo, contains the default settings we use.

Our conventions are:

- Lines should be kept to 80 characters where reasonable.
- LF endings (unix style), not CRLF (windows style)
- spaces, not tabs
- indent to four (4) spaces (“Four shalt be the number thou shalt count, and the number of the counting shall be four. Three shalt thou not count, neither count thou five…”)
- braces shall be on their own lines, like this:

```cpp
if (p)
{
    foo();
}
```
Naming Conventions

- classes should be names using UpperCamelCase
- functions should be in lowerCamelCase
- member variables should be prefixed with “m_”, followed by the name in lowerCamelCase – for example, “m_numberOfPoints”
- there should be one class per file, and the name of the file should match the class name – that is, class PointData should live in files PointData.hpp and PointData.cpp.

Other Conventions

- Surround all code with “namespace pdal {...}”; where justifiable, you may introduce a nested namespace.
- All exceptions that are not caught internally should be of type pdal_error. Exceptions used as local error handling should always be caught.
- Don’t put member function bodies in the class declaration in the header file, unless clearly justified for performance reasons. Use the “inline” keyword in these cases(?).
  - Use const.
  - Don’t put “using” declarations in headers.
  - Document all public (and protected) member functions using doxygen markup.

#include Conventions

- For public headers from the ./include/pdal directory, use angle brackets: #include <pdal/Stage.h>
- For private headers (from somewhere in ./src), use quotes: #include “Support.hpp”
- Don’t #include a file where a simple forward declaration will do. (Note: this only applies to pdal files; don’t forward declare from system or 3rd party headers.)
- Don’t include a file unless it actually is required to compile the source unit.
- Don’t use manual include guards. All reasonable compilers support the once pragma:

```c
#pragma once
```
14.2.2 Contributors

Numerous organizations, companies, and individuals have contributed time, money, and code to build PDAL up into a highly capable software package. Without these contributions, PDAL would not progress as quickly, and its quality wouldn’t be as high. The development team is proud of the software, and it collectively represents years of experiences doing point cloud data management. We hope you’ll find it useful too.

This page is to recognize these contributors and their contributions. Thanks.

Engineering Contributors

Hobu (http://hobu.co) is the primary company behind the design, testing, development, and distribution of PDAL. Two Hobu team members primarily interact with PDAL. Howard Butler (https://github.com/hobu) founded the project, and he provides project leadership and software development. Andrew Bell (https://github.com/abellgithub) has contributed design, refactoring, and new feature development of PDAL over the past couple of years.

Michael Gerlek (http://github.com/mpgerlek) helped bootstrap PDAL by providing its first design, basic primitive objects, and first stage implementations.

Bradley Chambers (https://github.com/chambbj) from Grover Consulting Services (https://grovercsllc.com/) has contributed numerous features and capabilities to the PDAL project, including Poisson sampling (page 233) and Progressive Morphological Filters. He is also a prolific Tutorials (page 305) writer.
Funding Contributors

The US Army Corps of Engineers Remote Sensing / GIS Center of Expertise at CRREL sponsors development of PDAL for its use in point cloud data management systems. CRREL’s GRiD project manages LiDAR and point cloud data for a multitude of U.S. Army Corps missions. Find out more about GRiD in this LiDAR Magazine article.

NSF, in collaboration with Dr. Craig Glennie at the University of Houston, supports PDAL with funding support to develop and enhance statistical methods, transformation operations, tutorial and example development, and PCL integration.
14.2.3 Docs

Requirements

To build the PDAL documentation yourself, you need to install the following items:

- **Sphinx** (http://sphinx-doc.org/)
- **Breathe** (https://github.com/michaeljones/breathe)
- **Doxygen** (http://www.stack.nl/~dimitri/doxygen/)
- **LaTeX** (https://en.wikipedia.org/wiki/LaTeX)

Sphinx (http://sphinx-doc.org/) and Breathe (https://github.com/michaeljones/breathe)

Python dependencies should be installed from PyPI (https://pypi.python.org/pypi) with `pip` or `easy_install`.

```
(sudo) pip install sphinx sphinxconfig-bibtex breathe
```

Note: If you are installing these packages to a system-wide directory, you may need the `sudo` in front of the `pip`, though it might be better that instead you use virtual environments (https://pypi.python.org/pypi/virtualenv) instead of installing the packages system-wide.

Doxygen

The PDAL documentation also depends on **Doxygen** (http://www.stack.nl/~dimitri/doxygen/), which can be installed from source or from binaries from the doxygen website (http://www.stack.nl/~dimitri/doxygen/download.html). If you are on Max OS X and use `homebrew` (http://mxcl.github.io/homebrew/), you can install doxygen with a simple `brew install doxygen`. 
Latex

Latex (https://en.wikipedia.org/wiki/LaTeX) and pdflatex (https://www.tug.org/applications/pdftex/) are used to generate the companion PDF of the website.

dvipng

For math output, we depend on dvipng (https://en.wikipedia.org/wiki/Dvipng) to turn Latex (https://en.wikipedia.org/wiki/LaTeX) output into math PNGs.

Generation

Once you have installed all the doc dependencies, you can then build the documentation itself. The doc/ directory in the PDAL source tree contains a Makefile which can be used to build all documentation. For a list of the output formats supported by Sphinx, simply type make. For example, to build html documentation:

```
cd doc
make doxygen html
```

The html docs will be placed in doc/build/html/. The make doxygen is necessary to re-generate the API documentation from the source code using Breathe (https://github.com/michaeljones/breathe) and Sphinx (http://sphinx-doc.org/).

Note: For a full build of the C++ API (page 519) documentation, you need to make doxygen to have it build its XML output which is consumed by Breathe (https://github.com/michaeljones/breathe) before make html can be issued.

Website

The http://pdal.io website is regenerated from the *-maintenance branch using Travis (page 518). It will be committed by the PDAL-docs GitHub (http://github.com/PDAL/PDAL) user and pushed to the https://github.com/PDAL/pdal.github.io repository. The website is then served via GitHub Pages (https://pages.github.com/).

Note: The website is regenerated and pushed only on the after_success Travis (page 518) call. If the tests aren’t passing, the website won’t be updated.
14.2.4 Building Docker Containers for PDAL

PDAL’s repository (page 14) is linked to DockerHub (https://hub.docker.com/r/pdal/pdal/) for automatic building of Docker (https://www.docker.com/) containers. PDAL keeps three Docker containers current.

- pdal/ubuntu-dependencies:latest – PDAL’s dependencies
- pdal/pdal:latest – PDAL master
- pdal/pdal:1.5 – PDAL maintenance branch

Note: Containers are built upon the Dependencies (page 507) container, but the Dependencies (page 507) container is not pinned to specific Bionic or PDAL release times. It corresponds to wherever the dependencies tag of the PDAL source tree at https://github.com/PDAL/PDAL resides.

Dependencies

The PDAL dependencies Docker container is used by both the latest and release branch Docker containers. The dependencies container is also used during Continuous Integration (page 518) testing by Travis. It is built using the Dockerfile at https://github.com/PDAL/PDAL/blob/master/scripts/docker/ubuntu/dependencies/Dockerfile

The pdal/dependencies:latest image is regenerated by force-pushing a tag of the SHA you wish to use to have DockerHub (https://hub.docker.com/r/pdal/pdal/) build.

```
git tag -f dependencies
git push origin refs/tags/dependencies -f
```

Note: The dependencies container is currently built upon Ubuntu Bionic (http://releases.ubuntu.com/18.04/). When the next Ubuntu LTS is released, the PDAL project will likely move to it.

Maintenance

A PDAL container corresponding to the last major release is automatically created and maintained with every commit to the active release branch. For example, the 1.4-maintenance branch will have a corresponding pdal/pdal:1.4 container made with every commit on DockerHub (https://hub.docker.com/r/pdal/pdal/). Users are encouraged to use these containers for testing, bug confirmation, and deployment.
Latest (or master)

A PDAL container corresponding to a developer-selected release point is made available at `pdal/pdal:latest` and corresponds to the manual push of a `docker-master` tag by PDAL developers. This container is typically used for testing and verification of fixes, and it is recommended that users looking to depend on PDAL’s Docker containers always use known release versions off of the last stable release branch.

**Warning:** You should be using the *Maintenance* (page 507) Docker container for any production-oriented operations. Only use the latest one to test or prototype a latest, unreleased feature.

Fig. 1: Docker containers on maintenance branch correspond to major PDAL releases.

Fig. 2: The `pdal/pdal:latest` branch is current relative to the `docker-master` branch in GitHub.

```bash
$ git tag -f docker-master
$ git push origin refs/tags/docker-master -f
```
14.2.5 Alpine

This page is intended to provide information about Alpine that may be useful for PDAL developers, especially when it comes to adding new PDAL dependencies.

Packages

When adding a dependency to PDAL, you will need to update our Travis configuration for continuous integration and testing, and Dockerfiles for automated builds. Begin by checking for your package in https://pkgs.alpinelinux.org/packages. Packages containing binaries can typically be found by searching for the library/package name alone. Development files are typically grouped in a separate subpackage with -dev appended to the package name. Libraries are sometimes grouped in yet another subpackage with -libs appended. It may take a little inspection of the package contents to determine exactly what you are getting with a particular package.

If a package does not yet exist, you’ll need to consult https://wiki.alpinelinux.org/wiki/Creating_an_Alpine_package or phone a friend. Alpine developers can frequently be found on the IRC channel #alpine-devel.

Travis

We currently run our Travis CI builds by first pulling alpine:3.6 and then running a script within the Alpine container. Any new dependencies that are required for PDAL to be built and tested will need to be added to https://github.com/PDAL/PDAL/blob/master/scripts/ci/script.sh.

Docker

Our Docker automated builds are built from the Dockerfiles located in https://github.com/PDAL/PDAL/tree/master/scripts/docker. There are folders for each supported release as well as master, and there are variants for Alpine and Ubuntu based images. In the Alpine Dockerfiles, any development dependencies should be added in the apk add step that uses the --virtual switch, as these will be deleted after compilation. Any runtime dependencies should be added to the regular apk add step.
14.2.6 Testing

Unit Tests

A unit test framework is provided, with the goal that all (nontrivial) classes will have unit tests. At the very least, each new class should have a corresponding unit test file stubbed in, even if there aren’t any tests yet.

- Our unit tests also include testing of the command line Applications (page 25) and known plugins.
- We use the Google C++ Test Framework (https://code.google.com/p/googletest/), but a local copy of it is embedded in the PDAL source tree, and you don’t have to have it available as a dependency.
- Unit tests for features that are configuration-dependent, e.g. laszip compression, should be put under the same #ifdef guards as the classes being tested.
- The Support class, in the ./test/unit directory, provides some functions for comparing files, etc, that are useful in writing test cases.
- Unit tests should not be long-running.

Running the Tests

To run all unit tests, issue the following command from your build directory:

```
$ ctest
```

make test or ninja test should still work as well.

Depending on the which optional components you’ve chose to build, your output should resemble the following:

```
Test project /Users/hobu/dev/git/pdal
    Start 1: pdal_bounds_test
1/61 Test #1: pdal_bounds_test ...................... Passed 0.
  02 sec
    Start 2: pdal_config_test
2/61 Test #2: pdal_config_test ...................... Passed 0.
  02 sec
    Start 3: pdal_file_utils_test
3/61 Test #3: pdal_file_utils_test .................... Passed 0.
  02 sec
    Start 4: pdal_georeference_test
4/61 Test #4: pdal_georeference_test .................. Passed 0.
  02 sec
```
Start 5: pdal_kdindex_test
5/61 Test #5: pdal_kdindex_test .................. Passed 0.
→03 sec
Start 6: pdal_log_test
6/61 Test #6: pdal_log_test ...................... Passed 0.
→03 sec
Start 7: pdal_metadata_test
7/61 Test #7: pdal_metadata_test ............... Passed 0.
→02 sec
Start 8: pdal_options_test
8/61 Test #8: pdal_options_test .................. Passed 0.
→02 sec
Start 9: pdal_pdalutils_test
9/61 Test #9: pdal_pdalutils_test ............... Passed 0.
→02 sec
Start 10: pdal_pipeline_manager_test
10/61 Test #10: pdal_pipeline_manager_test ....... Passed 0.
→03 sec
Start 11: pdal_point_view_test
11/61 Test #11: pdal_point_view_test ............ Passed 2.
→03 sec
Start 12: pdal_point_table_test
12/61 Test #12: pdal_point_table_test ............ Passed 0.
→03 sec
Start 13: pdal_spatial_reference_test
13/61 Test #13: pdal_spatial_reference_test ...... Passed 0.
→07 sec
Start 14: pdal_support_test
14/61 Test #14: pdal_support_test ............... Passed 0.
→02 sec
Start 15: pdal_user_callback_test
15/61 Test #15: pdal_user_callback_test ............ Passed 0.
→02 sec
Start 16: pdal_utils_test
16/61 Test #16: pdal_utils_test .................. Passed 0.
→02 sec
Start 17: pdal_lazperf_test
17/61 Test #17: pdal_lazperf_test ................ Passed 0.
→04 sec
Start 18: pdal_io_bpf_test
18/61 Test #18: pdal_io_bpf_test ................ Passed 0.
→20 sec
Start 19: pdal_io_buffer_test
19/61 Test #19: pdal_io_buffer_test ............... Passed 0.
→02 sec

(continues on next page)

14.2. Project 511
Start 20: pdal_io_faux_test
20/61 Test #20: pdal_io_faux_test ................. Passed 0.
→04 sec
Start 21: pdal_io_ilvis2_test
21/61 Test #21: pdal_io_ilvis2_test ................. Passed 0.
→06 sec
Start 22: pdal_io_las_reader_test
22/61 Test #22: pdal_io_las_reader_test ............ Passed 0.
→49 sec
Start 23: pdal_io_las_writer_test
→27 sec
Start 24: pdal_io_optech_test
24/61 Test #24: pdal_io_optech_test ................. Passed 0.
→03 sec
Start 25: pdal_io_ply_reader_test
25/61 Test #25: pdal_io_ply_reader_test ............ Passed 0.
→03 sec
Start 26: pdal_io_ply_writer_test
26/61 Test #26: pdal_io_ply_writer_test ............ Passed 0.
→02 sec
Start 27: pdal_io_qfit_test
27/61 Test #27: pdal_io_qfit_test ................. Passed 0.
→03 sec
Start 28: pdal_io_sbet_reader_test
28/61 Test #28: pdal_io_sbet_reader_test ............ Passed 0.
→04 sec
Start 29: pdal_io_sbet_writer_test
29/61 Test #29: pdal_io_sbet_writer_test ............ Passed 0.
→03 sec
Start 30: pdal_io_terrasolid_test
30/61 Test #30: pdal_io_terrasolid_test ............ Passed 0.
→03 sec
Start 31: pdal_filters_chipper_test
31/61 Test #31: pdal_filters_chipper_test ............ Passed 0.
→03 sec
Start 32: pdal_filters_colorization_test
32/61 Test #32: pdal_filters_colorization_test ..... Passed 11.
→40 sec
Start 33: pdal_filters_crop_test
33/61 Test #33: pdal_filters_crop_test ............ Passed 0.
→04 sec
Start 34: pdal_filters_decimation_test
34/61 Test #34: pdal_filters_decimation_test ....... Passed 0.
→02 sec
Start 35: pdal_filters_divider_test
35/61 Test #35: pdal_filters_divider_test .......... Passed 0.
03 sec
Start 36: pdal_filters_ferry_test
36/61 Test #36: pdal_filters_ferry_test .......... Passed 0.
04 sec
Start 37: pdal_filters_merge_test
37/61 Test #37: pdal_filters_merge_test .......... Passed 0.
03 sec
Start 38: pdal_filters_reprojection_test
38/61 Test #38: pdal_filters_reprojection_test ...... Passed 0.
03 sec
Start 39: pdal_filters_range_test
39/61 Test #39: pdal_filters_range_test .......... Passed 0.
05 sec
Start 40: pdal_filters_randomize_test
40/61 Test #40: pdal_filters_randomize_test ........ Passed 0.
02 sec
Start 41: pdal_filters_sort_test
41/61 Test #41: pdal_filters_sort_test .......... Passed 0.
39 sec
Start 42: pdal_filters_splitter_test
42/61 Test #42: pdal_filters_splitter_test .......... Passed 0.
03 sec
Start 43: pdal_filters_stats_test
43/61 Test #43: pdal_filters_stats_test .......... Passed 0.
03 sec
Start 44: pdal_filters_transformation_test
44/61 Test #44: pdal_filters_transformation_test ... Passed 0.
03 sec
Start 45: pdal_merge_test
45/61 Test #45: pdal_merge_test .................... Passed 0.
07 sec
Start 46: pc2pc_test
46/61 Test #46: pc2pc_test .......................... Passed 0.
15 sec
Start 47: xml_schema_test
47/61 Test #47: xml_schema_test .................... Passed 0.
02 sec
Start 48: pdal_filters_attribute_test
48/61 Test #48: pdal_filters_attribute_test ........ Passed 0.
09 sec
Start 49: pdal_plugins_cpd_kernel_test
49/61 Test #49: pdal_plugins_cpd_kernel_test ........***Exception:
(continued from previous page)

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Test Case</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>50/61 Test</td>
<td>#50: hexbintest</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>03 sec</td>
<td></td>
</tr>
<tr>
<td>51/61 Test</td>
<td>#51: icetest</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>04 sec</td>
<td></td>
</tr>
<tr>
<td>52/61 Test</td>
<td>#52: mrsidtest</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>06 sec</td>
<td></td>
</tr>
<tr>
<td>53/61 Test</td>
<td>#53: pdal_io_nitf_writer_test</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>08 sec</td>
<td></td>
</tr>
<tr>
<td>54/61 Test</td>
<td>#54: pdal_io_nitf_reader_test</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>04 sec</td>
<td></td>
</tr>
<tr>
<td>55/61 Test</td>
<td>#55: ocitest</td>
<td>Failed 0</td>
</tr>
<tr>
<td></td>
<td>06 sec</td>
<td></td>
</tr>
<tr>
<td>56/61 Test</td>
<td>#56: pcltest</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>28 sec</td>
<td></td>
</tr>
<tr>
<td>57/61 Test</td>
<td>#57: pgpointcloudtest</td>
<td>Passed 1</td>
</tr>
<tr>
<td></td>
<td>66 sec</td>
<td></td>
</tr>
<tr>
<td>58/61 Test</td>
<td>#58: plangtest</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>14 sec</td>
<td></td>
</tr>
<tr>
<td>59/61 Test</td>
<td>#59: python_predicate_test</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>16 sec</td>
<td></td>
</tr>
<tr>
<td>60/61 Test</td>
<td>#60: python_programmable_test</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>15 sec</td>
<td></td>
</tr>
<tr>
<td>61/61 Test</td>
<td>#61: sqlitetest</td>
<td>Passed 0</td>
</tr>
<tr>
<td></td>
<td>55 sec</td>
<td></td>
</tr>
</tbody>
</table>

97% tests passed, 2 tests failed out of 61

Total Test time (real) = 21.57 sec

The following tests FAILED:
- 49 - pdal_plugins_cpd_kernel_test (OTHER_FAULT)
- 55 - ocitest (Failed)

For a more verbose output, use the -V flag. Or, to run an individual test suite, use -R <suite
name>. For example:

```bash
$ ctest -V -R pdal_io_bpf_test
```

Should produce output similar to:

```bash
UpdateCTestConfiguration from :/Users/hobu/dev/git/pdal/
  → DartConfiguration.tcl
UpdateCTestConfiguration from :/Users/hobu/dev/git/pdal/
  → DartConfiguration.tcl
Test project /Users/hobu/dev/git/pdal
Constructing a list of tests
Done constructing a list of tests
Checking test dependency graph...
Checking test dependency graph end
test 18
  Start 18: pdal_io_bpf_test
18: Test command: /Users/hobu/dev/git/pdal/bin/pdal_io_bpf_test
18: Environment variables:
18:    PDAL_DRIVER_PATH=/Users/hobu/dev/git/pdal/lib
18:    Test timeout computed to be: 9.99988e+06
18: [==========] Running 20 tests from 1 test case.
18: [----------] Global test environment set-up.
18: [----------] 20 tests from BPFTest
18: [ RUN ] BPFTest.test_point_major
18: [ OK ] BPFTest.test_point_major (8 ms)
18: [ RUN ] BPFTest.test_dim_major
18: [ OK ] BPFTest.test_dim_major (3 ms)
18: [ RUN ] BPFTest.test_byte_major
18: [ OK ] BPFTest.test_byte_major (4 ms)
18: [ RUN ] BPFTest.test_point_major_zlib
18: [ OK ] BPFTest.test_point_major_zlib (6 ms)
18: [ RUN ] BPFTest.test_dim_major_zlib
18: [ OK ] BPFTest.test_dim_major_zlib (4 ms)
18: [ RUN ] BPFTest.test_byte_major_zlib
18: [ OK ] BPFTest.test_byte_major_zlib (5 ms)
18: [ RUN ] BPFTest.roundtrip_byte
18: [ OK ] BPFTest.roundtrip_byte (15 ms)
18: [ RUN ] BPFTest.roundtrip_dimension
18: [ OK ] BPFTest.roundtrip_dimension (10 ms)
18: [ RUN ] BPFTest.roundtrip_point
18: [ OK ] BPFTest.roundtrip_point (11 ms)
18: [ RUN ] BPFTest.roundtrip_byte_compression
18: [ OK ] BPFTest.roundtrip_byte_compression (16 ms)
18: [ RUN ] BPFTest.roundtrip_dimension_compression
18: [ OK ] BPFTest.roundtrip_dimension_compression (13 ms)
(continues on next page)
18: [ RUN ] BPFTest.roundtrip_point_compression
18: [ OK  ] BPFTest.roundtrip_point_compression (14 ms)
18: [ RUN ] BPFTest.roundtrip_scaling
18: [ OK  ] BPFTest.roundtrip_scaling (10 ms)
18: [ RUN ] BPFTest.extra_bytes
18: [ OK  ] BPFTest.extra_bytes (15 ms)
18: [ RUN ] BPFTest.bundled
18: [ OK  ] BPFTest.bundled (17 ms)
18: [ RUN ] BPFTest.inspect
18: [ OK  ] BPFTest.inspect (1 ms)
18: [ RUN ] BPFTest.mueller
18: [ OK  ] BPFTest.mueller (0 ms)
18: [ RUN ] BPFTest.flex
18: [ OK  ] BPFTest.flex (9 ms)
18: [ RUN ] BPFTest.flex2
18: [ OK  ] BPFTest.flex2 (7 ms)
18: [ RUN ] BPFTest.outputdims
18: [ OK  ] BPFTest.outputdims (14 ms)
18: [----------] 20 tests from BPFTest (182 ms total)
18: [----------] Global test environment tear-down
18: [==========] 20 tests from 1 test case ran. (182 ms total)
1/1 Test #18: pdal_io_bpf_test ................. Passed 0.200 sec

The following tests passed:
pdal_io_bpf_test

100% tests passed, 0 tests failed out of 1

$ bin/pdal_io_test

Again, the output should resemble the following:

[==========] Running 20 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 20 tests from BPFTest
[ RUN ] BPFTest.test_point_major
[ OK  ] BPFTest.test_point_major (7 ms)
[ RUN ] BPFTest.test_dim_major
[ OK  ] BPFTest.test_dim_major (3 ms)
[ RUN ] BPFTest.test_byte_major
[ OK  ] BPFTest.test_byte_major (4 ms)
This invocation allows us to alter Google Test’s default behavior. For more on the available flags type:

```
$ bin/<test_name> --help
```

Key among these flags are the ability to list tests (`--gtest_list_tests`) and to run only
select tests (--gtest_filter).

**Note:** If the PostgreSQL PointCloud plugin was enabled on the CMake command line (with -DBUILD_PLUGIN_PGPPOINTCLOUD=ON) then ctest will attempt to run the pgpointcloud tests. And you will get PostgreSQL connection errors if the libpq environment variables ([https://www.postgresql.org/docs/current/static/libpq-envars.html](https://www.postgresql.org/docs/current/static/libpq-envars.html)) are not correctly set in your shell. This is for example how you can run the pgpointcloud tests:

```bash
$ PGUSER=pdal PGPASSWORD=pdal PGHOST=localhost ctest -R pgpointcloudtest
```

**Test Data**

Use the directory ./test/data to store files used for unit tests. A vfunction is provided in the Support class for referencing that directory in a configuration-independent manner.

Temporary output files from unit tests should go into the ./test/temp directory. A Support function is provided for referencing this directory as well.

Unit tests should always clean up and remove any files that they create (except perhaps in case of a failed test, in which case leaving the output around might be helpful for debugging).

### 14.2.7 Continuous Integration

PDAL *regression tests* (page 510) are run on a per-commit basis by at least two continuous integration platforms.

**Status**


[building](https://ci.appveyor.com/project/hobu/pdal)

**Travis**

The Travis continuous integration platform runs the PDAL test suite on Alpine Linux. The build status and other supporting information can be found at [https://travis-ci.org/PDAL/PDAL](https://travis-ci.org/PDAL/PDAL). Its configuration can be found at [https://github.com/PDAL/PDAL/blob/master/.travis.yml](https://github.com/PDAL/PDAL/blob/master/.travis.yml). All administrators of the GitHub PDAL group have rights to modify the Travis configuration.

It uses the `alpine:edge` Docker image found at [https://hub.docker.com/_/alpine/](https://hub.docker.com/_/alpine/) as a base platform. If you want to add new functionality based on a dependency, you will need to ensure
that the dependency is available in https://pkgs.alpinelinux.org/packages and update the Travis configuration YAML accordingly.

**AppVeyor**

PDAL uses the AppVeyor continuous integration platform to run the PDAL compilation and test suite on Windows. The build status and other supporting information can be found at https://ci.appveyor.com/project/hobu/pdal Its configuration can be found at https://github.com/PDAL/PDAL/blob/master/appveyor.yml All administrators of the GitHub PDAL group have rights to modify the AppVeyor configuration.

Howard Butler (http://github.com/hobu) currently pays the bill to run in the AppVeyor upper performance processing tier. The AppVeyor configuration depends on Conda (page 16) for dependencies. If you want to add new test functionality based on a dependency, you will need to update Conda (page 16) with a new package to do so.

### 14.3 API

PDAL is a C++ library, and its primary API is in that language. There is also a Python (page 295) API that allows reading of data and interaction with Numpy (http://www.numpy.org/).

**Note:** Users looking for documentation on how to use PDAL's command line applications should look here (page 25) and users looking for documentation on how to contribute to PDAL should look here (page 445).

#### 14.3.1 C++ API

```cpp
class pdal::BOX2D

BOX2D (page 519) represents a two-dimensional box with double-precision bounds.

Subclassed by pdal::BOX3D (page 523)
```
Public Functions

```cpp
inline BOX2D ()
Construct an “empty” bounds box.

inline BOX2D (double minx, double miny, double maxx, double maxy)
Construct and initialize a bounds box.

Parameters
• minx – Minimum X value.
• miny – Minimum Y value.
• maxx – Maximum X value.
• maxy – Maximum Y value.

bool empty () const
Determine whether a bounds box has not had any bounds set.

Returns Whether the bounds box is empty.

bool valid () const
Determine whether a bounds box has had any bounds set.

Returns Whether the bounds box is valid.

void clear ()
Clear the bounds box to an empty state.

BOX2D (page 519) & grow (double x, double y)
Expand the bounds of the box to include the specified point.

Parameters
• x – X point location.
• y – Y point location.

BOX2D (page 519) & grow (double dist)
Expand the bounds of the box in all directions by a specified amount.

Parameters dist – Distance by which to expand the box.

inline bool contains (double x, double y) const
Determine if a bounds box contains a point.

Parameters
• x – X dimension value.
• y – Y dimension value.
```
Returns Whether both dimensions are equal to or less than the maximum box values and equal to or more than the minimum box values.

```
inline bool equal (const BOX2D (page 519) &other) const
Determine if the bounds of this box are the same as that of another box.

Empty bounds boxes are always equal.
```

**Parameters** other – Bounds box to check for equality.

**Returns** true if the provided box has equal limits to this box, false otherwise.

```
inline bool operator==(BOX2D (page 519) const &other) const
Determine if the bounds of this box are the same as that of another box.

Empty bounds boxes are always equal.
```

**Parameters** other – Bounds box to check for equality.

**Returns** true if the provided box has equal limits to this box, false otherwise.

```
inline bool operator!=(BOX2D (page 519) const &other) const
Determine if the bounds of this box are different from that of another box.

Empty bounds boxes are never unequal.
```

**Parameters** other – Bounds box to check for inequality.

**Returns** true if the provided box has limits different from this box, false otherwise.

```
inline BOX2D (page 519) &grow (const BOX2D (page 519) &other)
Expand this box to contain another box.
```

**Parameters** other – Box that this box should contain.

```
inline void clip (const BOX2D (page 519) &other)
Clip this bounds box by another so it will be contained by the other box.
```

**Parameters** other – Clipping box for this box.

```
inline bool contains (const BOX2D (page 519) &other) const
Determine if another bounds box is contained in this bounds box.

Equal limits are considered to be contained.
```

**Parameters** other – Bounds box to check for containment.

**Returns** true if the provided box is contained in this box, false otherwise.

```
inline bool overlaps (const BOX2D (page 519) &other) const
Determine if another box overlaps this box.
```
Parameters **other** – Box to test for overlap.

**Returns** Whether the provided box overlaps this box.

```cpp
inline std::string toBox (uint32_t precision = 8) const
```
Convert this box to a string suitable for use in SQLite.

**Parameters** **precision** – Precision for output [default: 8]

**Returns** String format of this box.

```cpp
inline std::string toWKT (uint32_t precision = 8) const
```
Convert this box to a well-known text string.

**Parameters** **precision** – Precision for output [default: 8]

**Returns** String format of this box.

```cpp
inline std::string toGeoJSON (uint32_t precision = 8) const
```
Convert this box to a GeoJSON text string.

**Parameters** **precision** – Precision for output [default: 8]

**Returns** String format of this box.

```cpp
void parse (const std::string &s, std::string::size_type &pos)
```
Parse a string as a *BOX2D* (page 519).

**Parameters**

- **s** – String representation of the box.
- **pos** – Position in the string at which to start parsing. On return set to parsing end position.

### Public Members

double **minx**
Minimum X value.

double **maxx**
Maximum X value.

double **miny**
Minimum Y value.

double **maxy**
Maximum Y value.
Public Static Functions

```cpp
static const BOX2D (page 519) &getDefaultSpatialExtent ()
// Return a statically-allocated Bounds extent that represents infinity.

Returns: A bounds box with infinite bounds,
```

struct error : public std::runtime_error
```

Public Functions

```cpp
inline error (const std::string &err)
```

```cpp
class pdal::BOX3D : private pdal::BOX2D (page 519)
// BOX3D (page 523) represents a three-dimensional box with double-precision bounds.
```

Public Functions

```cpp
inline BOX3D ()
    // Clear the bounds box to an empty state.
```

```cpp
inline BOX3D (const BOX3D (page 523) &box)
```

```cpp
BOX3D (page 523) &operator= (const BOX3D (page 523) &box) = default
```

```cpp
inline explicit BOX3D (const BOX2D (page 519) &box)
```

```cpp
inline BOX3D (double minx, double miny, double minz, double maxx, double maxy, double maxz)
    // Construct and initialize a bounds box.
```

Parameters

- minx – Minimum X value.
- miny – Minimum Y value.
- minz – Minimum Z value.
- maxx – Maximum X value.
- maxy – Maximum Y value.
- maxz – Maximum Z value.

```cpp
bool empty () const
    // Determine whether a bounds box has not had any bounds set (is in a state as if
    // default-constructed).

Returns: Whether the bounds box is empty.
bool valid() const
    Determine whether a bounds box has had any bounds set.

    **Returns** \true if the bounds box is not empty

*BOX3D* (page 523) &grow (double x, double y, double z)
    Expand the bounds of the box if a value is less than the current minimum or greater
    than the current maximum.

    If the bounds box is currently empty, both minimum and maximum box bounds
    will be set to the provided value.

    **Parameters**
    - **x** – X dimension value.
    - **y** – Y dimension value.
    - **z** – Z dimension value.

void clear()
    Clear the bounds box to an empty state.

*inline* bool contains (double x, double y, double z) const
    Determine if a bounds box contains a point.

    **Parameters**
    - **x** – X dimension value.
    - **y** – Y dimension value.
    - **z** – Z dimension value.

    **Returns** Whether both dimensions are equal to or less than the maximum
    box values and equal to or more than the minimum box values.

*inline* bool contains (const *BOX3D* (page 523) &other) const
    Determine if another bounds box is contained in this bounds box.

    Equal limits are considered to be contained.

    **Parameters** other – Bounds box to check for containment.

    **Returns** \true if the provided box is contained in this box, \false otherwise.

*inline* bool equal (const *BOX3D* (page 523) &other) const
    Determine if the bounds of this box are the same as that of another box.

    Empty bounds boxes are always equal.

    **Parameters** other – Bounds box to check for equality.

    **Returns** \true if the provided box has equal limits to this box, \false otherwise.
inline bool operator== (BOX3D (page 523) const &rhs) const
Determine if the bounds of this box are the same as that of another box.

Empty bounds boxes are always equal.

Parameters other – Bounds box to check for equality.

Returns true if the provided box has equal limits to this box, false otherwise.

inline bool operator!= (BOX3D (page 523) const &rhs) const
Determine if the bounds of this box are different from that of another box.

Empty bounds boxes are never unequal.

Parameters other – Bounds box to check for inequality.

Returns true if the provided box has limits different from this box, false otherwise.

inline BOX3D (page 523) &grow (const BOX3D (page 523) &other)
Expand this box to contain another box.

Parameters other – Box that this box should contain.

inline BOX3D (page 523) &grow (double dist)
Expand this box by a specified amount.

Parameters dist – Distance by which box should be expanded.

inline void clip (const BOX3D (page 523) &other)
Clip this bounds box by another so it will be contained by the other box.

Parameters other – Clipping box for this box.

inline bool overlaps (const BOX3D (page 523) &other) const
Determine if another box overlaps this box.

Parameters other – Box to test for overlap.

Returns Whether the provided box overlaps this box.

inline BOX2D (page 519) to2d () const
Convert this box to 2-dimensional bounding box.

Returns Bounding box with Z dimension stripped.

inline std::string toBox (uint32_t precision = 8) const
Convert this box to a string suitable for use in SQLite.

Parameters precision – Precision for output [default: 8]

Returns String format of this box.

inline std::string toWKT (uint32_t precision = 8) const
Convert this box to a well-known text string.
**Parameters** precision – Precision for output [default: 8]

**Returns** String format of this box.

```c
void parse (const std::string &s, std::string::size_type &pos)
```

Parse a string as a *BOX3D* (page 523).

**Parameters**

- **s** – String representation of the box.
- **pos** – Position in the string at which to start parsing. On return set to parsing end position.

**Public Members**

double minz
  Minimum Z value.

double maxz
  Maximum Z value.

double minx
  Minimum X value.

double maxx
  Maximum X value.

double miny
  Minimum Y value.

double maxy
  Maximum Y value.

**Public Static Functions**

```c
static const BOX3D (page 523) &getDefaultSpatialExtent ()
```

Return a statically-allocated Bounds extent that represents infinity.

**Returns** A bounds box with infinite bounds,

```c
struct error : public std::runtime_error
```
**Public Functions**

```cpp
inline error (const std::string &err)
```

**pdal::Charbuf**

```cpp
class pdal::Charbuf : public std::streambuf
Allow a data buffer to be used at a std::streambuf.
```

**Public Functions**

```cpp
inline PDAL_DLL Charbuf ()
Construct an empty Charbuf (page 527).
```

```cpp
inline PDAL_DLL Charbuf (std::vector<char> &v, pos_type bufOffset = 0)
Construct a Charbuf (page 527) that wraps a byte vector.
```

**Parameters**

- `v` – Byte vector to back streambuf.
- `bufOffset` – Offset in vector (ignore bytes before offset).

```cpp
inline PDAL_DLL Charbuf (char *buf, size_t count, pos_type bufOffset = 0)
Construct a Charbuf (page 527) that wraps a byte buffer.
```

**Parameters**

- `buf` – Buffer to back streambuf.
- `count` – Size of buffer.
- `bufOffset` – Offset in vector (ignore bytes before offset).

```cpp
PDAL_DLL void initialize (char *buf, size_t count, pos_type bufOffset = 0)
Set a buffer to back a Charbuf (page 527).
```

**Parameters**

- `buf` – Buffer to back streambuf.
- `count` – Size of buffer.
- `bufOffset` – Offset in vector (ignore bytes before offset).
namespace pdal::Dimension

**Typedefs**

```cpp
typedef std::vector<Detail (page 530)> DetailList
```

** Enums **

```cpp
enum BaseType
    Values:
    enumerator None
    enumerator Signed
    enumerator Unsigned
    enumerator Floating

denum Type
    Values:
    enumerator None
    enumerator Unsigned8
    enumerator Signed8
    enumerator Unsigned16
    enumerator Signed16
    enumerator Unsigned32
    enumerator Signed32
    enumerator Unsigned64
    enumerator Signed64
    enumerator Float
    enumerator Double
```
Functions

**inline** `BaseType (page 528) fromName (std::string name)`

**inline** `std::string toName (BaseType (page 528) b)`

**inline** `std::size_t size (Type (page 528) t)`

**inline** `BaseType (page 528) base (Type (page 528) t)`

**inline** `std::string interpretationName (Type (page 528) dimtype)`

Get a string representation of a datatype.

**Parameters** `[in] dimtype – Dimension (page 528) type.**

**Returns** String representation of dimension type.

**inline** `Type (page 528) type (std::string s)`

Get the type corresponding to a type name.

**Parameters** `s` – Name of type.

**Returns** Corresponding type enumeration value.

**inline** `Type (page 528) type (const std::string &baseType, size_t size)`

**inline** `std::size_t extractName (const std::string &s, std::string::size_type p)`

Extract a dimension name of a string.

*Dimension (page 528) names start with an alpha and continue with numbers or underscores.*

**Parameters**

- `s` – String from which to extract dimension name.
- `p` – Position at which to start extracting.

**Returns** Number of characters in the extracted name.

**inline** `std::string fixName (std::string name)`

**inline** `bool nameValid (std::string name)`

**inline** `std::istream &operator>>(std::istream &in, Dimension (page 528)::Type (page 528) &type)`

**inline** `std::ostream &operator<<(std::ostream &out, const Dimension (page 528)::Type (page 528) &type)`
**Variables**

```cpp
static const int COUNT = (std::numeric_limits<uint16_t>::max)()
static const int PROPRIETARY = 0xF000
```

```cpp
class Detail
    #include <DimDetail.hpp>
```

```cpp
pdal::Extractor
```

```cpp
class pdal::Extractor
Buffer wrapper for input of binary data from a buffer.
Subclassed by pdal::BeExtractor, pdal::LeExtractor, pdal::SwitchableExtractor
```

**Public Functions**

```cpp
inline Extractor(const char *buf, std::size_t size)
Construct an extractor to operate on a buffer.
```

- **Parameters**
  - `buf` – Buffer to extract from.
  - `size` – Buffer size.

```cpp
inline operator bool()
Determine if the buffer is good.
```

- **Returns** Whether the buffer is good.

```cpp
inline void seek(std::size_t pos)
Seek to a position in the buffer.
```

- **Parameters** `pos` – Position to seek in buffer.

```cpp
inline void skip(std::size_t cnt)
Advance buffer position.
```

- **Parameters** `cnt` – Number of bytes to skip in buffer.

```cpp
inline size_t position() const
Return the get position of buffer.
```

- **Returns** Get position.

```cpp
inline bool good() const
Determine whether the extractor is good (the get pointer is in the buffer).
```

- **Returns** Whether the get pointer is valid.
inline void get (std::string &s, size_t size)
   Extract a string of a particular size from the buffer.
   Trim trailing null bytes.

Parameters

   • s – String to extract to.
   • size – Number of bytes to extract from buffer into string.

inline void get (std::vector<char> &buf)
   Extract data to char vector.
   Vector must be sized to indicate number of bytes to extract.

Parameters buf – Vector to which bytes should be extracted.

inline void get (std::vector<unsigned char> &buf)
   Extract data to unsigned char vector.
   Vector must be sized to indicate number of bytes to extract.

Parameters buf – Vector to which bytes should be extracted.

inline void get (char *buf, size_t size)
   Extract data into a provided buffer.

Parameters

   • buf – Pointer to buffer to which bytes should be extracted.
   • size – Number of bytes to extract.

inline void get (unsigned char *buf, size_t size)
   Extract data into a provided unsigned buffer.

Parameters

   • buf – Pointer to buffer to which bytes should be extracted.
   • size – Number of bytes to extract.

virtual Extractor (page 530) &operator>>(uint8_t &v) = 0
virtual Extractor (page 530) &operator>>(int8_t &v) = 0
virtual Extractor (page 530) &operator>>(uint16_t &v) = 0
virtual Extractor (page 530) &operator>>(int16_t &v) = 0
virtual Extractor (page 530) &operator>>(uint32_t &v) = 0
virtual Extractor (page 530) &operator>>(int32_t &v) = 0
virtual Extractor (page 530) &operator>>(uint64_t &v) = 0
virtual Extractor (page 530) &operator>>(int64_t &v) = 0
**Extractor** (page 530) & operator>>(float & v) = 0

**Extractor** (page 530) & operator>>(double & v) = 0

```cpp
pdal::FileUtils
namespace pdal::FileUtils

**Functions**

std::istream *openFile (std::string const & filename, bool asBinary = true)
Open an existing file for reading.

Parameters
- **filename** – Filename.
- **asBinary** – Read as binary file (don’t convert /r/n to /n)

Returns Pointer to opened stream.

std::ostream *createFile (std::string const & filename, bool asBinary = true)
Create a file and open for writing.

Parameters
- **filename** – Filename.
- **asBinary** – Write as binary file (don’t convert /n to /r/n)

Returns Pointer to opened stream.

bool directoryExists (const std::string & dirname)
Determine if a directory exists.

Parameters **dirname** – Name of directory.

Returns Whether a directory exists.

bool createDirectory (const std::string & dirname)
Create a directory.

Parameters **dirname** – Directory name.

Returns Whether the directory was created.

bool createDirectories (const std::string & path)
Create all directories in the provided path.

Parameters **dirname** – Path name.

Returns \false on failure
void **deleteDirectory** (**const** std::string **&dirname**)  
Delete a directory and its contents.

**Parameters** **dirname** – Directory name.

std::vector<std::string> **directoryList** (**const** std::string **&dirname**)  
List the contents of a directory.

**Parameters** **dirname** – Name of directory to list.

**Returns** List of entries in the directory.

void **closeFile** (std::ostream **&ofs**)  
Close a file created with createFile.

**Parameters** **ofs** – Pointer to stream to close.

void **closeFile** (std::istream **&ifs**)  
Close a file created with openFile.

**Parameters** **ifs** – Pointer to stream to close.

bool **deleteFile** (**const** std::string **&filename**)  
Delete a file.

**Parameters** **filename** – Name of file to delete.

**Returns** true if successful, false otherwise

void **renameFile** (**const** std::string **&dest**, **const** std::string **&src**)  
Rename a file.

**Parameters**

- **dest** – Desired filename.
- **src** – Source filename.

bool **fileExists** (**const** std::string **&filename**)  
Determine if a file exists.

**Parameters** **Filename**. –

**Returns** Whether the file exists.

uintmax_t **fileSize** (**const** std::string **&filename**)  
Get the size of a file.

**Parameters** **filename** – Filename.

**Returns** Size of file.

std::string **readFileIntoString** (**const** std::string **&filename**)  
Read a file into a string.

**Parameters** **filename** – Filename.
Returns File contents as a string

std::string getcwd()
Get the current working directory with trailing separator.

Returns The current working directory.

std::string toCanonicalPath(std::string filename)
Return the path with all “.”, “..” and symbolic links removed.
The file must exist.

Parameters filename – Name of file to convert to canonical path.

Returns Canonical version of provided filename, or empty string.

std::string toAbsolutePath(const std::string &filename)
If the filename is an absolute path, just return it otherwise, make it absolute
(relative to current working dir) and return it.

Parameters filename – Name of file to convert to absolute path.

Returns Absolute version of provided filename.

std::string toAbsolutePath(const std::string &filename, const std::string base)
If the filename is an absolute path, just return it otherwise, make it absolute
(relative to base dir) and return that.

Parameters

• filename – Name of file to convert to absolute path.

• base – Base name to use.

Returns Absolute version of provided filename relative to base.

std::string getFilename(const std::string &path)
Return the file component of the given path, e.g.

“d:/foo/bar/a.c” -> “a.c”

Parameters path – Path from which to extract file component.

Returns File part of path.

std::string getDirectory(const std::string &path)
Return the directory component of the given path, e.g.

“d:/foo/bar/a.c” -> “d:/foo/bar/”

Parameters path – Path from which to extract directory component.

Returns Directory part of path.
std::string stem(const std::string &path)
    Return the filename stripped of the extension.
    . and .. are returned unchanged.

Parameters path – File path from which to extract file stem.

Returns Stem of filename.

bool isDirectory(const std::string &path)
    Determine if path is a directory.

Parameters path – Directory to check.

Returns Whether the path represents a directory.

bool isAbsolutePath(const std::string &path)
    Determine if the path is an absolute path.

Parameters path – Path to test.

Returns Whether the path is absolute.

void fileTimes(const std::string &filename, struct tm *createTime,
               struct tm *modTime)
    Get the file creation and modification times.

Parameters

• filename – Filename.
• createTime – Pointer to creation time structure.
• modTime – Pointer to modification time structure.

std::string extension(const std::string &path)
    Return the extension of the filename, including the separator (.).

Parameters path – File path from which to extract extension.

Returns Extension of filename.

std::vector<std::string> glob (std::string filespec)
    Expand a filespec to a list of files.

Parameters filespec – File specification to expand.

Returns List of files that correspond to provided file specification.
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

- **readOnly** – Must be true at this time.
- **pos** – Starting position of file to map.
- **size** – Number of bytes in file to map.

**Returns** *MapContext* (page 536). `addr()` gets the mapped address. `what()` gets any error message. `addr()` returns `nullptr` on error.

*MapContext* (page 536) **unmapFile** (*MapContext* (page 536) `ctx`)

Unmap a previously mapped file.

**Parameters** `ctx` – Previously returned *MapContext* (page 536)

**Returns** *MapContext* (page 536) indicating current state of the file mapping.

```cpp
struct MapContext
    #include <FileUtils.hpp>
```

pdal::Filter

**class** pdal::Filter : **public virtual** pdal::Stage (page 553)

Subclassed by SplitFilter, pdal::ApproximateCoplanarFilter, pdal::AssignFilter, pdal::CSFilter, pdal::ChipperFilter, pdal::ClusterFilter, pdal::ColorinterpFilter, pdal::ColorizationFilter, pdal::CovarianceFeaturesFilter, pdal::CpdFilter, pdal::CropFilter, pdal::DBSCANFilter, pdal::DEMFilter, pdal::DecimationFilter, pdal::DelaunayFilter, pdal::DividerFilter, pdal::ELMFilter, pdal::EigenvaluesFilter, pdal::EstimateRankFilter, pdal::FaceRasterFilter, pdal::FarthestPointSamplingFilter, pdal::FerryFilter, pdal::GreedyProjection, pdal::GroupByFilter, pdal::HagDelaunayFilter, pdal::HagDemFilter, pdal::HagNnFilter, pdal::HeadFilter, pdal::HexBin, pdal::IQRFilter, pdal::InfoFilter, pdal::IterativeClosestPoint, pdal::LOFFilter, pdal::LiTreeFilter, pdal::LloydKMeansFilter, pdal::LocateFilter, pdal::MADFilter, pdal::MatlabFilter, pdal::MergeFilter, pdal::MiniballFilter, pdal::MongoExpressionFilter, pdal::MortonOrderFilter, pdal::NNDistanceFilter, pdal::NeighborClassifierFilter, pdal::NormalFilter, pdal::OptimalNeighborhood, pdal::OutlierFilter, pdal::OverlayFilter, pdal::PMFFilter, pdal::PlaneFitFilter, pdal::PoissonFilter, pdal::ProjPipelineFilter, pdal::RadialDensityFilter, pdal::RandomizeFilter, pdal::RangeFilter, pdal::ReciprocityFilter, pdal::ReprojectionFilter, pdal::ReturnsFilter, pdal::SMRFilter, pdal::SampleFilter, pdal::SeparateScanLineFilter, pdal::ShellFilter, pdal::SkewnessBalancingFilter, pdal::SortFilter, pdal::SplitterFilter, pdal::StatsFilter, pdal::StreamCallbackFilter, pdal::TailFilter, pdal::TransformationFilter, pdal::VoxelCenterNearestNeighborFilter, pdal::VoxelCentroidNearestNeighborFilter, pdal::VoxelDownsizeFilter
Public Types

defined enum WhereMergeMode
Values:
    enumerator True
    enumerator False
    enumerator Auto

Public Functions

Filter()
~Filter()

Filter (page 536) &operator= (const Filter (page 536)&) = delete
Filter (const Filter (page 537)&) = delete

void splitView(const PointViewPtr &views, PointViewPtr &keeps, PointViewPtr &skips)

WhereMergeMode (page 537) mergeMode (const) const

bool eval (PointRef &p) const

struct Args

Public Members

expr::ConditionalExpression m_where

Arg *m_whereArg

Filter (page 536)::WhereMergeMode (page 537) m_whereMerge

Arg *m_whereMergeArg

pdal::IStream

class pdal::IStream
  Stream wrapper for input of binary data.
  Subclassed by pdal::IBeStream, pdal::ILeStream, pdal::ISwitchableStream
Public Functions

\begin{verbatim}
inline PDAL_DLL IStream() 
    Default constructor.

inline PDAL_DLL IStream(const std::string &filename)
    Construct an IStream (page 537) from a filename.

    Parameters filename -- File from which to read.

inline PDAL_DLL IStream(std::istream *stream)
    Construct an IStream (page 537) from an input stream pointer.

    Parameters stream -- Stream from which to read.

inline PDAL_DLL ~IStream()

inline PDAL_DLL int open(const std::string &filename)
    Open a file to extract.

    Parameters filename -- Filename.

    Returns -1 if a stream is already assigned, 0 otherwise.

inline PDAL_DLL void close()
    Close the underlying stream.

inline PDAL_DLL operator bool()
    Return the state of the stream.

    Returns The state of the underlying stream.

inline PDAL_DLL void seek(std::streampos pos)
    Seek to a position in the underlying stream.

    Parameters pos -- Position to seek to,

inline PDAL_DLL void seek(std::streampos off, std::ios_base::seekdir way)
    Seek to an offset from a specified position.

    Parameters

    • off -- Offset.

    • way -- Absolute position for offset (beg, end or cur)

inline PDAL_DLL void skip(std::streamoff offset)
    Skip relative to the current position.

    Parameters offset -- Offset from the current position.

inline PDAL_DLL std::streampos position() const
    Determine the position of the get pointer.

    Returns Current get position.
\end{verbatim}
inline PDAL_DLL bool good () const
    Determine if the underlying stream is good.

    Returns  Whether the underlying stream is good.

inline PDAL_DLL std::istream * stream ()
    Fetch a pointer to the underlying stream.

    Returns  Pointer to the underlying stream.

inline PDAL_DLL void pushStream (std::istream * strm)
    Temporarily push a stream to read from.

    Parameters strm – New stream to read from.

inline PDAL_DLL std::istream * popStream ()
    Pop the current stream and return it.

    The last stream on the stack cannot be popped.

    Returns  Pointer to the popped stream.

inline PDAL_DLL void get (std::string &s, size_t size)
    Fetch data from the stream into a string.

    NOTE - Stops when a null byte is encountered. Use a buffer/vector reader to read data with embedded nulls.

    Parameters
      • s – String to fill.
      • size – Maximum number of bytes to extract.

inline PDAL_DLL void get (std::vector< char > & buf)
    Fetch data from the stream into a vector of char.

    Parameters buf – Buffer to fill.

inline PDAL_DLL void get (std::vector< unsigned char > & buf)
    Fetch data from the stream into a vector of unsigned char.

    Parameters buf – Buffer to fill.

inline PDAL_DLL void get (char * buf, size_t size)
    Fetch data from the stream into the specified buffer of char.

    Parameters
      • buf – Buffer to fill.
      • size – Number of bytes to extract.

inline PDAL_DLL void get (unsigned char * buf, size_t size)
    Fetch data from the stream into the specified buffer of unsigned char.
Parameters

- **buf** – Buffer to fill.
- **size** – Number of bytes to extract.

```
pdal::Log
```

class pdal::Log

`pdal::Log` (page 540) is a logging object that is provided by `pdal::Stage` (page 553) to facilitate logging operations.

Log stream operations

```
inline std::ostream *getLogStream()
```

Returns the stream object that is currently being used for log operations regardless of logging level of the instance.

```
std::ostream &get(LogLevel level = LogLevel::Info)
```

Returns the log stream given the logging level.

Parameters **level** – logging level to request. If the logging level asked for with `pdal::Log::get` (page 540) is less than the logging level of the `pdal::Log` (page 540) instance

```
void floatPrecision(int level)
```

Sets the floating point precision.

```
void clearFloat()
```

Clears the floating point precision settings of the streams.

Destructor

```
~Log()
```

The destructor will clean up its own internal log stream, but it will not touch one that is given via the constructor.
Logging level

**inline** LogLevel **getLevel** ()

*Returns* the logging level of the *pdal::Log* (page 540) instance

**inline** void **setLevel** (LogLevel *v*)

Sets the logging level of the *pdal::Log* (page 540) instance.

*Parameters* *v* – logging level to use for *get*() (page 540) comparison operations

**inline** void **setLeader** (const std::string &leader)

Set the leader string (deprecated).

*Parameters* [in] leader – Leader string.

**inline** void **pushLeader** (const std::string &leader)

Push the leader string onto the stack.

*Parameters* leader – Leader string

**inline** std::string **leader** () const

Get the leader string.

*Returns* The current leader string.

**inline** void **popLeader** ()

Pop the current leader string.

std::string **getLevelString** (LogLevel *v*) const

*Returns* A string representing the LogLevel

**Public Static Functions**

**static** LogPtr **makeLog** (std::string const &leaderString, std::string const &outputName, bool timing = false)

**static** LogPtr **makeLog** (std::string const &leaderString, std::ostream *v, bool timing = false)
pdal::Metadata
class pdal::Metadata

Public Functions
inline Metadata()
inline Metadata(const std::string &name)
inline MetadataNode (page 542) getNode() const

Public Static Functions
static std::string PDAL_DLL inferType (const std::string &val)

class pdal::MetadataNode

Public Functions
inline MetadataNode()
inline MetadataNode(const std::string &name)
inline MetadataNode (page 542) add (const std::string &name)
inline MetadataNode (page 542) addList (const std::string &name) const
inline MetadataNode (page 542) clone (const std::string &name) const
inline MetadataNode (page 542) add (MetadataNode (page 542) node)
inline MetadataNode (page 542) addList (MetadataNode (page 542) node)
inline MetadataNode (page 542) addEncoded (const std::string &name,
const unsigned char *buf,
size_t size, const std::string &descrip = std::string())
inline MetadataNode (page 542) addListEncoded (const std::string &name,
const unsigned char *buf,
size_t size, const std::string &descrip = std::string())
inline MetadataNode (page 542) addWith_Type (const std::string &name, 
            const std::string &value, 
            const std::string &type, 
            const std::string &de-
            scrip)

inline MetadataNode (page 542) add (const std::string &name, const double &value, const std::string &descrip = std::string(), size_t precision = 10)

template<typename T>
inline MetadataNode (page 542) add (const std::string &name, const T &value, const std::string &descrip = std::string())

template<typename T>
inline MetadataNode (page 542) addList (const std::string &name, const T &value, const std::string &descrip = std::string())

inline MetadataNode (page 542) addOrUpdate (const std::string &lname, 
            const double &value, 
            const std::string &de-
            scrip = std::string(), size_t precision = 10)

template<typename T>
inline MetadataNode (page 542) addOrUpdate (const std::string &lname, const T &value)

template<typename T>
inline MetadataNode (page 542) addOrUpdate (const std::string &lname, const T &value, const std::string &descrip)

inline MetadataNode (page 542) addOrUpdate (MetadataNode n)

inline std::string type () const

inline MetadataType kind () const

inline std::string name () const

template<typename T>
inline T (page 543) value () const

inline std::string value () const

inline std::string jsonValue () const
inline std::string description() const
inline MetadataNodeList children() const
inline MetadataNodeList children(const std::string &name) const
inline bool hasChildren() const
inline std::vector<std::string> childNames() const
inline operator bool() const
inline bool operator!() const
inline bool valid() const
inline bool empty() const
template<typename PREDICATE>
inline MetadataNode (page 542) find(PREDICATE (page 544) p) const
template<typename PREDICATE>
inline MetadataNodeList findChildren(PREDICATE (page 544) p)
template<typename PREDICATE>
inline MetadataNode (page 542) findChild(PREDICATE (page 544) p) const
inline MetadataNode (page 542) findChild(const char *s) const
inline MetadataNode (page 542) findChild(std::string s) const

pdal::Options

class pdal::Options

Public Functions

inline Options ()
inline explicit Options (const Option &opt)
void add (const Option &option)
void add (const Options (page 544) &options)
void addConditional (const Option &option)
void addConditional (const Options (page 544) &option)
void remove (const Option &option)
inline void replace (const Option &option)
**Public Static Functions**

*static Options* (page 544) **fromFile** *(const std::string &filename, bool throwOnError = true)*

**pdal::PointTable**

*Warning:* doxygen:Class: Cannot find class “pdal::PointTable” in doxygen xml output for project “api” from directory: doxygen/xml

**pdal::PointView**

*class* pdal::PointView: public pdal::PointContainer
Public Functions

PointView (const PointView (page 546)&) = delete

PointView (page 545) &operator= (const PointView (page 545)&) = delete

PointView (PointTableRef pointTable)

PointView (PointTableRef pointTable, const SpatialReference &srs)

virtual ~PointView ()

PointViewIter begin ()

PointViewIter end ()

inline int id () const

inline point_count_t size () const

inline bool empty () const

inline void appendPoint (const PointView (page 545) &buffer, PointId id)

inline void append (const PointView (page 545) &buf)

inline PointViewPtr makeNew () const

    Return a new point view with the same point table as this point buffer.

inline PointRef point (PointId id)

    template<class T>

    inline T (page 546) getFieldAs (Dimension (page 528)::Id dim, PointId pointIndex) const

    inline void getField (char *pos, Dimension (page 528)::Id d, Dimension (page 528)::Type (page 528) type, PointId id) const

    template<typename T>

    void setField (Dimension (page 528)::Id dim, PointId idx, T (page 546) val)

    inline void setField (Dimension (page 528)::Id dim, Dimension (page 528)::Id d, Dimension (page 528)::Type (page 528) type, PointId idx, const void *val)

    template<typename T>

    inline virtual bool compare (Dimension (page 528)::Id dim, PointId id1, PointId id2) const

    inline virtual bool compare (Dimension (page 528)::Id dim, PointId id1, PointId id2) const

    void calculateBounds (BOX2D (page 519) &box) const

        Returns  a cumulated bounds of all points in the PointView (page 545).
Note: This method requires that an X, Y, and Z dimension be available, and that it can be casted into a double data type using the pdal::Dimension::applyScaling() method. Otherwise, an exception will be thrown.

```cpp
void calculateBounds (BOX3D (page 523) &box) const
void dump (std::ostream &ostr) const
inline bool hasDim (Dimension (page 528)::Id id) const
inline std::string dimName (Dimension (page 528)::Id id) const
inline Dimension (page 528)::IdList dims () const
inline std::size_t pointSize () const
inline std::size_t dimSize (Dimension (page 528)::Id id) const
inline Dimension (page 528)::Type (page 528) dimType (Dimension (page 528)::Id id) const
inline DimTypeList dimTypes () const
inline virtual PointLayoutPtr layout () const
inline PointTableRef table () const
inline SpatialReference spatialReference () const
inline void getPackedPoint (const DimTypeList &dims, PointId idx, char *buf) const
    Fill a buffer with point data specified by the dimension list.
    Parameters
    • [in] dims – List of dimensions/types to retrieve.
    • [in] idx – Index of point to get.
    • [in] buf – Pointer to buffer to fill.
inline void setPackedPoint (const DimTypeList &dims, PointId idx, const char *buf)
    Load the point buffer from memory whose arrangement is specified by the dimension list.
    Parameters
    • [in] dims – Dimension/types of data in packed order
    • [in] idx – Index of point to write.
```
• [in] buf – Packed data buffer.

`inline char *getPoint(PointId id)`
Provides access to the memory storing the point data.

Though this function is public, other access methods are safer and preferred.

`inline char *getOrAddPoint(PointId id)`
Provides access to the memory storing the point data.

Though this function is public, other access methods are safer and preferred.

`inline void clearTemps()`

`MetadataNode` (page 542) `toMetadata()` `const`

`void invalidateProducts()`

TriangularMesh `*createMesh(const std::string &name)`
Creates a mesh with the specified name.

**Parameters**
`name` – Name of the mesh.

**Returns**
Pointer to the new mesh. Null is returned if the mesh already exists.

TriangularMesh `*mesh(const std::string &name = "")`
Get a pointer to a mesh.

**Parameters**
`name` – Name of the mesh.

**Returns**
New mesh. Null is returned if the mesh already exists.

Rasterd `*createRaster(const std::string &name, const RasterLimits &limits, double noData = 0)`
Creates a raster with the specified name.

**Parameters**

• `name` – Name of the raster.

• `limits` – Limits of the raster to create.

**Returns**
Pointer to the new raster. Null is returned if the raster already exists.

Rasterd `*raster(const std::string &name = "")`
Get a pointer to a raster.

**Parameters**
`name` – Name of the raster.

**Returns**
Pointer to the New raster. Null

`KD3Index &build3dIndex()`

`KD2Index &build2dIndex()`
Friends

friend class plang::Invocation

pdal::ProgramArgs

class pdal::ProgramArgs

Parses command lines, provides validation and stores found values in bound variables.

Add arguments with add (page 549). When all arguments have been added, use parse (page 551) to validate command line and assign values to variables bound with add (page 549).

Public Functions

inline Arg &add(const std::string &name, const std::string description, std::string &var, std::string def)

Add a string argument to the list of arguments.

Parameters

• name – Name of argument. Argument names are specified as “longname[,shortname]”, where shortname is an optional one-character abbreviation.

• description – Description of the argument.

• var – Reference to variable to bind to argument.

• def – Default value of argument.

Returns Reference to the new argument.

inline Arg &add(const std::string &name, const std::string &description, std::vector<std::string> &var)

Add a list-based (vector) string argument.

Parameters

• name – Name of argument. Argument names are specified as “longname[,shortname]”, where shortname is an optional one-character abbreviation.

• description – Description of the argument.

• var – Reference to variable to bind to argument.

Returns Reference to the new argument.
inline bool set (const std::string &name) const
    Return whether the argument (as specified by it’s longname) had its value set
during parsing.

template<typename T>
inline Arg &add (const std::string &name, const std::string &description,
    std::vector<T> &var)
Add a list-based (vector) argument.

Parameters

• name – Name of argument. Argument names are specified as
  “longname[,shortname]”, where shortname is an optional
  one-character abbreviation.

• description – Description of the argument.

• var – Reference to variable to bind to argument.

Returns Reference to the new argument.

template<typename T>
inline Arg &add (const std::string &name, const std::string description,
    std::vector<T> &var, std::vector<T> def)
Add a list-based (vector) argument with a default.

Parameters

• name – Name of argument. Argument names are specified as
  “longname[,shortname]”, where shortname is an optional
  one-character abbreviation.

• description – Description of the argument.

• var – Reference to variable to bind to argument.

Returns Reference to the new argument.

template<typename T>
inline Arg &add (const std::string &name, const std::string description, T
    (page 550) &var, T def)
Add an argument to the list of arguments with a default.

Parameters

• name – Name of argument. Argument names are specified as
  “longname[,shortname]”, where shortname is an optional
  one-character abbreviation.

• description – Description of the argument.

• var – Reference to variable to bind to argument.
def – Default value of argument.

**Returns** Reference to the new argument.

template<typename T>
**inline** Arg & add(const std::string & name, const std::string description, T & var)

Add an argument to the list of arguments.

**Parameters**

- **name** – Name of argument. Argument names are specified as “longname[,shortname]”, where shortname is an optional one-character abbreviation.
- **description** – Description of the argument.
- **var** – Reference to variable to bind to argument.

**Returns** Reference to the new argument.

**inline** void parseSimple(std::vector<std::string> & s)

Parse a command line as specified by its argument vector.

No validation occurs and only argument value exceptions are raised, but assignments are made to bound variables where possible.

**Parameters** **s** – List of strings that constitute the argument list.

**inline** void parse(const std::vector<std::string> & s)

Parse a command line as specified by its argument list.

Parsing validates the argument vector and assigns values to variables bound to added arguments.

**Parameters** **s** – List of strings that constitute the argument list.

**inline** void addSynonym(const std::string & name, const std::string & synonym)

Add a synonym for an argument.

**Parameters**

- **name** – Longname of existing argument.
- **synonym** – Synonym for argument.

**inline** void reset()

Reset the state of all arguments and bound variables as if no parsing had occurred.

**inline** std::string commandLine() const

Return a string suitable for use in a “usage” line for display to users as help.

**inline** void dump(std::ostream & out, size_t indent, size_t totalWidth) const

Write a formatted description of arguments to an output stream.
Write a list of the names and descriptions of arguments suitable for display as help information.

Parameters
- **out** – Stream to which output should be written.
- **indent** – Number of characters to indent all text.
- **totalWidth** – Total width to assume for formatting output. Typically this is the width of a terminal window.

```cpp
inline void dump2 (std::ostream &out, size_t nameIndent, size_t descripIndent, size_t totalWidth) const
```

Write a verbose description of arguments to an output stream.

Each argument is on its own line. The argument’s description follows on subsequent lines.

Parameters
- **out** – Stream to which output should be written.
- **nameIndent** – Number of characters to indent argument lines.
- **descripIndent** – Number of characters to indent description lines.
- **totalWidth** – Total line width.

```cpp
inline void dump3 (std::ostream &out) const
```

Write a JSON array of arguments to an output stream.

Parameters **out** – Stream to which output should be written.

---

**pdal::Reader**

*pdal::Reader* (page 552) are classes that provided interfaces to various the various point cloud formats and hands them off to a PDAL pipeline in a common format that is described via the *pdal::Schema*.

**class Reader : public virtual pdal::Stage** (page 553)

Subclassed by *pdal::BpfReader, pdal::BufferReader, pdal::DbReader, pdal::E57Reader, pdal::EptReader, pdal::EsriReader, pdal::FauxReader, pdal::GDALReader, pdal::GeoWaveReader, pdal::HdfReader, pdal::IcebridgeReader, pdal::Ilvis2Reader, pdal::LasReader, pdal::MatlabReader, pdal::MbReader, pdal::MemoryViewReader, pdal::MrsidReader, pdal::OSGReader, pdal::ObjReader, pdal::OptechReader, pdal::PcdReader, pdal::PlyReader, pdal::PtsReader, pdal::QfitReader, pdal::RdbReader, pdal::RxpReader, pdal::SbetReader, pdal::TIndexReader, pdal::TerrasolidReader, pdal::TextReader, pdal::TileDBReader*
**pdal::Stage**

*pdal::Stage* (page 553) is the base class of *pdal::Filter* (page 536), *pdal::Reader* (page 552), and *pdal::MultiFilter* classes that implement the reading API in a PDAL pipeline.

**class pdal::Stage**

A stage performs the actual processing in PDAL.

Stages may read data, modify or filter read data, create metadata or write processed data.

Stages are linked with *setInput()* (page 553) into a pipeline. The pipeline is run with by calling in sequence *prepare()* (page 553) and *execute()* (page 553) on the stage at the end of the pipeline. PipelineManager can also be used to create and run a pipeline.

Subclassed by *pdal::Filter* (page 536), *pdal::Reader* (page 552), pdal::Streamable, *pdal::Writer* (page 575)

**Public Functions**

*Stage()*

*virtual ~Stage()*

*inline void setInput (Stage &input)*

Add a stage to the input list of this stage.

**Parameters input** – *Stage* (page 553) to use as input.

*inline void setProgressFd (int fd)*

Set a file descriptor to which progress information should be written.

**Parameters fd** – Progress file descriptor.

*QuickInfo preview ()*

Retrieve some basic point information without reading all data when possible.

Usually implemented only by Readers.

*void prepare (PointTableRef table)*

Prepare a stage for execution.

This function needs to be called on the terminal stage of a pipeline (linked set of stages) before *execute* (page 553) can be called. Prepare recurses through all input stages.

**Parameters table** – PointTable being used for stage pipeline.

*PointViewSet execute (PointTableRef table)*

Execute a prepared pipeline (linked set of stages).
This performs the action associated with the stage by executing the run function of each stage in depth first order. Each stage is run to completion (all points are processed) before the next stages is run.

**Parameters**

- **table** – Point table being used for stage pipeline. This must be the same table used in the `prepare` (page 553) function.

```cpp
inline virtual void execute (StreamPointTable &table)
```

```cpp
inline virtual bool pipelineStreamable () const
```

**Returns** Whether the pipeline is streamable.

```cpp
inline virtual const Stage (page 553) *findNonstreamable () const
```

**Returns** `nullptr` if the stage is streamable, a pointer to this stage otherwise.

```cpp
void setSpatialReference (SpatialReference const &srs)
```

Set the spatial reference of a stage.

Set the spatial reference that will override that being carried by the `PointView` (page 545) being processed. This is usually used when reprojecting data to a new spatial reference. The stage spatial reference will be carried by PointViews processes by this stage to subsequent stages.

If called by a `Reader` (page 552) whose spatial reference has been set with option ‘spatialreference’ or ‘override_srs’, then this function will have no effect.

**Parameters**

- **srs** – Spatial reference to set.

```cpp
const SpatialReference &getSpatialReference () const
```

Get the spatial reference of the stage.

Get the spatial reference that will override that being carried by the `PointView` (page 545) being processed. This is usually used when reprojecting data to a new spatial reference. The stage spatial reference will be carried by PointViews processes by this stage to subsequent stages.

**Returns** The stage’s spatial reference.

```cpp
inline void setOptions (Options (page 544) options)
```

Set a stage’s options.

Set the options on a stage, clearing all previously set options.

**Parameters**

- **options** – `Options` (page 544) to set.

```cpp
void addConditionalOptions (const Options (page 544) &opts)
```

Add options if an option with the same name doesn’t already exist on the stage.
Parameters **opts** – *Options* (page 544) to add.

```cpp
void addAllArgs(const ProgramArgs &args)
```

Add a stage’s options to a *ProgramArgs* (page 549) set.

Parameters **args** – *ProgramArgs* (page 549) to add to.

### inline void addOptions(const Options &opts)

Add options to the existing option set.

Parameters **opts** – *Options* (page 544) to add.

### inline void removeOptions(const Options &opts)

Remove options from a stage’s option set.

Parameters **opts** – *Options* (page 544) to remove.

### inline void setLog(const LogPtr &log)

Set the stage’s log.

Parameters **log** – *Log* (page 540) pointer.

### inline virtual LogPtr log() const

Return the stage’s log pointer.

### void startLogging() const

Push the stage’s leader into the log.

### void stopLogging() const

Pop the stage’s leader from the log.

### inline bool isDebug() const

Determine whether the stage is in debug mode or not.

Returns The stage’s debug state.

### virtual std::string getName() const = 0

Return the name of a stage.

Returns The stage’s name.

### inline void setTag(const std::string &tag)

Set a specific tag name.

### inline virtual std::string tag() const

Return the tag name of a stage.

Returns The tag name.

### inline std::vector<Stage*> &getInputs()

Return a list of the stage’s inputs.

Returns A vector pointers to input stages.
inline MetadataNode (page 542) getMetadata() const
Get the stage’s metadata node.

Returns Stage (page 553)’s metadata.

void serialize(MetadataNode (page 542) root, PipelineWriter::TagMap &tags) const
Serialize a stage by inserting apporpritate data into the provided MetadataNode (page 542).

Used to dump a pipeline specification in a portable format.

Parameters
- root – Node to which a stages metadata should be added.
- tags – Pipeline writer’s current list of stage tags.

Public Static Functions

static bool parseName(std::string o, std::string::size_type &pos)
Parse a stage name from a string.

Return the name and update the position in the input string to the end of the stage name.

Parameters
- o – Input string to parse.
- pos – Parsing start/end position.

Returns Whether the parsed name is a valid stage name.

static bool parseTagName(std::string o, std::string::size_type &pos)
Parse a tag name from a string.

Return the name and update the position in the input string to the end of the tag name.

Parameters
- o – Input string to parse.
- pos – Parsing start/end position.
- tag – Parsed tag name.

Returns Whether the parsed name is a valid tag name.
class pdal::StageFactory

This class provides a mechanism for creating Stage (page 553) objects given a driver name.

Creates stages are owned by the factory and destroyed when the factory is destroyed. Stages can be explicitly destroyed with destroyStage() (page 557) if desired.

Note: Stage (page 553) creation is thread-safe.

Public Functions

StageFactory (bool ignored = true)
Create a stage factory.

Parameters ignored – Ignored argument.

Stage (page 553) *createStage (const std::string &type)
Create a stage and return a pointer to the created stage.

The factory takes ownership of any successfully created stage.

Parameters stage_name – Type of stage to by created.

Returns Pointer to created stage.

void destroyStage (Stage (page 553) *stage)
Destroy a stage created by this factory.

This doesn’t need to be called unless you specifically want to destroy a stage as all stages are destroyed when the factory is destroyed.

Parameters stage – Pointer to stage to destroy.

Public Static Functions

static std::string inferReaderDriver (const std::string &filename)
Infer the reader to use based on a filename.

Find the default reader for a file.

Parameters

• filename – Filename that should be analyzed to determine a driver.
• **filename** – Filename for which to infer a reader.

**Returns**  Driver name or empty string if no reader can be inferred from the filename.

**Returns**  Name of the reader driver associated with the file.

```cpp
static std::string inferWriterDriver (const std::string &filename)
```

Find the default writer for a file.

**Parameters**  filename – Filename for which to infer a writer.

**Returns**  Driver name or empty string if no writer can be inferred from the filename.

**Returns**  Name of the writer driver associated with the file.

**pdal::Utils**

`cpp:namespace: pdal::Utils` is a set of utility functions.

```cpp
namespace pdal::Utils
```

#### Typedefs

```cpp
using BacktraceEntries = std::deque<BacktraceEntry>
```

#### Functions

```cpp
std::string toJSON (const MetadataNode &m)
```

```cpp
void toJSON (const MetadataNode &m, std::ostream &o)
```

```cpp
std::ostream *createFile (const std::string &path, bool asBinary)
```

Create a file (may be on a supported remote filesystem).

**Parameters**

• **path** – Path to file to create.

• **asBinary** – Whether the file should be written in binary mode.

**Returns**  Pointer to the created stream, or NULL.

```cpp
bool isRemote (const std::string &path)
```

Open a file (potentially on a remote filesystem).

**Parameters**
• **path** – Path (potentially remote) of file to open.
• **asBinary** – Whether the file should be opened binary.

**Returns**  Pointer to stream opened for input.

```cpp
std::string fetchRemote(const std::string &path)
std::istream *openFile(const std::string &path, bool asBinary)
void closeFile(std::ostream *out)
```

Close an output stream.

**Parameters**  `out` – Stream to close.

```cpp
void closeFile(std::istream *in)
```

Close an input stream.

**Parameters**  `out` – Stream to close.

```cpp
bool fileExists(const std::string &path)
```

Check to see if a file exists.

**Parameters**  `path` – Path to file.

**Returns**  Whether the file exists or not.

```cpp
double computeHausdorff(PointViewPtr srcView, PointViewPtr candView)
std::string dllDir()
```

```cpp
inline void printError(const std::string &s)
```

```cpp
inline double toDouble(const Everything &e, Dimension (page 528)::Type (page 528) type)
```

```cpp
template<typename INPUT>
inline Everything extractDim(INPUT &ext, Dimension (page 528)::Type (page 528) type)
```

```cpp
template<typename OUTPUT>
inline Everything insertDim(OUTPUT &ins, Dimension (page 528)::Type (page 528) type, const Everything &e)
```

```cpp
inline MetadataNode toMetadata(const BOX2D &bounds)
```

```cpp
inline MetadataNode toMetadata(const BOX3D &bounds)
```

```cpp
inline int openProgress(const std::string &filename)
```

```cpp
inline void closeProgress(int fd)
```

```cpp
inline void writeProgress(int fd, const std::string &type, const std::string &text)
```
std::vector< std::string > PDAL_DLL maybeGlob (const std::string &path)

```cpp
template<>
inline StatusWithReason (page 574) fromString (const std::string &s,
    Eigen::MatrixXd &matrix)
```

```cpp
template<>
inline StatusWithReason (page 574) fromString (const std::string &s, SrsBounds &srsBounds)
```

```cpp
template<typename CONTAINER, typename VALUE>
bool contains (const CONTAINER (page 560) &cont, const VALUE (page 560) &val)
    Determine if a container contains a value.

    Parameters
    • cont – Container.
    • val – Value.

    Returns true if the value is in the container, false otherwise.

```cpp
template<typename KEY, typename VALUE>
bool contains (const std::map<KEY (page 560), VALUE (page 560)> &c, const KEY (page 560) &v)
    Determine if a map contains a key.

    Parameters
    • c – Map.
    • v – Key value.

    Returns true if the value is in the container, false otherwise.
```

```cpp
template<typename CONTAINER, typename VALUE>
void remove (CONTAINER (page 560) &cont, const VALUE (page 560) &val)
    Remove all instances of a value from a container.

    Parameters
    • cont – Container.
    • v – Value to remove.
```

```cpp
template<typename CONTAINER, typename PREDICATE>
void remove_if (CONTAINER (page 560) &cont, PREDICATE (page 560) p)
    Remove all instances matching a unary predicate from a container.

    Parameters
    • cont – Container.
• \( p \) – Predicate indicating whether a value should be removed.

```cpp
PDAL_DLL std::vector< std::string > backtrace ()
```
Generate a backtrace as a list of strings.

**Returns** List of functions at the point of the call.

*BacktraceEntries* (page 558) `backtraceImpl ()`

```cpp
template<class T> PDAL_DLL const T & clamp (const T &t, const T &minimum, const T &maximum)
```
Clamp value to given bounds.

Clamps the input value t to bounds specified by min and max. Used to ensure that row and column indices remain within valid bounds.

**Parameters**

• \( t \) – the input value.

• \( \text{min} \) – the lower bound.

• \( \text{max} \) – the upper bound.

**Returns** the value to clamped to the given bounds.

```cpp
PDAL_DLL void random_seed (unsigned int seed)
```
Set a seed for random number generation.

**Parameters** seed – Seed value.

```cpp
PDAL_DLL double random (double minimum, double maximum)
```
Generate a random value in the range \([\text{minimum}, \text{maximum}]\).

**Parameters**

• \( \text{minimum} \) – Lower value of range for random number generation.

• \( \text{maximum} \) – Upper value of range for random number generation.

```cpp
inline PDAL_DLL bool compare_approx (double v1, double v2, double tolerance)
```
Determine if two values are within a particular range of each other.

**Parameters**

• \( v1 \) – First value to compare.

• \( v2 \) – Second value to compare.

• \( \text{tolerance} \) – Maximum difference between \( v1 \) and \( v2 \)

```cpp
inline double sround (double r)
```
Round double value to nearest integral value.

**Parameters** \( r \) – Value to round

**Returns** Rounded value
inline std::string tolower (const std::string &s)
    Convert a string to lowercase.

    Returns  Converted string.

inline std::string toupper (const std::string &s)
    Convert a string to uppercase.

    Returns  Converted string.

inline bool iequals (const std::string &s, const std::string &s2)
    Compare strings in a case-insensitive manner.

    Parameters
    • s – First string to compare.
    • s2 – Second string to compare.

    Returns  Whether the strings are equal.

inline bool startsWith (const std::string &s, const std::string &prefix)
    Determine if a string starts with a particular prefix.

    Parameters
    • s – String to check for prefix.
    • prefix – Prefix to search for.

    Returns  Whether the string begins with the prefix.

inline bool endsWith (const std::string &s, const std::string &postfix)
    Determine if a string ends with a particular postfix.

    Parameters
    • s – String to check for postfix.
    • postfix – Postfix to search for.

    Returns  Whether the string ends with the postfix.

inline int cksum (char *buf, size_t size)
    Generate a checksum that is the integer sum of the values of the bytes in a buffer.

    Parameters
    • buf – Pointer to buffer.
    • size – Size of buffer.

    Returns  Generated checksum.

PDAL_DLL int getenv (std::string const &name, std::string &val)
    Fetch the value of an environment variable.
Parameters

• **name** – Name of environment variable.

• **name** – Value of the environment variable if it exists, empty otherwise.

**Returns** 0 on success, -1 on failure

```cpp
PDAL_DLL int setenv (const std::string &env, const std::string &val)
```
Set the value of an environment variable.

**Parameters**

• **env** – Name of environment variable.

• **val** – Value of environment variable.

**Returns** 0 on success, -1 on failure

```cpp
PDAL_DLL int unsetenv (const std::string &env)
```
Clear the value of an environment variable.

**Parameters** **env** – Name of the environment variable to clear.

**Returns** 0 on success, -1 on failure

```cpp
PDAL_DLL void eatwhitespace (std::istream &s)
```
Skip stream input until a non-space character is found.

**Parameters** **s** – Stream to process.

```cpp
PDAL_DLL void trimLeading (std::string &s)
```
Remove whitespace from the beginning of a string.

**Parameters** **s** – String to be trimmed.

```cpp
PDAL_DLL void trimTrailing (std::string &s)
```
Remove whitespace from the end of a string.

**Parameters** **s** – String to be trimmed.

```cpp
inline void trim (std::string &s)
```
Remove whitespace from the beginning and end of a string.

**Parameters** **s** – String to be trimmed.

```cpp
PDAL_DLL bool eatcharacter (std::istream &s, char x)
```
If specified character is at the current stream position, advance the stream position by 1.

**Parameters**

• **s** – Stream to insect.

• **x** – Character to check for.
**PDAL: Point cloud Data Abstraction Library, Release 2.2.0**

**Returns** true if the character is at the current stream position, false otherwise.

**PDAL_DLL std::string base64_encode (const unsigned char *buf, size_t size)**

Convert a buffer to a string using base64 encoding.

**Parameters**
- **buf** – Pointer to buffer to encode.
- **size** – Size of buffer.

**Returns** Encoded buffer.

**PDAL_DLL std::string base64_encode (std::vector<uint8_t> const &bytes)**

Convert a buffer to a string using base64 encoding.

**Parameters** **bytes** – Pointer to buffer to encode.

**Returns** Encoded buffer.

**PDAL_DLL std::vector< uint8_t > base64_decode (std::string const &input)**

Decode a base64-encoded string into a buffer.

**Parameters** **input** – String to decode.

**Returns** Buffer containing decoded string.

**PDAL_DLL FILE * portable_popen (const std::string &command, const std::string &mode)**

Start a process to run a command and open a pipe to which input can be written and from which output can be read.

**Parameters** **command** – Command to run in subprocess. \mode Either ‘r’, ‘w’ or ‘r+’ to specify if the pipe should be opened as read-only, write-only or read-write.

**Returns** Pointer to FILE for input/output from the subprocess.

**PDAL_DLL int portable_pclose (FILE *fp)**

Close file opened with portable_popen (page 564).

**Parameters** **fp** – Pointer to file to close.

**Returns** 0 on success, -1 on failure.

**PDAL_DLL int run_shell_command (const std::string &cmd, std::string &output)**

Create a subprocess and set the standard output of the command into the provided output string.

**Parameters**
- **cmd** – Command to run.
- **output** – String to which output from the command should be written,
PDAL_DLL std::string replaceAll (std::string input, const std::string &replaceWhat, const std::string &replaceWithWhat)
Replace all instances of one string found in the input with another value.

Parameters

• input – Input string to modify.
• replaceWhat – Token to locate in input string.
• replaceWithWhat – Replacement for found tokens.

Returns Modified version of input string.

PDAL_DLL std::vector< std::string > wordWrap (std::string const &inputString, size_t lineLength, size_t firstLength=0)
Break a string into a list of strings to not exceed a specified length.

Whitespace is condensed to a single space and each string is free of whitespace at the beginning and end when possible. Optionally, a line length for the first line can be different from subsequent lines.

Parameters

• inputString – String to split into substrings.
• lineLength – Maximum length of substrings.
• firstLength – When non-zero, the maximum length of the first substring. When zero, the first firstLength is assigned the value provided in lineLength.

Returns List of substrings generated from the input string.

PDAL_DLL std::vector< std::string > wordWrap2 (std::string const &inputString, size_t lineLength, size_t firstLength=0)
Break a string into a list of strings to not exceed a specified length.

The concatenation of the returned substrings will yield the original string. The algorithm attempts to break the original string such that each substring begins with a word.

Parameters

• inputString – String to split into substrings.
• lineLength – Maximum length of substrings.
• firstLength – When non-zero, the maximum length of the first substring. When zero, the first firstLength is assigned the value provided in lineLength.

Returns List of substrings generated from the input string.

PDAL_DLL std::string escapeJSON (const std::string &s)
Add escape characters or otherwise transform an input string so as to be a valid JSON string.
Parameters s – Input string.

Returns Valid JSON version of input string.

PDAL_DLL std::string demangle (const std::string &s)
Demangle a C++ symbol into readable form.
Demangle strings using the compiler-provided demangle function.

Parameters

• s – String to demangle.

• [in] s – String to be demangled.

Returns Demangled symbol.

Returns Demangled string

PDAL_DLL int screenWidth ()
Return the screen width of an associated tty.

Returns The tty screen width or 80 if on Windows or it can’t be
determined.

PDAL_DLL std::string escapeNonprinting (const std::string &s)
Escape non-printing characters by using standard notation (i.e.
) or hex notation (\x10) as as necessary.

Parameters s – String to modify.

Returns Copy of input string with non-printing characters converted to
printable notation.

PDAL_DLL double normalizeLongitude (double longitude)
Normalize longitude so that it’s between (-180, 180].

Parameters longitude – Longitude to normalize.

Returns Normalized longitude.

PDAL_DLL std::string hexDump (const char *buf, size_t count)
Convert an input buffer to a hexadecimal string representation similar to the output
of the UNIX command ‘od’.
This is mostly used as an occasional debugging aid.

Parameters

• buf – Point to buffer to dump.

• count – Size of buffer.

Returns Buffer converted to hex string.
template<typename PREDICATE> PDAL_DLL std::string::size_type extract (const std::string &s, std::string::size_type p, PREDICATE pred)
Count the number of characters in a string that meet a predicate.

Parameters
- s – String in which to start counting characters.
- p – Position in input string at which to start counting.
- pred – Unary predicate that tests a character.

Returns Then number of characters matching the predicate.

inline PDAL_DLL std::string::size_type extractSpaces (const std::string &s, std::string::size_type p)
Count the number of characters spaces in a string at a position.

Parameters
- s – String in which to start counting characters.
- p – Position in input string at which to start counting.

Returns Then number of space-y characters matching the predicate.

template<typename PREDICATE> PDAL_DLL std::vector< std::string > split (const std::string &s, PREDICATE p)
Split a string into substrings based on a predicate.
Characters matching the predicate are discarded.

Parameters
- s – String to split.
- p – Predicate returns true if a char in a string is a split location.

Returns Substrings.

template<typename PREDICATE> PDAL_DLL std::vector< std::string > split2 (const std::string &s, PREDICATE p)
Split a string into substrings.
Characters matching the predicate are discarded, as are empty strings otherwise produced by split) (page 567).

Parameters
- s – String to split.
- p – Predicate returns true if a char in a string is a split location.

Returns Vector of substrings.

inline PDAL_DLL std::vector< std::string > split (const std::string &s, char tChar)
Split a string into substrings based a splitting character.
The splitting characters are discarded.
Parameters

- **s** – String to split.
- **p** – Character indicating split positions.

**Returns**  Substrings.

```cpp
inline PDAL_DLL std::vector< std::string > split2 (const std::string &s, char tChar)
```

Split a string into substrings based a splitting character.

The splitting characters are discarded as are empty strings otherwise produced by `split()` (page 567).

Parameters

- **s** – String to split.
- **p** – Character indicating split positions.

**Returns**  Substrings.

```cpp
PDAL_DLL std::vector< std::string > simpleWordexp (const std::string &s)
```

Perform shell-style word expansion (break a string into arguments).

This only does simple handling of quoted values and backslashes and doesn’t support fancier shell behavior. Use the real wordexp() if you need all that. The behavior of escaped values in a string was surprising to me, so try the shell first if you think you’ve found a problem.

Parameters  **s** – Input string to parse.

**Returns**  List of arguments.

```cpp
template<typename T>
std::string typeidName ()
```

Return a string representation of a type specified by the template argument.

**Returns**  String representation of the type.

```cpp
inline RedirectStream (page 574) redirect (std::ostream &out, std::ostream &dst)
```

Redirect a stream to some other stream, by default a null stream.

Parameters

- **out** – Stream to redirect.
- **dst** – Destination stream.

**Returns**  Context for stream restoration (see `restore()` (page 569)).

```cpp
inline RedirectStream (page 574) redirect (std::ostream &out)
```

Redirect a stream to a null stream.

Parameters  **out** – Stream to redirect.
Returns Context for stream restoration (see `restore()` (page 569)).

**inline** `RedirectStream` (page 574) `redirect` (std::ostream &`out`, const std::string &`file`)

Redirect a stream to some file.

**Parameters**

- **out** – Stream to redirect.
- **file** – Name of file where stream should be redirected.

**Returns** Context for stream restoration (see `restore()` (page 569)).

**inline** void `restore` (std::ostream &`out`, RedirectStream &`redir`)

Restore a stream redirected with `redirect()` (page 568).

**Parameters**

- **out** – Stream to be restored.
- **redir** – RedirectStream (page 574) returned from corresponding `redirect()` (page 568) call.

**template<typename T_OUT>**

bool `inRange` (double `in`)

Determine whether a double value may be safely converted to the given output type without over/underflow.

If the output type is integral the input will be rounded before being tested.

**Parameters** `in` – Value to range test.

**Returns** Whether value can be safely converted to template type.

**template<typename T_IN, typename T_OUT>**

bool `inRange` (T_IN `in`)

Determine whether a value may be safely converted to the given output type without over/underflow.

If the output type is integral and different from the input type, the value will be rounded before being tested.

**Parameters** `in` – Value to range test.

**Returns** Whether value can be safely converted to template type.

**template<typename T_IN, typename T_OUT>**

bool `numericCast` (T_IN `in`, T_OUT &`out`)

Convert a numeric value from one type to another.

Floating point values are rounded to the nearest integer before a conversion is attempted.

**Parameters**
• \texttt{in} – Value to convert.
• \texttt{out} – Converted value.

**Returns** true if the conversion was successful, false if the datatypes/input value don’t allow conversion.

```
template<>
inline bool numericCast (double in, float &out)
    Convert a numeric value from double to float.

    Specialization to handle NaN.

    **Parameters**
    • \texttt{in} – Value to convert.
    • \texttt{out} – Converted value.

    **Returns** true if the conversion was successful, false if the datatypes/input value don’t allow conversion.
```

```
template<typename T>
std::string toString (const T &from)
    Convert a value to its string representation by writing to a stringstream.

    **Parameters** \texttt{from} – Value to convert.

    **Returns** String representation.

inline std::string toString (bool from)
    Convert a bool to a string.

inline std::string toString (double from, size_t precision = 10)
    Convert a double to string with a precision of 10 decimal places.

    **Parameters** \texttt{from} – Value to convert.

    **Returns** String representation of numeric value.

inline std::string toString (float from)
    Convert a float to string with a precision of 10 decimal places.

    **Parameters** \texttt{from} – Value to convert.

    **Returns** String representation of numeric value.

inline std::string toString (long long from)
    Convert a long long int to string.

    **Parameters** \texttt{from} – Value to convert.

    **Returns** String representation of numeric value.

inline std::string toString (unsigned long from)
    Convert an unsigned long long int to string.
Parameters \texttt{from} – Value to convert.

\textbf{Returns} String representation of numeric value.

\texttt{inline std::string toString (long from)}

Convert a long int to string.

\textbf{Parameters} \texttt{from} – Value to convert.

\textbf{Returns} String representation of numeric value.

\texttt{inline std::string toString (unsigned int from)}

Convert an unsigned int to string.

\textbf{Parameters} \texttt{from} – Value to convert.

\textbf{Returns} String representation of numeric value.

\texttt{inline std::string toString (int from)}

Convert an int to string.

\textbf{Parameters} \texttt{from} – Value to convert.

\textbf{Returns} String representation of numeric value.

\texttt{inline std::string toString (unsigned short from)}

Convert an unsigned short to string.

\textbf{Parameters} \texttt{from} – Value to convert.

\textbf{Returns} String representation of numeric value.

\texttt{inline std::string toString (short from)}

Convert a short int to string.

\textbf{Parameters} \texttt{from} – Value to convert.

\textbf{Returns} String representation of numeric value.

\texttt{inline std::string toString (char from)}

Convert a char (treated as numeric) to string.

\textbf{Parameters} \texttt{from} – Value to convert.

\textbf{Returns} String representation of numeric value.

\texttt{inline std::string toString (unsigned char from)}

Convert an unsigned char (treated as numeric) to string.

\textbf{Parameters} \texttt{from} – Value to convert.

\textbf{Returns} String representation of numeric value.

\texttt{inline std::string toString (signed char from)}

Convert a signed char (treated as numeric) to string.

\textbf{Parameters} \texttt{from} – Value to convert.
**Returns** String representation of numeric value.

```cpp
template<typename T>
StatusWithReason (page 574) fromString (const std::string &from, T &to)
```

Convert a string to a value by reading from a string stream.

**Parameters**

- **from** – String to convert.
- **to** – Converted value.

**Returns** `true` if the conversion was successful, `false` otherwise.

```cpp
template<>
inline StatusWithReason (page 574) fromString (const std::string &from, std::string &to)
```

**Parameters** `to` – Converted numeric value.

**Returns** `true` if the conversion was successful, `false` otherwise.

```cpp
template<>
inline StatusWithReason (page 574) fromString (const std::string &s, unsigned char &to)
```

**Parameters** `to` – Converted numeric value.

**Returns** `true` if the conversion was successful, `false` otherwise.

```cpp
template<>
inline StatusWithReason (page 574) fromString (const std::string &s, signed char &to)
```

**Parameters** `to` – Converted numeric value.

**Returns** `true` if the conversion was successful, `false` otherwise.
template<>  
**inline** *StatusWithReason* (page 574) *fromString* (*const* std::string &*s*, double &*d*)

Specialization conversion from string to double to handle Nan.

**Parameters**

- *s* – String to be converted.
- *d* – Converted value.

**Returns** true if the conversion was successful, false otherwise.

```cpp
template<typename E>
std::underlying_type<E> (page 573)::type toNative (E (page 573) *e*)
```

Return the argument cast to its underlying type.

**Parameters** *e* – Variable for which to find the underlying type.

**Returns** Converted variable.

```cpp
template<>  
**inline** *StatusWithReason* (page 574) *fromString* (*const* std::string &*from*,
         pdal::i3s::Obb &*obb*)
```

```cpp
template<>  
**inline** *StatusWithReason* (page 574) *fromString* (*const* std::string &*from*,
         pdal::expr::AssignStatement &*stmt*)
```

```cpp
template<>  
**inline** *StatusWithReason* (page 574) *fromString* (*const* std::string &*from*,
         pdal::expr::ConditionalExpression &*expr*)
```

```cpp
*StatusWithReason* (page 574) *fromString* (*const* std::string &*s*, EptBounds &*bounds*)
```

**Variables**

```cpp
const char dynamicLibExtension[] = ".so"
```

```cpp
const char dirSeparator = '/'
```

```cpp
const char pathListSeparator = ':'
```

```cpp
struct BacktraceEntry
     #include <BacktraceImpl.hpp>
```

14.3. API  573
Public Functions

inline BacktraceEntry()

Public Members

std::string libname
void *addr
std::string symname
int offset

class StatusWithReason
#include <Utils.hpp>

Public Functions

inline StatusWithReason()
inline StatusWithReason(bool ok)
StatusWithReason(int code)
inline StatusWithReason(int code, const std::string &what)
inline int code() const
inline operator bool() const
inline std::string what() const

struct RedirectStream
#include <Utils.hpp>

Public Functions

inline RedirectStream()
Public Members

std::ofstream *m_out
std::streambuf *m_buf
std::unique_ptr<NullOStream> m_null

pdal::Writer

class Writer : public virtual pdal::Stage (page 553)

A Writer (page 575) is a terminal stage for a PDAL pipeline.

It usually writes output to a file, but this isn’t a requirement. The class provides support
for some operations common for producing point output.

Subclassed by pdal::DbWriter, pdal::E57Writer, pdal::EptAddonWriter, pdal::FbxWriter,
pdal::FlexWriter, pdal::GeoWaveWriter, pdal::GltfWriter, pdal::MatlabWriter,
pdal::NullWriter, pdal::PcdWriter, pdal::PlyWriter, pdal::RasterWriter, pdal::SbetWriter,
pdal::TextWriter, pdal::TileDBWriter

14.3.2 libLAS C API to PDAL transition guide

Author Vaclav Petras

Contact wenzeslaus@gmail.com

Date 09/04/2015

This page shows how to port code using libLAS C API to PDAL API (which is C++). The new
code is not using full power of PDAL but it uses just what is necessary to read content of a
LAS file.

Includes

libLAS include:

```
#include <liblas/capi/liblas.h>
```

For PDAL, in addition to PDAL headers, we also include standard headers which will be useful later:

```
#include <memory>
#include <pdal/PointTable.hpp>
#include <pdal/PointView.hpp>
#include <pdal/LasReader.hpp>
```

(continues on next page)
#include <pdal/LasHeader.hpp>
#include <pdal/Options.hpp>

Initial steps

Opening the dataset in libLAS:

```c
LASReaderH LAS_reader;
LASHeaderH LAS_header;
LASSRSH LAS_srs;
LAS_reader = LASReader_Create(in_opt->answer);
LAS_header = LASReader_GetHeader(LAS_reader);
```

The higher level of abstraction in PDAL requires a little bit more code for the initial steps:

```c
pdal::Option las_opt("filename", in_opt->answer);
pdal::Options las_opts;
las_opts.add(las_opt);
pdal::PointTable table;
pdal::LasReader las_reader;
las_reader.setOptions(las_opts);
las_reader.prepare(table);
pdal::PointViewSet point_view_set = las_reader.execute(table);
pdal::PointViewPtr point_view = *point_view_set.begin();
pdal::Dimension::IdList dims = point_view->dims();
pdal::LasHeader las_header = las_reader.header();
```

The PDAL code is also different in the way that we read all the data right away while in libLAS we just open the file. To make use of other readers supported by PDAL, see StageFactory class.

The test if the file was loaded successfully, the test of the header pointer was used with libLAS:

```c
if (LAS_header == NULL) {
    /* fail */
}
```

In general, PDAL will throw a `pdal_error` exception in case something is wrong and it can’t recover such in the case when the file can’t be opened. To handle the exceptional state by yourself, you can wrap the code in `try-catch` block:

```c
try {
    /* actual code */
} catch {
```
Dataset properties

We assume we defined all the following variables as double.

The general properties from the LAS file are retrieved from the header in libLAS:

```c
scale_x = LASHeader_GetScaleX(LAS_header);
scale_y = LASHeader_GetScaleY(LAS_header);
scale_z = LASHeader_GetScaleZ(LAS_header);

offset_x = LASHeader_GetOffsetX(LAS_header);
offset_y = LASHeader_GetOffsetY(LAS_header);
offset_z = LASHeader_GetOffsetZ(LAS_header);

xmin = LASHeader_GetMinX(LAS_header);
xmax = LASHeader_GetMaxX(LAS_header);
ymin = LASHeader_GetMinY(LAS_header);
ymax = LASHeader_GetMaxY(LAS_header);
```

And the same applies PDAL:

```c
scale_x = las_header.scaleX();
scale_y = las_header.scaleY();
scale_z = las_header.scaleZ();

offset_x = las_header.offsetX();
offset_y = las_header.offsetY();
offset_z = las_header.offsetZ();

xmin = las_header.minX();
xmax = las_header.maxX();
ymin = las_header.minY();
ymax = las_header.maxY();
```

The point record count in libLAS:

```c
unsigned int n_features = LASHeader_GetPointRecordsCount(LAS_header);
```

is just point count in PDAL:

```c
unsigned int n_features = las_header.pointCount();
```

WKT of a spatial reference system is obtained from the header in libLAS:
LAS_srs = LASHeader_GetSRS(LAS_header);
char* projstr = LASSRS_GetWKT_CompoundOK(LAS_srs);

In PDAL, spatial reference is part of the PointTable:

char* projstr = table.spatialRef().getWKT(pdal::SpatialReference::eCompoundOK).c_str();

Whether the time or color is supported by the LAS format, one would have to determine from the format ID in libLAS:

las_point_format = LASHeader_GetDataFormatId(LAS_header);
have_time = (las_point_format == 1 ... 

In PDAL, there is a convenient function for it in the header:

have_time = las_header.hasTime();
have_color = las_header.hasColor();

The presence of color, time and other dimensions can be also determined with:

pdal::Dimension::IdList dims = point_view->dims();

Iterating over points

libLAS:

while ((LAS_point = LASReader_GetNextPoint(LAS_reader)) != NULL) {
    // ...
}

PDAL:

for (pdal::PointId idx = 0; idx < point_view->size(); ++idx) {
    // ...
}
Point validity

The correct usage of libLAS required to test point validity:

```c
LASPoint_IsValid(LAS_point)
```

In PDAL, there is no need to do that and the caller can assume that all the points provided by PDAL are valid.

Coordinates

libLAS:

```c
x = LASPoint_GetX(LAS_point);
y = LASPoint_GetY(LAS_point);
z = LASPoint_GetZ(LAS_point);
```

In PDAL, point coordinates are one of the dimensions:

```c
using namespace pdal::Dimension;
x = point_view->getFieldAs<double>(Id::X, idx);
y = point_view->getFieldAs<double>(Id::Y, idx);
z = point_view->getFieldAs<double>(Id::Z, idx);
```

Thanks to `using namespace pdal::Dimension` we can just write `Id::X` etc.

Returns

libLAS:

```c
int return_no = LASPoint_GetReturnNumber(LAS_point);
int n_returns = LASPoint_GetNumberOfReturns(LAS_point);
```

PDAL:

```c
int return_no = point_view->getFieldAs<int>(Id::ReturnNumber, idx);
int n_returns = point_view->getFieldAs<int>(Id::NumberOfReturns, idx);
```
### Classes

**libLAS:**

```cppint point_class = (int) LASPoint_GetClassification(LAS_point);
```

**PDAL:**

```cppint point_class = point_view->getFieldAs<int>(Id::Classification, idx);
```

### Color

**libLAS:**

```cpp
LASColorH LAS_color = LASPoint_GetColor(LAS_point);
int red = LASColor_GetRed(LAS_color);
int green = LASColor_GetGreen(LAS_color);
int blue = LASColor_GetBlue(LAS_color);
```

**PDAL:**

```cpp
int red = point_view->getFieldAs<int>(Id::Red, idx);
int green = point_view->getFieldAs<int>(Id::Green, idx);
int blue = point_view->getFieldAs<int>(Id::Blue, idx);
```

For LAS format, `hasColor()` method of `LasHeader` to see if the format supports RGB. However, in general, you can test use `hasDim(Id::Red), hasDim(Id::Green) and hasDim(Id::Blue)` method calls on the point, to see if the color was defined.

### Time

**libLAS:**

```cpp
double time = LASPoint_GetTime(LAS_point);
```

**PDAL:**

```cpp
double time = point_view->getFieldAs<double>(Id::GpsTime, idx);
```
Other point attributes

libLAS:

LASPoint_GetIntensity(LAS_point)
LASPoint_GetScanDirection(LAS_point)
LASPoint_GetFlightLineEdge(LAS_point)
LASPoint_GetScanAngleRank(LAS_point)
LASPoint_GetPointSourceId(LAS_point)
LASPoint_GetUserData(LAS_point)

PDAL:

point_view->getFieldAs<int>(Id::Intensity, idx)
point_view->getFieldAs<int>(Id::ScanDirectionFlag, idx)
point_view->getFieldAs<int>(Id::EdgeOfFlightLine, idx)
point_view->getFieldAs<int>(Id::ScanAngleRank, idx)
point_view->getFieldAs<int>(Id::PointSourceId, idx)
point_view->getFieldAs<int>(Id::UserData, idx)

Memory management

In libLAS C API, we need to explicitly take care of freeing the memory:

LASSRS_Destroy(LAS_srs);
LASHeader_Destroy(LAS_header);
LASReader_Destroy(LAS_reader);

When using C++ and PDAL, the objects created on stack free the memory when they go out of scope. When using smart pointers, they will take care of the memory they manage. This does not apply to special cases such as exit() function calls.

14.4 FAQ

• How do you pronounce PDAL?
  The proper spelling of the project name is PDAL, in uppercase. It is pronounced to rhyme with “GDAL”.

• Why do I get the error “Couldn’t create ... stage of type ...”?
  In almost all cases this error occurs because you’re trying to run a stage that is built as a plugin and the plugin (a shared library file or DLL) can’t be found by pdal. You can verify whether the plugin can be found by running pdal --drivers
If you’ve built pdal yourself, make sure you’ve requested to build the plugin in question (set BUILD_PLUGIN_TILEDB=ON, for example, in CMakeCache.txt).

If you’ve successfully built the plugin, a shared object called 

```
libpdal_plugin_<plugin type>_<plugin name>_<shared library extension>
```

should have been created that’s installed in a location where pdal can find it. pdal will search the following paths for plugins: 

```
.. /lib, .. /lib, .. /bin, .. /bin.
```

You can also override the default search path by setting the environment variable PDAL_DRIVER_PATH to a list of directories that pdal should search for plugins.

- Why do I get the error ```Unable to convert scaled value ... ```

This error usually occurs when writing LAS files, but can occur with other formats.

Simply, the output format you’ve chosen doesn’t support values as large (or small) as those that you’re trying to write. For example, if the output format specifies 32-bit signed integers, attempting to write a value larger than 2,147,483,647 will cause this error, as 2,147,483,647 is the largest 32-bit signed integer.

The LAS format always stores X, Y and Z values as 32-bit integers. You can specify a scale factor to be applied to those values in order to change their magnitude, but their precision is limited to 32 bits. If the value you’re attempting to write, when divided by the scale factor you’ve specified, is larger than 2,147,483,647, you will get this error. For example, if you attempt to write the value 6 with a scale factor of .000000001, you’ll get this error, as 6 / .000000001 is 6,000,000,000, which is larger than the maximum integer value.

- Why am I using 100GB of memory when trying to process a 10GB LAZ file?

If you’re performing an operation that is using standard mode (page 47), PDAL will read all points into memory at once. Compressed files, like LAZ, can decompress to much larger sizes before PDAL can process the data. Furthermore, some operations (notably DEM creation (page 119)) can use large amounts of additional memory during processing before the output can be written. Depending on the operation, PDAL will attempt operate in stream mode (page 47) to limit memory consumption when possible.

- What is PDAL’s relationship to PCL?

PDAL is PCL’s data translation cousin. PDAL is focused on providing a declarative pipeline syntax for orchestrating translation operations. PDAL also supports reading and writing PCL PCD files using readers.pcd (page 88) and writers.pcd (page 137).
See also:

PCL (page 7) describes PDAL and PCL’s relationship.

• What is PDAL’s relationship to libLAS?

The idea behind libLAS was limited to LIDAR data and basic manipulation. libLAS was also trying to be partially compatible with LASlib and LAStools. PDAL, on the other hand, aims to be a ultimate library and a set of tools for manipulating and processing point clouds and is easily extensible by its users. Howard Butler talked more about this history in a GeoHipster interview (http://geohipster.com/2018/03/05/howard-butler-like-good-song-open-source-software-chance-immortal/) in 2018.

• Are there any command line tools in PDAL similar to LAStools?

Yes. The pdal (page 25) command provides a wide range of features which go far beyond basic LIDAR data processing. Additionally, PDAL is licensed under an open source license (this applies to the whole library and all command line tools).

See also:

Applications (page 25) describes application operations you can achieve with PDAL.

• Is there any compatibility with libLAS’s LAS Utility Applications or LAStools?

No. The the command line interface was developed from scratch with focus on usability and readability. You will find that the pdal command has several well-organized subcommands such as info or translate (see Applications (page 25)).

• I get GeoTIFF errors. What can I do about them?

(Readers.las Error) Geotiff directory contains key 0 with short entry and more than one value.

If readers.las (page 71) is emitting error messages about GeoTIFF, this means the keys that were written into your file were incorrect or at least not readable by libgeotiff (https://trac.osgeo.org/geotiff). Rewrite the file using PDAL to fix the issue:

```
pdal translate badfile.las goodfile.las --writers.las.forward=all
```
14.5 License

Unless otherwise indicated, all files in the PDAL distribution are

Copyright (c) 2019, Hobu, Inc. (howard@hobu.co)

and are released under the terms of the BSD open source license.

This file contains the license terms of all files within PDAL.

14.5.1 Overall PDAL license (BSD)

Copyright (c) 2019, Hobu, Inc. (howard@hobu.co)

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of Hobu, Inc. or Flaxen Consulting LLC nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
14.6 References

14.6.1 Citation

To cite PDAL in publications use:

PDAL Contributors, 2018. PDAL Point Data Abstraction Library.

A BibTeX entry for LaTeX users is

@misc{pdal_contributors_2018_2556738,
    author = {PDAL Contributors},
    title = {PDAL Point Data Abstraction Library},
    month = nov,
    year = 2018,
    doi = {10.5281/zenodo.2556738},
    url = {https://doi.org/10.5281/zenodo.2556738}
}

14.6.2 Reference
CHAPTER

FIFTEEN

INDICES AND TABLES

• genindex
• search
[LLW+19] missing journal in Xuechen2019


INDEX

A
Apps, 10

B
Bindings, 299
boundary, 370

C
capstone, 436
Citation, 585
classification, 398, 415, 418
classifications, 385
Clipping, 374
CloudCompare, 8
Colorization, 380
Command line, 10
Compile, 303
Conda, 17, 297, 356
coordinate system, 358
csd, 430
CSV, 358

D
Denoising, 384
Density, 387
density, 392
DSM, 404
DTM, 404

E
elevation model, 404
Embed, 295
EPT, 367
Extension, 296

F
filtering, 398, 415
Fusion, 8

G
GDAL, 380
georeferencing, 352, 430
GeoWave, 63, 122
GNSS/IMU, 352, 430
ground, 398, 415

H
hexagon tessellation, 387
histogram, 430

I
info command, 357
Install, 297, 302
installation, 357
intensity, 418

J
Java, 299, 302, 303
JNI, 299
JSON, 358
Julia, 11

L
LAStools, 7
libLAS, 8

M
matplotlib, 430
metadata, 358
nearby, 361
nearest, 361
Numpy, 11, 295, 430

OGR, 370, 374, 387
Optech, 430
OrfeoToolbox, 8
outliers, 384

PCL, 7
pdal::BOX2D (C++ class), 519
pdal::BOX2D::BOX2D (C++ function), 520
pdal::BOX2D::clear (C++ function), 520
pdal::BOX2D::clip (C++ function), 520
pdal::BOX2D::contains (C++ function), 520, 521
pdal::BOX2D::empty (C++ function), 520
pdal::BOX2D::equal (C++ function), 521
pdal::BOX2D::error (C++ struct), 523
pdal::BOX2D::error::error (C++ function), 523
pdal::BOX2D::getDefaultSpatialExtent (C++ function), 523
pdal::BOX2D::grow (C++ function), 521
pdal::BOX2D::maxx (C++ member), 522
pdal::BOX2D::maxy (C++ member), 522
pdal::BOX2D::minx (C++ member), 522
pdal::BOX2D::miny (C++ member), 522
pdal::BOX2D::operator!= (C++ function), 521
pdal::BOX2D::operator== (C++ function), 521
pdal::BOX2D::overlaps (C++ function), 521
pdal::BOX2D::parse (C++ function), 522
pdal::BOX2D::toBox (C++ function), 522
pdal::BOX2D::toGeoJSON (C++ function), 522
pdal::BOX2D::toWKT (C++ function), 522
pdal::BOX2D::valid (C++ function), 520
pdal::BOX3D (C++ class), 523
pdal::BOX3D::BOX3D (C++ function), 523
pdal::BOX3D::clear (C++ function), 524
pdal::BOX3D::clip (C++ function), 525
pdal::BOX3D::contains (C++ function), 524
pdal::BOX3D::empty (C++ function), 523
pdal::BOX3D::equal (C++ function), 524
pdal::BOX3D::error (C++ struct), 526
pdal::BOX3D::error::error (C++ function), 527
pdal::BOX3D::getDefaultSpatialExtent (C++ function), 526
pdal::BOX3D::grow (C++ function), 524, 525
pdal::BOX3D::maxx (C++ member), 526
pdal::BOX3D::maxy (C++ member), 526
pdal::BOX3D::maxz (C++ member), 526
pdal::BOX3D::minx (C++ member), 526
pdal::BOX3D::miny (C++ member), 526
pdal::BOX3D::minz (C++ member), 526
pdal::BOX3D::operator!= (C++ function), 525
pdal::BOX3D::operator== (C++ function), 523
pdal::BOX3D::operator== (C++ function), 524
pdal::BOX3D::overlaps (C++ function), 525
pdal::BOX3D::parse (C++ function), 526
pdal::BOX3D::to2d (C++ function), 525
pdal::BOX3D::toBox (C++ function), 522
pdal::Box3D::toWKT (C++ function), 525
pdal::Box3D::valid (C++ function), 524
pdal::Charbuf (C++ class), 527
pdal::Charbuf::Charbuf (C++ function), 527
pdal::Dimension (C++ type), 528
pdal::Dimension::base (C++ function), 529
pdal::Dimension::BaseType (C++ enum), 528
pdal::Dimension::BaseType::Floating (C++ enumerator), 528
pdal::Dimension::BaseType::None (C++ enumerator), 528
pdal::Dimension::BaseType::Signed (C++ enumerator), 528
pdal::Dimension::BaseType::Unsigned (C++ enumerator), 528
pdal::Dimension::COUNT (C++ member), 530
pdal::Dimension::Detail (C++ class), 530
pdal::Dimension::DetailList (C++ type), 528
pdal::Dimension::extractName (C++ function), 529
pdal::Dimension::fixName (C++ function), 529
pdal::Dimension::fromName (C++ function), 529
pdal::Dimension::interpretationName (C++ function), 529
pdal::Dimension::toName (C++ function), 529
pdal::Dimension::Type (C++ enum), 528
pdal::Dimension::type (C++ function), 529
pdal::Dimension::Type::Double (C++ enumerator), 528
pdal::Dimension::Type::Float (C++ enumerator), 528
pdal::Dimension::Type::None (C++ enumerator), 528
pdal::Dimension::Type::Signed16 (C++ enumerator), 528
pdal::Dimension::Type::Signed32 (C++ enumerator), 528
pdal::Dimension::Type::Signed64 (C++ enumerator), 528
pdal::Dimension::Type::Signed8 (C++ enumerator), 528
pdal::Dimension::Type::Unsigned16 (C++ enumerator), 528
pdal::Dimension::Type::Unsigned32 (C++ enumerator), 528
pdal::Dimension::Type::Unsigned64 (C++ enumerator), 528
pdal::Dimension::Type::Unsigned8 (C++ enumerator), 528
pdal::Extractor (C++ class), 530
pdal::Extractor::Extractor (C++ function), 530
pdal::Extractor::get (C++ function), 530, 531
pdal::Extractor::good (C++ function), 530
pdal::Extractor::operator bool (C++ function), 530
pdal::Extractor::operator>> (C++ function), 531, 532
pdal::Extractor::position (C++ function), 530
pdal::Extractor::seek (C++ function), 530
pdal::Extractor::skip (C++ function), 530
function), 545
pdal::Options::getValues (C++ function), 545
pdal::Options::Options (C++ function), 544
pdal::Options::remove (C++ function), 544
pdal::Options::replace (C++ function), 544, 545
pdal::Options::toCommandLine (C++ function), 545
pdal::Options::toMetadata (C++ function), 544
pdal::PointView (C++ class), 545
pdal::PointView::~PointView (C++ function), 546
pdal::PointView::append (C++ function), 546
pdal::PointView::appendPoint (C++ function), 546
pdal::PointView::begin (C++ function), 546
pdal::PointView::build2dIndex (C++ function), 548
pdal::PointView::build3dIndex (C++ function), 548
pdal::PointView::calculateBounds (C++ function), 546, 547
pdal::PointView::clearTemps (C++ function), 548
pdal::PointView::compare (C++ function), 546
pdal::PointView::createMesh (C++ function), 548
pdal::PointView::createRaster (C++ function), 548
pdal::PointView::dimName (C++ function), 547
pdal::PointView::dims (C++ function), 547
pdal::PointView::dimSize (C++ function), 547
pdal::PointView::dimType (C++ function), 547
pdal::PointView::dimTypes (C++ function), 547
pdal::PointView::dump (C++ function), 547
pdal::PointView::empty (C++ function), 546
pdal::PointView::end (C++ function), 546
pdal::PointView::getField (C++ function), 546
pdal::PointView::getOrAddPoint (C++ function), 548
pdal::PointView::getPackedPoint (C++ function), 547
pdal::PointView::getPoint (C++ function), 548
pdal::PointView::hasDim (C++ function), 547
pdal::PointView::id (C++ function), 546
pdal::PointView::invalidateProducts (C++ function), 548
pdal::PointView::layout (C++ function), 547
pdal::PointView::makeNew (C++ function), 546
pdal::PointView::mesh (C++ function), 548
pdal::PointView::operator= (C++ function), 546
pdal::PointView::point (C++ function), 546
pdal::PointView::pointSize (C++ function), 546
pdal::PointView::PointView (C++ function), 546
pdal::PointView::raster (C++ function), 548
pdal::PointView::setField (C++ function), 546
pdal::PointView::setPackedPoint (C++ function), 547
pdal::PointView::size (C++ function), 546
pdal::PointView::spatialReference (C++ function), 547
pdal::PointView::table (C++ function), 547
pdal::PointView::toMetadata (C++ function), 548
pdal::ProgramArgs (C++ class), 549
pdal::ProgramArgs::add (C++ function), 549–551
pdal::ProgramArgs::addSynonym (C++ function), 551
pdal::ProgramArgs::commandLine (C++ function), 551
pdal::ProgramArgs::dump (C++ function), 551
pdal::ProgramArgs::dump2 (C++ function), 552
pdal::ProgramArgs::dump3 (C++ function), 552
pdal::ProgramArgs::parse (C++ function), 551
pdal::ProgramArgs::parseSimple (C++ function), 551
pdal::ProgramArgs::reset (C++ function), 551
pdal::ProgramArgs::set (C++ function), 550
pdal::Reader (C++ class), 552
pdal::Stage (C++ class), 553
pdal::Stage::~Stage (C++ function), 553
pdal::Stage::addAllArgs (C++ function), 555
pdal::Stage::addConditionalOptions (C++ function), 554
pdal::Stage::addOptions (C++ function), 555
pdal::Stage::execute (C++ function), 553, 554
pdal::Stage::findNonstreamable (C++ function), 554
pdal::Stage::getInputs (C++ function), 555
pdal::Stage::getMetadata (C++ function), 556
pdal::Stage::getSpatialReference (C++ function), 554
pdal::Stage::isDebug (C++ function), 555
pdal::Stage::parseName (C++ function), 556
pdal::Stage::parseTagName (C++ function), 556
pdal::Stage::pipelineStreamable (C++ function), 554
pdal::Stage::prepare (C++ function), 553
pdal::Stage::preview (C++ function), 553
pdal::Stage::removeOptions (C++ function), 555
pdal::Stage::serialize (C++ function), 556
pdal::Stage::setInput (C++ function), 553
pdal::Stage::setLog (C++ function), 555
pdal::Stage::setOptions (C++ function), 554
pdal::Stage::setProgressFd (C++ function), 553
pdal::Stage::setSpatialReference (C++ function), 554
pdal::Stage::setTag (C++ function), 555
pdal::Stage::Stage (C++ function), 553
pdal::Stage::startLogging (C++ function), 555
pdal::Stage::stopLogging (C++ function), 555
pdal::Stage::tag (C++ function), 555
pdal::StageFactory (C++ class), 557
pdal::StageFactory::createStage (C++ function), 557
pdal::StageFactory::destroyStage (C++ function), 557

Index 597
pdal::StageFactory::inferReaderDriver (C++ function), 557
pdal::StageFactory::inferWriterDriver (C++ function), 558
pdal::StageFactory::StageFactory (C++ function), 557
pdal::Utils (C++ type), 558
pdal::Utils::BacktraceEntries (C++ type), 558
pdal::Utils::BacktraceEntry (C++ struct), 573
pdal::Utils::BacktraceEntry::addr (C++ member), 574
pdal::Utils::BacktraceEntry::BacktraceEntry (C++ function), 559
pdal::Utils::BacktraceEntry::libname (C++ member), 574
pdal::Utils::BacktraceEntry::offset (C++ member), 574
pdal::Utils::BacktraceEntry::symname (C++ member), 574
pdal::Utils::backtraceImpl (C++ function), 561
pdal::Utils::base64_encode (C++ function), 564
pdal::Utils::cksum (C++ function), 562
pdal::Utils::closeFile (C++ function), 559
pdal::Utils::closeProgress (C++ function), 559
pdal::Utils::computeHausdorff (C++ function), 559
pdal::Utils::contains (C++ function), 560
pdal::Utils::createFile (C++ function), 558
pdal::Utils::dirSeparator (C++ member), 573
pdal::Utils::dllDir (C++ function), 559
pdal::Utils::dynamicLibExtension (C++ member), 573
pdal::Utils::endsWith (C++ function), 562
pdal::Utils::extractDim (C++ function), 559
pdal::Utils::fetchRemote (C++ function), 559
pdal::Utils::fileExists (C++ function), 557
pdal::Utils (C++ type), 558
pdal::Utils::fromString (C++ function), 560, 572, 573
pdal::Utils::iequals (C++ function), 562
pdal::Utils::inRange (C++ function), 569
pdal::Utils::insertDim (C++ function), 559
pdal::Utils::isRemote (C++ function), 558
pdal::Utils::numericCast (C++ function), 569, 570
pdal::Utils::openFile (C++ function), 559
pdal::Utils::openProgress (C++ function), 559
pdal::Utils::pathListSeparator (C++ member), 573
pdal::Utils::printError (C++ function), 559
pdal::Utils::redirect (C++ function), 568, 569
pdal::Utils::RedirectStream (C++ struct), 574
pdal::Utils::RedirectStream::m_buf (C++ member), 575
pdal::Utils::RedirectStream::m_null (C++ member), 575
pdal::Utils::RedirectStream::m_out (C++ member), 575
pdal::Utils::remove (C++ function), 560
pdal::Utils::remove_if (C++ function), 560
pdal::Utils::restore (C++ function), 569
pdal::Utils::sround (C++ function), 598
PDAL: Point cloud Data Abstraction Library, Release 2.2.0

pdal::Utils::startsWith (C++ function), 562
pdal::Utils::StatusWithReason (C++ class), 574
pdal::Utils::StatusWithReason::code (C++ function), 574
pdal::Utils::StatusWithReason::operator bool (C++ function), 574
pdal::Utils::StatusWithReason (C++ function), 574
pdal::Utils::StatusWithReason::what (C++ function), 574
pdal::Utils::toDouble (C++ function), 559
pdal::Utils::toJSON (C++ function), 558
pdal::Utils::tolower (C++ function), 561
pdal::Utils::toMetadata (C++ function), 559
pdal::Utils::toNative (C++ function), 573
pdal::Utils::toString (C++ function), 570, 571
pdal::Utils::toupper (C++ function), 562
pdal::Utils::trim (C++ function), 563
pdal::Utils::typeidName (C++ function), 568
pdal::Utils::writeProgress (C++ function), 559
pdal::Writer (C++ class), 575
pip, 297
poisson, 392
Potree, 367
project, 436
pronounce, 581
Python, 11, 295–297, 430

R
range filter, 385
Raster, 380
rasterization, 418
References, 585
Reprojection, 364
RGB, 380
RIEGL, 430

S
sample, 392
Scala, 299, 302, 303
search, 361
SOCS, 352
software installation, 356
Source, 297
spatial reference system, 358
Stage, 552
Start Here, 357

T
thinning, 392

U
Utils, 558
UTM, 364, 430

V
Vector, 374
voxel sampling, 392

W
web services, 367
WGS84, 364, 430

Q
QGIS, 370
query, 361
Quickstart, 17

Index 599